Ausdrucksfähigkeit und effiziente Implementierbarkeit formaler Beschreibungstechniken am Beispiel von Estelle

Dipl.-Inf. Joachim Thees

Vom Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)
genehmigte Dissertation

Kaiserslautern, den 17. Februar 2005

D 386

Fachbereich Informatik
Technische Universität Kaiserslautern

Dekan: Prof. Dr. Hans Hagen

Berichterstatter: Prof. Dr. Reinhard Gotzhein
Prof. Dr. Jens B. Schmitt
Vorsitzender der
Promotionskommission: Prof. Dr. Otto Mayer

Datum der Einreichung: 19. November 2004
Datum der Aussprache: 17. Februar 2005

Danksagung

Diese Arbeit ist während meiner Tätigkeit als Mitarbeiter in der Arbeitsgruppe Rechnernetze von Herrn Prof. Reinhard Gotzhein (unter zeitweiliger Förderung durch die DFG) und später während meiner Tätigkeit als Leiter des Service-Center Informatik (SCI) an der TU Kaiserslautern entstanden. Zu ihrem Gelingen haben viele Menschen beigetragen, die mich während der Jahre meiner Arbeit an dieser Thematik in vielerlei Hinsicht unterstützt haben.

Mein besonderer Dank gilt Herrn Prof. Reinhard Gotzhein für die langjährige inhaltliche Betreuung und Unterstützung meiner Forschungsarbeiten, die große Geduld, die forscherischen Freiheiten und das Vertrauen, das er mir immer entgegengebracht hat. Ohne seine breite Unterstützung wären meine Forschungstätigkeiten sicher nicht in dieser Intensität und über so lange Zeit möglich gewesen.

Herr Prof. Jens Schmitt danke ich dafür, dass er sich spontan als Gutachter zur Verfügung gestellt hat, ohne den Umfang der ihm aufgebürdeten Arbeit vorher zu kennen.

Für ihre Diskussions- und Tool-Beiträge bei der Entwicklung des XEC-Toolkits danke ich Herrn Hartmuth Penner, Herrn Stefan Dahl und Herrn Michael Wenz. Auch möchte ich allen Beta-Testern von XEC – allen voran Herrn Luc Hohwiller – für ihr konstruktives Feedback und die damit verbundene Motivation weiterzumachen danken.

Für die fachliche, organisatorische und technische Unterstützung möchte ich meinen Freunden und Kollegen aus der Arbeitsgruppe Rechnernetze, dem SCI und dem ganzen Fachbereich danken. Mein besonderer Dank gilt Herrn Jan Bredereke und Herrn Christian Peper für viele ausführliche und fruchtbare Diskussionen bei der einen oder anderen Tasse Tee.

Für das (zum Teil vielfache) aufmerksame Durchsehen des Textes und die vielen kleinen und großen Hinweise danke ich besonders Frau Brigitte Schlachter, Herrn Dirk Barthel und Herrn Hans-Joachim Schmitt. Ich entschuldige mich nochmals für den Umfang der Arbeit (*„Sorry, ist etwas länger geworden ...“*).

Ich möchte meinen Eltern dafür danken, dass sie mir das Studium der Informatik ermöglicht und mich auf meinem Weg immer unterstützt haben. Meiner Schwester danke ich besonders für die gelegentliche Erinnerung daran, was eigentlich im Leben zählt.

Schließlich möchte ich allen Freunden und Kollegen danken, die mir in den vergangenen Jahren über das eine oder andere Motivationstief hinweggeholfen haben (und sei es auch durch permanentes Nachfragen, wann es denn endlich soweit sei).

Den größten Dank schulde ich jedoch Brigitte Schlachter, die mir über all die Jahre die Kraft und die Zuversicht gegeben hat, nicht aufzugeben, auch wenn ich manchmal kein Licht mehr im Dunkel erkennen konnte. Ohne Dich wäre es nicht gegangen! Dir möchte ich diese Arbeit widmen.

Kaiserslautern, Februar 2005 Joachim Thees

Zusammenfassung

Die formale Spezifikation von Kommunikationssystemen stellt durch die mit ihr verbundene *Abstraktion* und *Präzision* eine wichtige Grundlage für die formale Verifikation von Systemeigenschaften dar. Diese Abstraktion begrenzt jedoch auch die *Ausdrucksfähigkeit* der formalen Beschreibungstechnik und kann somit zu problemunangemessenen Spezifikationen führen.

Wir untersuchen anhand der formalen Beschreibungstechnik *Estelle* zunächst zwei solche Aspekte. Beide führen speziell in Hinsicht auf die Domäne von Estelle, der Spezifikation von Kommunikationsprotokollen, zu schwerwiegenden Beeinträchtigungen der Ausdrucksfähigkeit.

Eines dieser Defizite zeigt sich bei dem Versuch, in Estelle ein *offenes System* wie z. B. eine Protokollmaschine oder einen Kommunikationsdienst zu spezifizieren. Da Estelle-Spezifikationen nur geschlossene Systeme beschreiben können, werden solche Komponenten immer nur als Teil einer fest vorgegebenen Umgebung spezifiziert und besitzen auch nur in dieser eine formale Syntax und Semantik. Als Lösung für dieses Problem führen wir die kompatible syntaktische und semantische Estelle-Erweiterung *Open-Estelle* ein, die eine formale Spezifikation solcher offener Systeme und ihres Imports in verschiedene Umgebungen ermöglicht.

Ein anderes Defizit in der Ausdrucksfähigkeit von Estelle ergibt sich aus der strengen Typprüfung. Wir werden zeigen, dass es in heterogenen, hierarchisch strukturierten Kommunikationssystemen im Zusammenhang mit den dort auftretenden *horizontalen* und *vertikalen Typkompositionen* zu einer unangemessenen Modellierung von Nutzdatentypen an den Dienstschnittstellen kommt. Dieses Problem erweist sich beim Versuch einer generischen und nutzdatentypunabhängigen Spezifikation eines offenen Systems (z. B. mit Open-Estelle) sogar als fatal. Estelle erlaubt auch hier keine problemangemessene Modellierung solcher Fragestellungen. Deshalb führen wir die kompatible *Containertyp-Erweiterung* ein, durch die eine formale Spezifikation nutzdatentypunabhängiger und somit *generischer Schnittstellen* von Diensten und Protokollmaschinen ermöglicht wird.

Als Grundlage für unsere Implementierungs- und Optimierungsexperimente führen wir den „*eXperimental Estelle Compiler*“ (XEC) ein. Er ermöglicht aufgrund seines Implementierungskonzeptes eine sehr flexible Modellierung des Systemmanagements und ist insbesondere für die Realisierung verschiedener Auswahloptimierungen geeignet. XEC ist zudem mit verschiedenen Statistik- und Monitoring-Funktionalitäten ausgestattet, durch die eine effiziente quantitative Analyse der durchgeführten Implementierungsexperimente möglich ist. Neben dem vollständigen Sprachumfang von Estelle unterstützt XEC auch die meisten der hier eingeführten Estelle-Erweiterungen.

Neben der Korrektheit ist die *Effizienz* automatisch generierter Implementierungen eine wichtige Anforderung im praktischen Einsatz. Hier zeigt sich jedoch, dass viele der in formalen Protokollspezifikationen verwendeten Konstrukte nur schwer semantikkonform und zugleich effizient implementiert werden können. Entsprechend untersuchen wir anhand des Kontrollflusses und der Handhabung von Nutzdaten, wie die spezifizierten Operationen effizient implementiert werden können, ohne das Abstraktionsniveau senken zu müssen.

Die Optimierung des Kontrollflusses geschieht dabei ausgehend von der effizienten Realisierung der Basisoperationen der von XEC erzeugten Implementierungen primär anhand der Transitionsauswahl, da diese speziell bei komplexen Spezifikationen einen erheblichen Teil der Ausführungszeit beansprucht. Wir entwickeln dazu verschiedene heuristische Optimierungen der globalen Auswahl und der modullokalen Auswahl und werten diese sowohl analytisch wie auch experimentell aus. Wesentliche Ansatzpunkte sind dabei verschiedene ereignisgesteuerte Auswahlverfahren auf globaler Ebene und die Reduktion der zu untersuchenden Transitionen auf lokaler Ebene. Einige dieser Verfahren werden durch den Einsatz bestimmter Spezifikationsstilregeln oder Estelle-Erweiterungen wie die *Independent-Module-Erweiterung* in ihrer Wirksamkeit deutlich verbessert. Dabei können bzgl. der Anzahl der getesteten Module und Transitionen Leistungssteigerungen um ganze Größenordnungen erreicht werden. Die Überprüfung der Ergebnisse anhand der ausführungszeitbezogenen Leistungsbewertung bestätigt diese Ergebnisse.

Hinsichtlich der effizienten Handhabung von Daten untersuchen wir unterschiedliche Ansätze auf verschiedenen Ebenen, die jedoch in den meisten Fällen eine problemunangemessene Ausrichtung der Spezifikation auf die effiziente Datenübertragung erfordern. Eine überraschend elegante, problemorientierte und effiziente Lösung ergibt sich jedoch auf Basis der *Containertyp-Erweiterung*, die ursprünglich zur Steigerung des Abstraktionsniveaus eingeführt wurde. Dieses Ergebnis widerlegt die Vorstellung, dass Maßnahmen zur Steigerung der effizienten Implementierbarkeit auch immer durch eine Senkung des Abstraktionsniveaus erkauft werden müssen.

Inhalt

Abbildungsverzeichnis

Beispielverzeichnis

1. Einleitung

Formale Beschreibungstechniken[1] spielen bei der Spezifikation von Kommunikationsprotokollen eine wichtige Rolle, da die mit ihnen verbundene *Abstraktion* und *Präzision* die Grundlagen für die Validierung und Implementierung von Kommunikationssystemen darstellen.

Die *Korrektheit* solcher Implementierungen im Sinne der Einhaltung der formalen Semantik des beschriebenen Systems ist dabei das zentrale Bindeglied zwischen den formal nachgewiesenen Eigenschaften der *Spezifikation* und den Eigenschaften der *ausführbaren Implementierungen*. Insbesondere garantiert sie die Übertragbarkeit der anhand der formalen Spezifikation nachgewiesenen Eigenschaften auf jede korrekte Implementierung.

Aufgrund der mit dem Implementierungsvorgang verbundenen *Senkung des Abstraktionsniveaus* und der *Steigerung des Detaillierungsgrades* sind solche Implementierungen einer formalen Analyse meist kaum mehr zugänglich. Dies gilt besonders für komplexe Spezifikationen und deren Implementierungen. Somit stellt die Übertragbarkeit der anhand der formalen Spezifikation nachgewiesenen Eigenschaften oft den einzigen Weg zum (im mathematischen Sinne) formalen Nachweis von Eigenschaften der Implementierung dar.

Die Gewinnung von ausführbaren Implementierungen aus formalen Spezifikationen gewinnt insbesondere erheblich an Attraktivität, wenn der Implementierungsvorgang über weite Strecken oder sogar vollständig *automatisiert* werden kann. Der Nachweis der Korrektheit der Implementierung kann dabei auf den Nachweis der Korrektheit des Implementierungsverfahrens und der verwendeten Werkzeuge zurückgeführt werden. Insbesondere bei ausreichend semantiknahen Implementierungsverfahren kann dieser Nachweis im Sinne einer „offensichtlichen Korrektheit“ tatsächlich geführt werden und bildet damit die Möglichkeit zur zuverlässigen Gewinnung korrekter Implementierungen.

Dem praktischen Einsatz von formalen Beschreibungstechniken und den Methoden und Werkzeugen zur automatischen Gewinnung von ausführbaren Implementierungen stehen jedoch auch Nachteile entgegen. Diese beruhen auf Defiziten bei

- der *Ausdrucksfähigkeit* der formalen Beschreibungstechniken und
- der *Korrektheit* und *Effizienz* der automatisch gewonnenen Implementierungen.

Die *Beschränkung der Ausdrucksfähigkeit* stellt prinzipiell eine der wesentlichen Grundlagen der Abstraktion formaler Beschreibungstechniken dar, indem von nicht relevanten Aspekten abstrahiert wird. Dies sind u. a. plattform- oder implementierungsspezifische Details wie z. B. die Realisierung eines Kontrollzustandes eines endlichen Automaten. Diese Form der Abstraktion ist insbesondere eine essentielle Voraussetzung für formale Eigenschaftsbeweise und unterscheidet eine formale Beschreibung gerade von den in dieser Hinsicht nur wenig abstrakten ausführbaren Implementierungen.

1. „*Formal Description Techniques*“, FDTs

Teilweise bewirkt die Beschränkung der Ausdrucksfähigkeit formaler Beschreibungstechniken jedoch auch eine *Senkung des Abstraktionsniveaus*, wenn z. B. *offene Systeme* oder *generische Schnittstellen* nicht angemessen beschrieben werden können. Die Identifikation und Beseitigung solcher Beschränkungen vor dem Hintergrund der Spezifikation von Kommunikationssystemen ist eines der Hauptziele dieser Arbeit.

Das zweite Hauptziel betrifft die *automatische Gewinnung korrekter effizienter Implementierungen* aus formalen Spezifikationen. Wie wir bereits dargestellt haben, bietet eine semantiknahe und „offensichtlich korrekte" Implementierungsmethode einen effektiven Ansatz zur Gewinnung korrekter Implementierungen. Gerade diese Semantiknähe verursacht jedoch häufig eine beschränkte Effizienz der gewonnenen Implementierungen. Erschwerend kommt hinzu, dass durch das hohe Abstraktionsniveau der formalen Beschreibungstechniken in der Spezifikation häufig Strukturen und Mechanismen vorgegeben werden, die die Effizienz einer semantiknahen Implementierung beeinträchtigen können. Beispiele sind hier die Beschränkung auf Nachrichtenkommunikation zwischen eng gekoppelten Teilsystemen, eine Copy-Semantik bei der Datenübertragung oder eine aufwändige Synchronisation zwischen Teilsystemen. Einige Effizienzprobleme können dabei durch eine geschickte *Optimierung* des Implementierungsverfahrens gelöst werden. Andere erfordern *Erweiterungen der Ausdrucksfähigkeit* der formalen Beschreibungstechnik oder die Anwendung bestimmter *Spezifikationsstile*, um bereits auf Spezifikationsebene die Verwendung von Konstrukten und Konstellationen zu vermeiden, die nicht effizient und zugleich semantikkonform implementierbar sind.

Wir führen unsere Untersuchungen exemplarisch anhand der formalen Beschreibungstechnik *Estelle* [ISO97] durch. Estelle wurde zur Spezifikation von Kommunikationssystemen entwickelt und bietet aufgrund ihres semantiknah implementierbaren Typ- und Ausführungsmodells gute Grundlagen für eine automatische Implementierung im obigen Sinne. Wir werden sehen, dass viele der hier identifizierten Probleme auch in anderen formalen Beschreibungstechniken wie z. B. SDL[2] vorliegen und die entwickelten Lösungskonzepte ebenfalls übernommen werden können.

Wir beginnen in Kapitel 2 mit einem kurzen Überblick über Estelle und diskutieren dann mögliche *Zielsetzungen und Methoden der Optimierung* von Protokollimplementierungen. Auf dieser Basis stellen wir dann verschiedene Benchmarkspezifikationen zur späteren *quantitativen Evaluierung* von Implementierungstechniken vor.

In Kapitel 3 führen wir als eines der wesentlichen Ergebnisse dieser Arbeit den Estelle-Implementierungsgenerator *XEC* („*eXperimental Estelle Compiler*") ein, den wir eigens als Experimentierplattform für unsere Implementierungs- und Optimierungsexperimente entwickelt haben. Dieser ermöglicht aufgrund seines hohen Abstraktionsniveaus die Erprobung unterschiedlichster Optimierungstechniken unter Einhaltung der oben geforderten „offensichtlichen Korrektheit". XEC beinhaltet insbesondere auch verschiedene Mechanismen zur *Leistungsbewertung* der generierten Implementierungen und implementiert mehrere der im weiteren Verlauf vorgestellten Estelle-Erweiterungen.

Die automatische Gewinnung *effizienter Implementierungen* von Protokollspezifikationen auf Basis von XEC ist dann Gegenstand von Kapitel 4. Wir konzentrieren uns dabei auf die Optimierung der Kontrollflusssteuerung des generierten Systems. Zur quantitativen Evaluierung der hier entwickelten Optimierungstechniken wenden wir die in Kapitel 2 vorgestellten Benchmarkspezifikationen an, wobei wir die übertragenen Nutzdatenmengen minimieren um den mit ihrer Handhabung verbundenen Aufwand aus der Betrachtung weitgehend auszuschließen.

---

2. „Specification and Description Language", [ITU94]

Diese Handhabung von Daten in formalen Spezifikationen und ihren Implementierungen ist dann Gegenstand der beiden folgenden Kapitel.

Wir beginnen dazu in Kapitel 5 mit einer grundlegenden Analyse der *Abstraktion von Datentypen* in hierarchischen Protokollspezifikationen. Dabei zeigen sich erhebliche Defizite in der Ausdrucksfähigkeit von Estelle, die insbesondere bei der Spezifikation großer heterogener Systeme oder *generischer und offener Protokollkomponenten* (siehe auch Kapitel 7) zu Tage treten. Als wesentliches Ergebnis dieses Kapitels entwickeln wir dann die *Containertyp-Erweiterung* von Estelle und diskutieren Anwendungs- und Implementierungsaspekte.

In Kapitel 6 beschäftigen wir uns dann mit der *effizienten Übertragung von Daten* in automatisch generierten Estelle-Implementierungen. Wir betrachten dazu die syntaktischen und semantischen Randbedingungen des Datentransports zwischen Modulinstanzen. Ausgehend von einigen in Handimplementierungen vorzufindenden Optimierungstechniken entwickeln und evaluieren wir dann verschiedene Techniken zur Optimierung dieses Aspekts in automatisch erstellten Estelle-Implementierungen. Dabei kommen auch Estelle-Erweiterungen wie die bereits im vorangegangenen Kapitel vorgestellte Containertyp-Erweiterung zum Einsatz.

Die in Kapitel 7 eingeführte syntaktische und semantische Estelle-Erweiterung *Open-Estelle* ermöglicht die Spezifikation (topologisch) offener Systeme als eigenständige Spezifikationen sowie ihren formalen Import in andere Umgebungen. Wir zeigen in diesem Kapitel, wie Open-Estelle insbesondere in Kombination mit der in Kapitel 5 eingeführten Containertyp-Erweiterung die formale Wiederverwendung generischer Protokollkomponenten ermöglicht. In diesem Zusammenhang diskutieren wir auch alternative *kompositionsverträgliche* Semantikansätze für Estelle und Open-Estelle.

Schließlich fassen wir in Kapitel 8 die Ergebnisse unserer Untersuchungen zusammen und geben einen Ausblick auf mögliche künftige Forschungsziele.

2. Grundlagen

In diesem Kapitel geben wir zunächst einen kurzen Überblick über die formale Beschreibungstechnik *Estelle* (Abschnitt 2.1), anhand derer wir in den nachfolgenden Kapiteln unsere Untersuchungen zur Ausdrucksfähigkeit und effizienten Implementierbarkeit exemplarisch führen. Wir werden später an verschiedenen Stellen entsprechend auch die Übertragbarkeit unserer Ansätze auf andere formale Beschreibungstechniken wie z. B. SDL diskutieren.

In Abschnitt 2.2 identifizieren wir mögliche *Zielsetzungen* und *Methoden der Optimierung* von Protokollimplementierungen.

Die *quantitative Evaluierung* von Implementierungs- und Optimierungstechniken ist Gegenstand von Abschnitt 2.3.

2.1. Die FDT Estelle

Estelle [DeBu89] ist eine *formale Beschreibungstechnik* („formal description technique“, FDT), die seit 1989 international genormt ist ([ISO89], [ISO97][1]). Sie basiert auf ISO-Pascal und wurde zur Beschreibung von Kommunikationsprotokollen entwickelt. Estelle-Spezifikationen sind *formal* in dem Sinne, dass sie eine formale Syntax und eine formale Semantik besitzen. Dies erlaubt eine präzise Beschreibung des spezifizierten Systems und ist eine Voraussetzung für den Nachweis funktionaler und nicht-funktionaler Eigenschaften.

2.1.1 Die Struktur von Estelle-Spezifikationen

Eine Estelle-Spezifikation beschreibt ein dynamisch modifizierbares hierarchisches System von miteinander kommunizierenden erweiterten endlichen Automaten (EFSMs), den *Modulinstanzen* (siehe Abb. 2-1). Die Erzeugung und Freigabe von Modulinstanzen wird lokal durch die jeweilige Vatermodulinstanz kontrolliert. Die oberste Modulinstanz (die Instanz des Spezifikationsmoduls) wird dabei implizit zu Beginn der Ausführung der Spezifikation erzeugt.

1. Bei [ISO97] handelt es sich um eine überarbeitete Version von [ISO89], die sich inhaltlich im Wesentlichen nur durch einige Fehlerkorrekturen und erweiterte Beispiele unterscheidet.

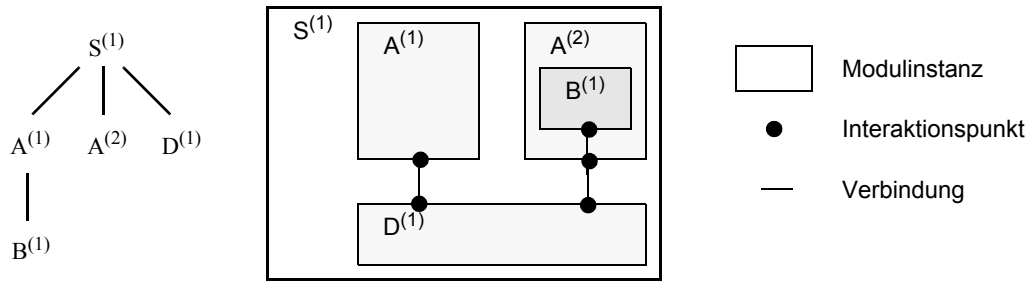


Abbildung 2-1: Dynamische Modulinstanz- und Verbindungshierarchie

Die Modulinstanzen werden jeweils durch die dynamische Instanziierung von *Modulen*[2] erzeugt, welche ausgehend vom Spezifikationsmodul in einer (statischen) Hierarchie die syntaktische Grundstruktur einer Estelle-Spezifikation bildet. So ist das im folgenden Beispiel und in Abb. 2-2 angegebene Spezifikationsgerüst die Grundlage für die in Abb. 2-1 dargestellte (dynamische) Modulinstanz- und Verbindungshierarchie.

Beispiel 2.1: Grundstruktur einer Estelle-Spezifikation

```
specification S;    {...}
     body A {...}
          body B {...}  end;
          body C {...}  end;
     end; {body A}
     body D {...}
     end;
end. {specification S}
```

(Ende von Beispiel 2.1)

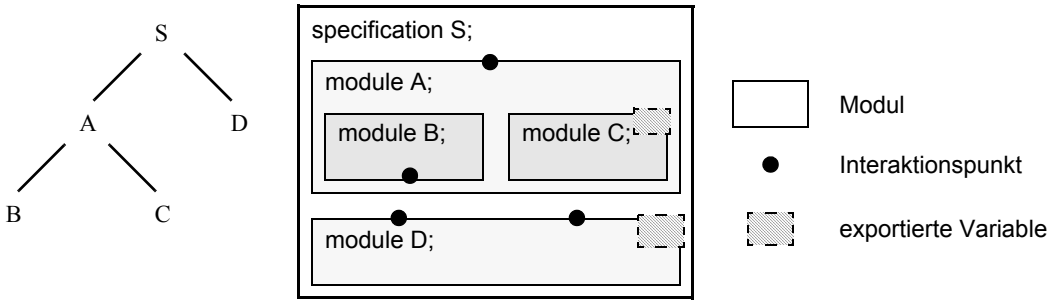


Abbildung 2-2: Statische Modulhierarchie zu Beispiel 2.1

Module haben eine wohldefinierte externe Schnittstelle (beschrieben durch den Modulheader) und ein privates Innenleben (beschrieben durch den Modulrumpf).[3] Die Kommunikation zwischen den Modulinstanzen basiert auf zwei orthogonalen Konzepten:

(i) asynchrone Nachrichtenübertragung und

(ii) gemeinsame Variablen.

2. Später werden in diesem Text – sofern sich die Bedeutung aus dem Kontext ergibt – *Modulinstanzen* auch einfach als *Module* bezeichnet. Dies entspricht u. a. dem Sprachgebrauch in [ISO97].

3. Eine Ausnahme bildet das Spezifikationsmodul, das keinen explizit definierten Modulheader besitzt; Estelle-Spezifikationen beschreiben geschlossene Systeme und haben somit keine äußere Schnittstelle.

Entsprechend besteht die äußere Schnittstelle eines Moduls aus einer Menge von (*externen*) *Interaktionspunkten* (i) und *exportierten Variablen* (ii, siehe auch Abb. 2-2). Die externen Interaktionspunkte dienen dabei als Schnittstelle für den asynchronen Austausch parametrisierter Nachrichten (*Interaktionen*), und die exportierten Variablen dienen als gemeinsame Variablen der Modulinstanz und ihrer Vatermodulinstanz.

Die dynamische Modulinstanzstruktur wird ergänzt durch eine dynamische Verbindungsstruktur, die den Zielpunkt der über einen Interaktionspunkt verschickten Nachrichten bestimmt. Zwischen zwei verbundenen Interaktionspunkten besteht ein bidirektionaler Kommunikationskanal. Nachrichten, die über einen verbundenen Interaktionspunkt geschickt werden, werden am anderen Endpunkt der Verbindung in einen FIFO-Puffer unbegrenzter Länge abgelegt, aus dem sie später (über den Empfängerinteraktionspunkt) in der Reihenfolge ihres Eintreffens entnommen werden können.

2.1.2 Zustand und Verhalten von Modulinstanzen

Jede Modulinstanz besitzt einen *lokalen Zustand*, der (unter anderem) den Kontrollzustand des Automaten und die Inhalte der lokalen Variablen umfasst. Das Verhalten der Modulinstanz wird durch eine Menge von *Transitionen* beschrieben. Jede Transition beschreibt eine Schaltbedingung und eine Schaltwirkung. Die *Schaltbedingung* legt fest, unter welchen (im Wesentlichen lokalen) Bedingungen die jeweilige Transition ausgeführt („*geschaltet*“, „*gefeuert*“) werden darf. Die *Schaltwirkung* beschreibt die (lokalen und globalen) Auswirkungen der Ausführung der Transition. Das Schalten einer Transition erfolgt dabei atomar.[4]

Syntaktisch wird eine Transition durch eine Menge von jeweils optionalen *Transitionsklauseln* und einen *Transitionsblock* repräsentiert. Transitionsklauseln beschreiben Schaltbedingungen und/oder Schaltwirkungen (siehe Tabelle 2.1). Der Transitionsblock besteht im Wesentlichen aus einem Pascal-ähnlichen Anweisungsblock, der beim Feuern der Transition ausgeführt wird.

Beispiel 2.2: Eine Transition mit mehreren Klauseln

```
TRANS
     FROM      unconnected
     TO        connected
     WHEN      IpToPartnerPM.ConReq{bConfirm: BOOLEAN}
          BEGIN
          nConnects := nConnects + 1;
          IF bConfirm THEN
               OUTPUT IpToPartnerPM.ConConf;
          END;
```

(Ende von Beispiel 2.2)

Die Auswahl der zu schaltenden Transitionen aus der Menge der schaltbereiten geschieht zum einen durch eine *modullokale Transitionsauswahl* und zum anderen durch eine *globale Auswahl* zwischen den Modulinstanzen. Die modullokale Transitionsauswahl bestimmt aus der Menge der *schaltbereiten*[5] Transitionen[6] eine Transition mit maximaler Priorität und bietet diese (nun-

4. Wir benutzen für das *Ausführen* einer *Transition* auch gleichberechtigt die in der deutschsprachigen Literatur verbreiteten Begriffe „*Feuern*“ („fire“) und „*Schalten*“. Analog kommen diese Begriffe auch für die *Modulinstanz* zur Anwendung, die die ausgeführte Transition enthält.

Tabelle 2.1: Schaltbedingungen und Schaltwirkungen von Transitionsklauseln

Klausel	Schaltbedingung	Schaltwirkung	Default
FROM	Kontrollzustand stimmt überein	—	von jedem Kontrollzustand
TO	—	neuer Kontrollzustand wird angenommen	Kontrollzustand bleibt unverändert
WHEN	Typ und Ziel des ersten Elements in der Warteschlange passen zum referenzierten Interaktionspunkt	Interaktion wird aus Warteschlange entfernt	spontane Transition
PROVIDED	der boolesche Ausdruck wird zu `true` ausgewertet	—	PROVIDED `true`
PRIORITY	Priorität der Transition	—	niedrigste Prioritätsklasse
DELAY	Transition ist bereit[a] seit angegebener Zeitspanne[b]	—	DELAY(0)
ANY	—[c]	—	—

a. d.h. die FROM- und WHEN-Bedingungen sind erfüllt
b. Zusätzlich gibt es eine zweite Zeitspanne mit *optionaler* Schaltbarkeit, die einen weiteren Indeterminismusfaktor bei der Auswahl bieten kann.
c. Die ANY-Klausel wird durch eine syntaktische Transformation expandiert.

mehr *schaltbare*[7]) zur Ausführung an.[8] Falls es mehrere Möglichkeiten gibt, erfolgt die Auswahl indeterministisch. Die globale Auswahl zwischen den angebotenen Transitionen der Modulinstanzen wird durch die Attributierung der Modulinstanzen gesteuert und erfolgt ebenfalls *teilweise indeterministisch*.

5. Eine Transition ist *(schalt-) bereit* („*enabled*“), wenn ihre WHEN-, PROVIDED- und FROM-Klauseln erfüllt sind (siehe Abschnitt 9.6.2 von [ISO97]).

6. Eine mögliche DELAY-Klausel muss ebenfalls erfüllt sein, indem die Transition eine bestimmte Zeit lang ununterbrochen schaltbereit war.

7. Eine Transition ist *schaltbar* („*fireable*“), wenn sie (1.) bereit ist, (2.) eine mögliche DELAY-Klausel erfüllt ist (also die Bereitschaft ausreichend lange unterbrechungsfrei bestanden hat) und sie (3.) unter den übrigen Transitionen ihrer Modulinstanz mit diesen Eigenschaften maximale Priorität hat (siehe Abschnitt 9.6.2 von [ISO97]).

8. Genauer muss zum Übergang von der *Schaltbereitschaft* zur *Schaltbarkeit* auch eine mögliche DELAY-Klausel erfüllt sein (siehe Abschnitt 3.4).

Aufgrund der Möglichkeit zu indeterministischem Verhalten wird die Semantik einer Estelle-Spezifikation durch eine *„next-state"-Relation*[9] beschrieben, die alle zulässigen Zustandsübergänge beschreibt. Daraus lässt sich (ausgehend vom initialen Zustand des Systems) die Menge der erlaubten Ereignis- und Zustandsfolgen ableiten. Das semantische Modell basiert dabei auf einem Server-Modell (siehe auch Abschnitt 4.2.1.1).

Die Semantik einer (geschlossenen) Spezifikation *S* wird *formal* durch eine Menge von „*Berechnungen*" („*computations*") dargestellt. Berechnungen sind Sequenzen $<sit_0, sit_1,....>$ so genannter „*globaler Situationen*" („*global situations"*), wobei sit_0 „*initial*" ist und für alle $j>0$, sit_j eine mögliche „*nächste globale Situation*" („*next global situation*") von sit_{j-1} bezüglich der „*next-state*" Relation ist. Im Wesentlichen beinhaltet eine globale Situation $sit = (gid_{SP}; A_1,...,A_n)$ die lokalen Zustände aller Modulinstanzen (gid_{SP}: Zustand der Eingangswarteschlangen, Werte lokaler Variablen, Modulinstanzstruktur, Verbindungsstruktur) und die Mengen $A_1,...,A_n$ der jeweils in den Subsystemen $S_1,...,S_n$ zum Feuern ausgewählten Transitionen. Somit definiert eine Estelle-Spezifikation ein *Transitionssystem* (S, δ), wobei *S* und δ sich auf die Menge globaler Situationen bzw. die next-state Relation beziehen. Nebenläufigkeit wird dabei durch Interleaving modelliert.

2.1.3 Anbindung an eine Umgebung

Standard-Estelle-Spezifikationen[10] beschreiben (topologisch) geschlossene Systeme, d.h. Systeme ohne Schnittstelle zu ihrer Umgebung. Ohne die Anwendung geeigneter Erweiterungen (wie „*Open-Estelle*", siehe Kapitel 7) ist die Ankopplung von in Estelle spezifizierten Systemen an reale Umgebungen (z. B. an einen IP-Service unter UNIX) nur durch pragmatische Mittel auf Implementierungsebene möglich [GoRoTh96]. Die in Estelle dazu bereits vorgesehenen Mittel sind primitive Funktionen bzw. Prozeduren. Diese erlauben den Zugriff auf Dienste, die nicht Bestandteil der Estelle-Spezifikation sind, machen jedoch die Existenz einer formalen Semantik der Spezifikation vom Vorhandensein einer geeigneten formalen Beschreibung dieser Dienste abhängig.

2.1.4 Estelle-Codegeneratoren

Estelle eignet sich durch seine operationale Verhaltensbeschreibung und seine einfache Typstruktur besonders für eine automatische Implementierung. Im Gegensatz zu anderen FDTs, die eigenschaftsorientierte Verhaltensbeschreibungen oder abstrakte, rekursive und axiomatische Typdefinitionen benutzen, kann Estelle relativ direkt implementiert werden: Estelle-Typen sind im Wesentlichen Pascal-Typen und haben somit eine feste Struktur. Sie können direkt in eine Implementierungssprache wie C abgebildet werden. Gleiches gilt für die Verhaltensbeschreibung von Estelle-Spezifikationen, die im Wesentlichen durch Transitionen mit sequentiellen, imperativen, Pascal-ähnlichen Anweisungsblöcken gegeben werden. Auch sie können direkt nach C abgebildet werden.

---

9. Ein derart beschriebenes System ist genau dann indeterministisch, wenn es einen erreichbaren Systemzustand gibt, für den die next-state-Relation *nicht rechtseindeutig* ist.

10. Im Folgenden wird der in [ISO97] beschriebene Sprachumfang zur Abgrenzung gegenüber den eingeführten Erweiterungen auch als „*Standard-Estelle*" bezeichnet.

Für Estelle wurden früher bereits mehrere Compiler mit unterschiedlichen Zielsetzungen entwickelt. Allen gemeinsam ist, dass sie aus einer Estelle-Spezifikation Pascal-, C- oder C++-Quellcode generieren, der dann zu einem ablauffähigen Programm übersetzt werden kann. Im Folgenden werden die gebräuchlichsten Compiler kurz vorgestellt:

- Der **NIST-Estelle-Compiler** war einer der ersten Estelle-Compiler und wurde vom amerikanischen *National Institute for Standards and Technologies* (NIST) entwickelt [SiSt90]. Er ist nicht mehr verfügbar und wurde von PET/DINGO (s. u.) abgelöst.
- Der **UHH-Estelle-Compiler** wurde an der Universität Hamburg entwickelt [KrGo93]. Er basiert auf dem oben erwähnten NIST-Compiler und erzeugt wie dieser C-Code. Der UHH-Compiler erlaubt die Erzeugung einer auf mehreren Rechnerknoten verteilt ausführbaren Implementierung einer Spezifikation. Einer seiner Schwachpunkte ist die fehlende Unterstützung von exportierten Variablen.
- Das Codegenerierungswerkzeug **PET/DINGO** besteht aus dem Frontend PET (*Portable Estelle Translator*) und dem Backend DINGO (*Distributed Implementation Generator*). Er wurde vom NIST [SiSt93] als Nachfolger des o. g. NIST-Estelle-Compilers entwickelt und unterstützt ebenfalls die Erzeugung einer auf mehreren Rechnerknoten verteilt ausführbaren Implementierung einer Spezifikation. Da PET/DINGO objektorientiert (in C++) implementiert wurde und auch der generierte Code auf einer C++-Klassenbibliothek aufsetzt, eignet sich PET/DINGO gut für die Einbettung von Werkzeugen zum Performance-Monitoring. Wir werden in Abschnitt 3.6.3 das Werkzeug PATO vorstellen, das zu diesem Zweck speziell für PET/DINGO entwickelt wurde. Außerdem nutzt der in Abschnitt 3 vorgestellte Estelle-Compiler XEC das Compiler-Frontend PET (siehe Abschnitt 3.1.1).
- Bei **EC** (*Estelle Compiler*) handelt es sich im Gegensatz zu den übrigen Werkzeugen um einen kommerziellen Estelle-Compiler, der von Bull S.A. entwickelt [RiCl89] und später vom französischen Institut National des Télécommunications (INT) übernommen wurde [Bud92]. EC ist Teil des Estelle-Entwicklungssystems **EDT** (*Estelle Development Toolset*), das unter anderem auch einen Estelle-Debugger (EDB) enthält. EC wurde mit dem Ziel entwickelt, eine optimierte Implementierung einer Estelle-Spezifikation zu erzeugen. Da es sich jedoch um ein kommerzielles Produkt handelt, sind die Quellen des Compilers und der Laufzeitbibliothek nicht zugänglich, was die Möglichkeiten zur detaillierten Leistungsanalyse stark einschränkt.

Wir werden in Kapitel 3 mit dem „*eXperimental Estelle Compiler*“ (**XEC**) einen weiteren Estelle-Codegenerator präsentieren, den wir als Grundlage für die in dieser Arbeit vorgestellten Implementierungs- und Optimierungsexperimente eigens neu entwickelt haben.

2.2. Effiziente Implementierung

Als Grundlage für die weiteren Untersuchungen identifizieren wir nun verschiedene *Effizienzbegriffe* hinsichtlich einer (manuell oder automatisch erstellten) Implementierung von Kommunikationssystemen und formulieren sie als mögliche *Optimierungsziele* (Abschnitt 2.2.1).

Aufbauend darauf führen wir in Abschnitt 2.2.2 die in dieser Arbeit grundsätzlich verfolgten Optimierungsansätze ein, wobei die automatische Implementierung von Estelle mit Hilfe von XEC im Vordergrund steht.

2.2.1 Optimierungsziele

Als *Optimierungsziele* bei der (automatischen oder auch manuellen) Implementierung von Kommunikationsprotokollen kommen zunächst ganz unterschiedliche Aspekte in Frage:

(i) geringe (Brutto-) Ausführungsdauer einer (primitiven oder komplexen) Operation

(ii) effiziente Ausführung des Kontrollflusses einer Protokolloperation

(iii) hoher Datendurchsatz bzw. effiziente Datenübertragung

(iv) geringe CPU-Belegung

(v) geringer (Haupt-) Speicherplatzbedarf

(vi) geringe „Turn-Around"-Dauer im Entwicklungsprozess

Die *Brutto-Ausführungsdauer einer Protokolloperation (i)* beinhaltet die gesamte (Real-) Zeit zur Durchführung einer Operation, z. B. eines Verbindungsaufbaus oder der Übertragung einer vorgegebenen Menge von Datenpaketen über ein Kommunikationssystem. Sie lässt sich aufteilen in den Zeitaufwand zur Durchführung des *Kontrollflusses* der Protokolloperation und in den möglicherweise enthaltenen Zeitaufwand zur Übertragung von *Nutzdaten*. Beide Aspekte können durch Variation der übertragenen Datenmengen einen unterschiedlichen Anteil an der *Brutto-Ausführungsdauer* der Protokolloperation haben und werden nach Möglichkeit getrennt betrachtet.

Zur Trennung beider Aspekte konzentrieren wir uns in Kapitel 4 zunächst auf Punkt *(ii)*, die *effiziente Ausführung des Kontrollflusses* einer Menge formal spezifizierter Protokolloperationen auf einer realen Maschine, wobei wir die dabei übertragenen Datenmengen *gering* halten, um den mit ihrer Handhabung verbundenen Zeitaufwand zu minimieren. Dabei gehen wir im Allgemeinen von einer sequentiellen Ausführung der Protokollimplementierungen aus und stellen Fragen zur effizienten Parallelisierbarkeit der Implementierungen zurück.[11]

Die *Optimierung des Datendurchsatzes* und eine *effiziente Datenübertragung (iii)* wird Gegenstand von Kapitel 5 sein. Dort werden wir – in Umkehrung des oben beschriebenen Vorgehens – durch den Einsatz von großen zu übertragenden Datenmengen den Anteil der zur Datenübertragung notwendigen Zeit in den Vordergrund stellen.

---

11. Allgemein basiert die Fragestellung der effizienten *parallelen* Implementierbarkeit von Protokollspezifikationen natürlich auch auf der Effizienz der zu Grunde liegenden Basisoperationen (siehe auch [FiHo94], [LaFi95], [Got+96], [KrGo93]) und ist daher eine auf den Zielsetzungen dieser Arbeit aufbauende Fragestellung (siehe auch Abschnitt 4.5).

Während die Punkte *(i)* bis *(iii)* auf eine Maximierung der *Gesamtleistung* des Kommunikationssystems abzielen, wird im praktischen Einsatz einer Protokollimplementierung oft auch die *sparsame Verwendung von Ressourcen* gefordert. Hier stehen die CPU- und die Hauptspeicherbelegung im Vordergrund.

Gerade die *CPU-Belegung*[12] *(iv)* ist bei Multitasking-Systemen, welche neben einer einzelnen Protokollimplementierung noch andere Aufgaben erfüllen, ein wichtiger Faktor. Hier gilt es, in den Zeiträumen, in denen auf externe Ereignisse (z. B. den Empfang eines Paketes) oder Timeouts (z. B. zum wiederholten Senden eines verlorengegangenen Paketes) gewartet wird, die CPU nicht unnötig zu belegen, gleichzeitig aber im Sinne von Punkt *(i)* schnellstmöglich auf protokollbezogene Ereignisse reagieren zu können.[13] Wir werden diesen Punkt im Zusammenhang mit der Plattformanbindung der Implementierung in Abschnitt 3.5.2 untersuchen.

Der *Speicherplatzbedarf (v)* spielt dagegen bei modernen Hardware-Plattformen meist nur eine untergeordnete Rolle.[14] Lediglich mögliche negative Auswirkungen auf die Ausführungsdauer z. B. durch Cache-Verdrängung oder gar Paging bzw. Swapping auf einen Sekundärspeicher auf Grund eines ungewöhnlich großen „Working-Sets“ einer Implementierung gilt es zu vermeiden. Im Bereich *eingebetteter Systeme* mit ihren weitaus restriktiveren Ressourcenbeschränkungen ergeben sich hier möglicherweise jedoch andere Anforderungen.

Im Gegensatz zu den bisher diskutierten Optimierungszielen, die sich ausschließlich auf *zur Laufzeit* relevante Eigenschaften beziehen, ist die Forderung nach einer *geringen Turn-Around-Dauer*[15] *(vi)* nur während der Entwicklungs- und Testphase relevant. Wir werden auf diesen Punkt in Kapitel 7 im Zusammenhang mit der Erweiterung *Open-Estelle* nochmals zurückkommen (siehe Abschnitt 7.5.3).

2.2.2 Ansatzpunkte zur Effizienzsteigerung

Bei der effizienten (automatischen oder manuellen) Implementierung eines formal spezifizierten Kommunikationsprotokolls sind verschiedene Ansätze zur Effizienzsteigerung denkbar.

Man kann diese in vier Ebenen unterteilen:

(a) Optimierung der *Basisoperationen* wie z. B. der Übertragung einer Nachricht oder der Prüfung einer Schaltbedingung,

(b) Optimierung auf *Managementebene*, z. B. wie kann die nächste schaltbare Transition bzw. ausführbare Protokolloperation möglichst effizient ermittelt werden,

(c) Anwendung eines implementierungsorientierten *Spezifikationsstils*,

12. Die Ausführungsdauer einer Operation ergibt sich (auf Single-CPU-Systemen) aus der Summe der CPU-Belegungszeiten und den Zeiten, in denen der Prozess (z. B. aufgrund einer freiwilligen Blockade) nicht gerechnet wird und damit die CPU *anderen Prozessen* zur Verfügung steht.

13. In eingebetteten Systemen kann zudem unabhängig von konkurrierenden rechenwilligen Prozessen eine geringe CPU-Belastung zur Minimierung des *Energieverbrauchs* erwünscht sein.

14. Systeme, die z. B. aufgrund eines extrem hohen Bandbreiten-Verzögerungsprodukts in Basisdiensten (z. B. Höchstgeschwindigkeitsnetze) große Datenmengen verarbeiten und zwischenspeichern müssen, erfordern u. U. auch nach heutigen Verhältnissen große Mengen an Speicherplatz. Sie sind jedoch nur bedingt eine Domäne automatisch generierter Protokollimplementierungen.

15. also der Zeitspanne von der Änderung einer Spezifikation über die Codegenerierung und Compilation zu einer ausführbaren Implementierung

(d) Steigerung der *Ausdrucksfähigkeit* der eingesetzten FDT hinsichtlich einer effizienten Implementierbarkeit.

Offensichtlich zielen die genannten Punkte auf ganz unterschiedliche Abstraktionsniveaus bei der Implementierung von formal beschriebenen Protokollen.

Wir gehen im Folgenden von einer *semantikkonformen Implementierung* des formal spezifizierten Protokolls aus. Dies bedeutet zunächst lediglich, dass die Strukturen der Implementierung, ihr Zustandsraum und die von ihr ausgeführten Operationen in einem *nachvollziehbaren Zusammenhang*[16] stehen. Wir werden diese Annahme im Folgenden anhand der einzelnen Punkte weiter verfeinern.

Die Optimierung der Basisoperationen *(a)* zielt auf eine möglichst effiziente Implementierung von Verarbeitungsprimitiven der spezifizierten Protokolloperationen ab. Diese beinhalten sowohl *Datenverarbeitungsoperationen* (z. B. das Kopieren eines Nutzdaten-Puffers oder die Übertragung eines Interaktionsparameters), als auch *kontrollflussorientierte Operationen* (z. B. der Test einer Transition auf Ausführbarkeit, der Wechsel eines Kontrollzustands).[17] Die Ausführung der Basisoperationen ist das *eigentliche Ziel der Protokollimplementierung* und sie ist damit zunächst unumgänglich.[18] Die höheren Managementebenen dienen lediglich der Koordination dieser Operationen.

Bei der Optimierung auf Managementebene *(b)* sollen innerhalb der durch das semantische Modell der Spezifikation vorgegebenen Sequenzen von Basisoperationen möglichst effizient *zulässige Folgen* selektiert und koordiniert werden. Erlaubt das semantische Modell Indeterminismus, so kann als weiteres Optimierungsziel zudem die Auswahl einer Sequenz angestrebt werden, die innerhalb des semantischen Modells eine Aufgabe mit möglichst effizienten[19] Basisoperationen erreicht. Man beachte: Während die erste Zielsetzung die *Effizienz des Managementprozesses* selbst betrifft, zielt die zweite auf die Auswahl einer *vorteilhaften Sequenz von Basisoperationen* aus den vom semantischen Modell erlaubten ab.

Diese Aufgabe kann auf unterschiedliche Arten erfüllt werden. Besonders nahe liegend ist dabei natürlich die enge Anlehnung an das semantische Modell der Spezifikationssprache. Dies ermöglicht die Gewinnung von Implementierungen, deren *Korrektheit* sich „*per Konstruktion*“ ergibt. Alternative Ansätze können sich auch weit vom semantischen Modell entfernen, wodurch jedoch die Korrektheit jeweils zu beweisen bleibt.

Die Anwendung eines implementierungsorientierten Spezifikationsstils *(c)* zielt darauf ab, bereits bei der Erstellung einer Spezifikation die Möglichkeit einer effizienten Implementierung zu berücksichtigen. Einige Ansätze hierzu sind durchaus unabhängig von einer konkreten Implementierungstechnik realisierbar, indem sie insbesondere die *Ausführungs-Komplexität* der Spezifikation selbst reduzieren. Typische Beispiele sind die Minimierung der Synchronisation von nebenläufigen Teilsystemen (in Estelle z. B. über die Modulattributierung) oder die Vermeidung unnötiger Kopieroperationen. Andere Optimierungsmaßnahmen erfordern teilweise

16. Die semantische Konformität einer Implementierung erfordert genau genommen die Spezialisierung der in der Semantik vorgegebenen Zustands- und Berechnungsfolgen im Sinne einer „implements“-Relation.

17. teilweise überlappen sich auch beide Aspekte

18. Insbesondere manuelle Implementierungen, die von der Ausführung dieser Basisoperationen abweichen, sind per se semantisch nicht konform (s. u.).

19. also von möglichst *wenigen* und (auf Implementierungsebene) *einfachen* Basisoperationen

jedoch intime Kenntnisse des Kodierungsverfahrens und machen somit eine Maßschneiderung der Spezifikation auf dieses Kodierungsverfahren (bzw. den Generator bei automatisch erzeugten Implementierungen) erforderlich.

Leider sind *problemorientierte* Spezifikationsstile häufig gerade nur wenig implementierungsorientiert, so dass hier oft entweder ein (zum Teil in die eine oder andere Richtung extremer) Kompromiss gesucht wird oder aufbauend auf eine *problemorientierte Spezifikation* in einem Zwischenschritt zunächst eine *implementierungsorientierte Spezifikation* erstellt wird. Die Nachteile beider Vorgehensweisen sind offensichtlich, zumal wenn bei stark zustands- und strukturorientierten Semantiken wie im Falle von Estelle ein semantikkonformer[20] Übergang von einer problemorientierten Spezifikation zu einer implementierungsorientierten Spezifikation kaum möglich erscheint (siehe auch Abschnitt 7.3.3).

Die Problematik der Semantikerhaltung bei solchen Transformationen wird noch kritischer, wenn bei der Implementierung weitgehend oder völlig von der Semantik der Spezifikation abgewichen wird, wie es in der Praxis häufig bei manuell erstellten Implementierungen geschieht. Die Ursache für dieses Abweichen sind meist *Über- oder Fehlspezifikationen*, die sich teilweise nicht aus dem Problem (also dem intendierten Protokollablauf) ergeben, sondern aus den Beschränkungen der eingesetzten FDT.

Typische Beispiele in Estelle sind:

- das Bedürfnis nach gemeinsamen Zustandsraumkomponenten zwischen Geschwister-Modulen, welches zu aufwändigen (expliziten und nachrichtenbasierten) Synchronisationen zwischen den Modulen führt (siehe Abschnitt 6.4.3),
- die Überspezifikation der Modulsynchronisation aufgrund der relativ strikten (impliziten) Synchronisation zwischen Modulen eines Subsystems (siehe Abschnitt 4.2.2.3),
- der Transport von Massendaten über einen Protokollstack durch mehrfache (semantische) Kopieroperationen, obwohl die Daten per Referenz „durchgereicht“ werden sollen (siehe Abschnitt 6.4).

Diese semantischen Brüche werden in manuell erzeugten Implementierungen auf Basis von Kontextwissen des Implementierers meist umgangen, indem nicht die Spezifikation, sondern *das eigentlich intendierte Protokoll* implementiert wird. Diese Vorgehensweise ist natürlich nicht wünschenswert und steht automatischen Implementierungsgeneratoren nur offen, wenn diese (meist entsprechend gesteuert durch den Spezifizierer) nicht semantikkonforme Implementierungen erzeugen.[21]

Ein radikaler Ansatz zur Lösung dieses Problems besteht in der *Erweiterung der FDT selbst* zum Zwecke einer Steigerung der Ausdrucksfähigkeit bzw. effizienten Implementierbarkeit der eingesetzten FDT (d), und in der Tat werden wir an einigen Stellen Estelle-Erweiterungen vorstellen, die bei einem oder mehreren der in Abschnitt 2.2.1 identifizierten Ziele wirksame und semantikkonforme Optimierungen erst ermöglichen.

---

20. im Sinne einer „*implementiert*“-Relation, (d.h. die zweite Spezifikation ist semantisch eine *Spezialisierung* der ersten)

21. Ein typisches Beispiel ist die (explizite) Übertragung von Daten zwischen Modulen „per Referenz“, was zunächst nur durch Manipulation des Codegenerators oder Umgehen der entsprechenden Prüfungen (z. B. durch primitive Funktionen, siehe [Cat98]) automatisch implementiert werden kann. Die semantischen Implikationen dieser Vorgehensweise müssen vom Spezifizierer überwacht werden, um möglicherweise resultierende Störungen der Semantik (z. B. der Copy-Semantik des Datentransportes) zu vermeiden (siehe auch Abschnitt 6.4).

In Kapitel 4 entwickeln und evaluieren wir verschiedene Optimierungstechniken für die automatische Implementierung formal spezifizierter Protokolle auf allen o. g. Ebenen, wobei jedoch der Aspekt der Optimierung auf Managementebene *(b)* im Vordergrund steht.

Die eingesetzten Maßnahmen zur Optimierung der Basisoperationen *(a)* werden dagegen als primäre Eigenschaft des Implementierungsgenerators bereits in Kapitel 3 weitgehend abgehandelt und haben nur geringen Einfluss auf unsere weiteren Optimierungsuntersuchungen.

Eine Ausnahme bildet hier lediglich die effiziente Datenübertragung (Kapitel 6), die neben dem Spezifikationsstil und möglichen Erweiterungen der Beschreibungstechnik auch von den zum Datenmanagement eingesetzten Basisoperationen *(a)* abhängt.

2.3. Performance-Evaluierung

Als Grundlage für die Bewertung der Leistung bzw. von leistungssteigernden Maßnahmen einer automatischen Implementierung von formalen Protokollspezifikationen benötigt man einen *Bewertungsmaßstab für die Effizienz* einer solchen Implementierung.

Wir haben bereits im letzten Abschnitt verschiedene Optimierungsziele wie (Brutto-) Ausführungsdauer, Durchsatz oder geringe CPU-Belegung identifiziert, und solche *ausführungszeitbezogenen* Größen bilden letztlich in Hinsicht auf den praktischen Einsatz automatisch generierter Implementierungen den nahe liegendsten Bewertungsmaßstab.

Abseits solcher ausführungszeitbezogener Größen kann man aber sowohl *analytisch* wie auch *experimentell* die Effizienz von Implementierungen anhand *zählbarer* Größen wie „*Anzahl zu testender Transitionen pro gefeuerter Transition*“ oder „*Anzahl der Nutzdatenkopieroperationen bei einem Ende-zu-Ende-Transport*“ abschätzen. Der Vorteil dieser zählbaren Größen ist dabei neben der *analytischen Zugänglichkeit* besonders die Unabhängigkeit von Messfehlern und die gute Reproduzierbarkeit im experimentellen Bereich. Entsprechend werden wir in Kapitel 4 (bzgl. der Ausführung des Kontrollflusses) und Kapitel 6 (bzgl. der Übertragung von Nutzdaten) regen Gebrauch davon machen.

Finaler Bewertungsmaßstab bleiben jedoch nicht zuletzt zur Berücksichtigung des mit vielen Optimierungsmethoden einhergehenden organisatorischen Overheads die ausführungszeitbezogenen Auswertungen. Entsprechend münden unsere Untersuchungen in Kapitel 4 und Kapitel 6 jeweils in entsprechenden Messungen. Hier sind hohe Genauigkeit und geringer messbedingter Overhead gefragt. Weiter sind zur Kompensation von statistischen Fehlern[22] Versuchswiederholungen und statistische Auswertungen erforderlich.

Wir werden später in Abschnitt 3.6 sehen, dass XEC sowohl für die Messung zählbarer Größen, als auch für Echtzeitmessungen sehr gute technische Voraussetzungen bietet. Als Beispiel seien hier integrierte Ereigniszähler und Ausführungszeitmessungen, der Monitoringsupport und nicht zuletzt die deterministische Zeitsimulation (siehe Abschnitt 3.5.2.2) genannt.

Eine wichtige Grundlage für eine realistische Leistungsbewertung ist die Auswahl geeigneter *Benchmarkspezifikationen*,[23] anhand derer die Leistungsmessungen erfolgen. Diese sollten bzgl. der auftretenden Spezifikationsszenarien und ihre Komplexität eine hohe *Praxisrelevanz* besitzen, um die *Übertragbarkeit* der gewonnenen Ergebnisse auf andere Spezifikationen zu rechtfertigen. Andererseits sollten sie einfach genug gestaltet sein, um einer *Analyse* zugänglich zu bleiben bzw. um Variationen des *Spezifikationsstils* oder gar *Erweiterungen der Spezifikationssprache* effizient anwenden zu können.

Um diesen Anforderungen gerecht zu werden, nutzen wir verschiedene Spezifikationen als Benchmarks, die teilweise *synthetisch* sind und nur dazu dienen, bestimmte Effekte isoliert zu betrachten, teilweise aber auch *komplexe reale Protokolle* (z. B. XTP 4.0) realisieren und so die damit verbundene Komplexität und Abdeckung von praxisrelevanten Protokollmechanismen bieten.

---

22. Die statistischen Schwankungen sind zu erheblichen Teilen durch die mangelnde Feinkontrolle über die Vergabe der Ressourcen (CPU, Hauptspeicherzugriff) in den betrachteten Systemplattformen (Sun-Sparc, Linux-IA32) begründet.

23. „*Benchmarks sind standardisierte Testverfahren, die es ermöglichen, bestimmte zeit- und performanzbezogene Eigenschaften (von Instanzen) von Systemen oder Systemklassen zu ermitteln, um diese untereinander in Bezug auf diese Eigenschaften hin zu vergleichen.*“ [Wik04]

2.3.1 Synthetische Benchmarks

Aufbauend auf der Untersuchung verschiedener Spezifikationen praxisrelevanter Protokolle wurde ein rein synthetischer Estelle-Benchmark [Dah96] entwickelt, der es gestattet, die automatische Implementierung derjenigen Estelle-Mechanismen isoliert und quantitativ zu bewerten, die sich als besonders laufzeitintensiv erwiesen haben. So können z. B. Modulinstanziierungen, Nachrichtenübertragungen mit unterschiedlich großen Argumenttypen, verschiedenen Auswahlsituationen usw. teilweise auch ohne Quellen für Compiler oder Laufzeitbibliothek quantitativ bewertet werden.

Diese Benchmarks wurden unter anderem bei der Entwicklung und Erprobung verschiedener Implementierungsvarianten von Detaillösungen der XEC-Laufzeitbibliothek eingesetzt. Aufgrund der inhärent geringen Praxisrelevanz als Gesamtbenchmark (speziell bzgl. der globalen Auswahl) verzichten wir hier jedoch zugunsten der anderen Benchmarkspezifikationen auf eine quantitative Auswertung.

2.3.2 Der Datenübertragungs- bzw. Ping-Pong-Benchmark

Diese synthetische Benchmarkspezifikation bildet ein Weitverkehrsszenario nach, bei dem zwei Hosts mit jeweils einem lokalen Kommunikationsstack über ein vermittelndes Netzwerk mit einer parametrierbaren Anzahl von „Hops“ kommunizieren (siehe Abb. 2-3).

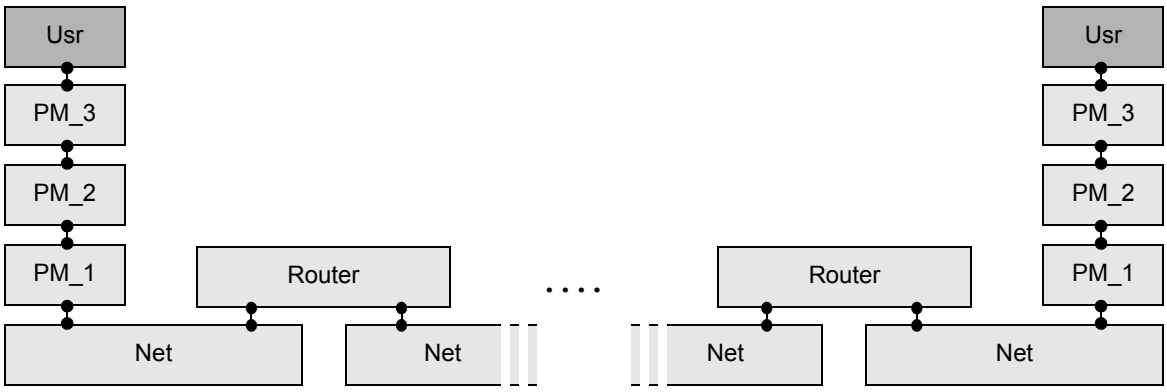


Abbildung 2-3: Modul- und Verbindungsstruktur des Datenübertragungsbenchmarks

Dieser Benchmark ist in vielen Aspekten parametrierbar und wurde ursprünglich zur Auswertung verschiedener Datenhandhabungs- und -Transportmechanismen unter Variation der transportierten Datenmengen entworfen und zeichnet sich dadurch aus, dass er ein sehr einfaches Kommunikationsverhalten für die einzelnen Knoten hat und somit den Aufwand zur Kontrollflussteuerung innerhalb der Module minimiert. In dieser Form wird er auch in Abschnitt 6.4 eingesetzt. In Anhang B.1 bis B.3 sind mehrere Versionen des Spezifikationstextes unter Anwendung verschiedener Datentransportmechanismen wiedergegeben.

Reduziert man die übertragenen Datenmengen der Ausgangsspezifikation (Anhang B.1) auf ein Minimum, so erhält man einen Benchmark, der aufgrund seiner konzeptionellen Einfachheit sehr gut zur Motivation und Erprobung globaler Auswahlmethoden geeignet ist. Entsprechend wird er insbesondere in Abschnitt 4.2 zur Bewertung solcher Methoden eingesetzt, zumal er im

Gegensatz zu den übrigen Spezifikationen die Anforderungen zur Anwendung eines Activity-Thread-Modells erfüllt (siehe Abschnitt 4.2.1.2). Die dort eingesetzte Version wird aufgrund ihres Kommunikationsmusters als „*Ping-Pong*“-Benchmark bezeichnet.

2.3.3 Der Sliding-Window-Applikationsbenchmark

Dieser Benchmark basiert auf dem in [Turn93] vorgestellten Spezifikationsfragment eines Sliding-Window-Protokolls. Ergänzt zu einer vollständigen Protokollmaschine und integriert in ein Ende-zu-Ende-Kommunikationsszenario kann man anhand dieser Spezifikation verschiedene Effekte in Transportschichtprotokollen nachbilden (Abschnitt 4.2.2.5, Anhang B.4). Insbesondere werden hier in erheblichem Umfang `DELAY`-Transitionen[24] eingesetzt (siehe auch Abschnitt 4.3.3). Wir nutzen diese Spezifikation entsprechend zur Evaluierung globaler (siehe Abschnitt 4.2) und modullokaler (siehe Abschnitt 4.3) Auswahlstrategien.

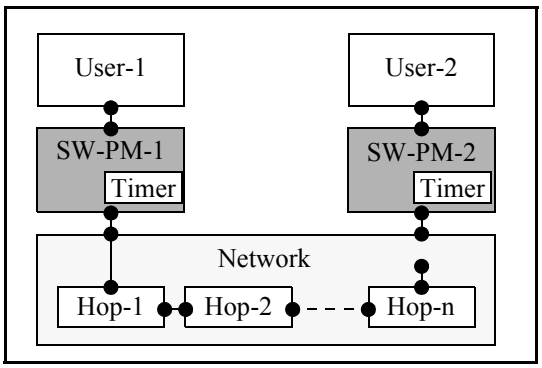


Abbildung 2-4: Modulinstanz- und Verbindungsstruktur des Sliding-Window-Benchmarks

2.3.4 Der XTP-Applikationsbenchmark

Die Estelle-Spezifikation des XTP-Protokolls (Abschnitt 2.3.4.2) stellt eine der größten bekannten Estelle-Spezifikationen dar und bildet aufgrund ihrer enormen Komplexität eine große Herausforderung an die effiziente Implementierung. Sie ist daher besonders als Benchmarkspezifikation zur Repräsentation komplexer Protokollmaschinen geeignet.

Wir geben zunächst einen kurzen Überblick über das implementierte Protokoll XTP 4.0, bevor wir dann in Abschnitt 2.3.4.2 näher auf die Estelle-Spezifikation eingehen.

24. insbesondere zur Modellierung einer selektiven Paketwiederholung

2.3.4.1 Das ‘Xpress Transport Protocol’

XTP 4.0 [XTP95] ist — im Gegensatz zu früheren XTP-Versionen — ein reines Transportschichtprotokoll.[25] Die Zielsetzung bei der Entwicklung war es, durch eine vollständige Neuentwicklung die Schwächen der etablierten Transportschichtprotokolle (wie TCP und TP4) zu überwinden und eine bessere Anpassung an die Eigenschaften moderner Kommunikationsmedien mit ihren hohen Bandbreiten und geringen Fehlerraten zu erreichen.

XTP ist ein sehr flexibles Protokoll, denn es erlaubt den jeweiligen Anwendungen eine umfassende Kontrolle über die genauen Eigenschaften der Datenübertragung. So können verschiedene Kommunikationsparadigmen (Datagramm, Transaktion, Verbindung) völlig unabhängig von der Fehlerkontrolle (völlig zuverlässig bis unkorrigiert) gewählt und sogar bei einer schon bestehenden Verbindung noch geändert werden. Auch die Unterstützung zuverlässiger Multicasts, die ein integraler Bestandteil von XTP ist, kann zusammen mit (und unabhängig von) den übrigen Mechanismen eingesetzt werden. Ebenso erlaubt XTP eine getrennte Raten- und Flusskontrolle und macht so ineffiziente Heuristiken, wie etwa den „slow-start“ von TCP, überflüssig. XTP 4.0 ist dabei nicht an einen speziellen Basisdienst gebunden, sondern kann nicht nur auf typischen Vermittlungsschicht-Diensten (z. B. IP), sondern auch direkt auf einfache LAN-Dienste (z. B. Ethernet IEEE 802.3) aufgesetzt werden.

Im Sinne eines Hochleistungsprotokolls war neben großer Flexibilität auch eine hohe Performance unter allen Betriebsbedingungen ein Ziel beim Design von XTP. Dazu wurden zum einen Protokollmechanismen eingesetzt, die es erlauben, die Anzahl der Synchronisationen zwischen Partnerinstanzen zu minimieren, um dadurch unnötige Wartezeiten zu vermeiden. So erlaubt z. B. der Einsatz eines impliziten Verbindungsaufbaus die Einsparung einer kompletten „round trip delay time“. Dies führt speziell im Transaktionsbetrieb zu enormen Leistungssteigerungen gegenüber einem regulären Verbindungsaufbau.[26]

Zum anderen wurde eine Leistungssteigerung bei der Verarbeitung von Paketen in der Protokollmaschine durch verschiedene Maßnahmen ermöglicht. Z. B. haben die Paketheader ein festes Format und erlauben daher eine schnelle Dekodierung. Die einzelnen Felder im Header liegen auf optimal ausgerichteten Offsets (8-Byte-Alignment für 8-Byte Strukturen usw.) und die einzelnen Komponenten eines Paketes (Segmente) haben eine auf 8-Byte-Vielfache aufgerundete Größe, so dass die in modernen Hardwarearchitekturen vorhandenen Optimierungen greifen können. Weiterhin erlaubt das Protokoll den Austausch von sogenannten „return keys“, durch die der Empfänger eines Paketes dieses ohne Suchoperationen einer Verbindung zuordnen kann. Diese und viele weitere Maßnahmen erlauben eine sehr effiziente Implementierung von XTP.

2.3.4.2 Estelle-Spezifikation von XTP

Die der Untersuchung zu Grunde liegende Estelle-Spezifikation wurde ursprünglich am Institut National des Télécommunications für XTP 3.6 erstellt [Bud93] und später an XTP 4.0 angepasst und speziell im Bereich der Multicastunterstützung erheblich erweitert [CaBo96]. Da Estelle nur geschlossene Systeme beschreiben kann, wurde der XTP Protokollautomat in ein

25. Die früheren XTP Versionen umfassten auch Funktionalitäten der Vermittlungsschicht; entsprechend bedeutete XTP damals ‘Xpress Transfer Protocol’.

26. Es gibt analoge Ansätze zur Erweiterung von TCP hin zu einer transaktionsorientierten Form T/TCP (siehe [RFC1379] und [RFC1644]).

vollständiges Umgebungsszenario eingebettet. Dieses enthält zusätzlich ein Netzwerkmodul und zwei Benutzermodule. Die Grobstruktur dieses Kommunikationsszenarios ist in Abb. 2-5 dargestellt.

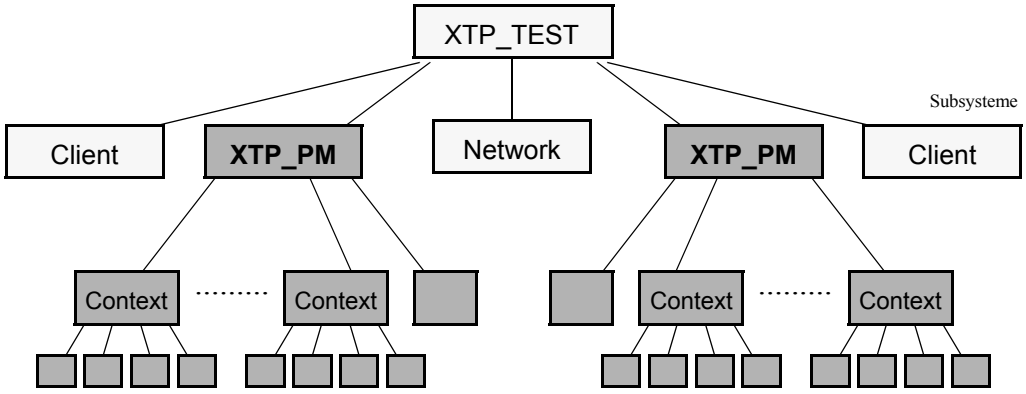


Abbildung 2-5: Die Modulinstanzhierarchie eines XTP-Testszenarios

Ziel bei der Erstellung der Spezifikation waren im Wesentlichen die Entwicklung einer geeigneten Dienstschnittstelle und die Erprobung der Protokollfunktionalität. Performanceaspekte wurden dabei nicht berücksichtigt. Dies macht sich besonders in der starken Strukturierung des Protokollautomaten in eine komplexe Modulhierarchie bemerkbar. Die sich daraus ergebende Notwendigkeit zum häufigen Austausch der Nutzdaten zwischen Modulen führt zu hohem Kopieraufwand. So wird ein Datenpaket allein auf seinem Weg vom Benutzermodul zum Netzwerk insgesamt siebenmal (!) kopiert, ebenso auf dem Weg vom Netzwerk zum Benutzermodul (siehe Abb. 2-6). Außerdem macht die Modularisierung zusammen mit der eingeschränkten Möglichkeit zu gemeinsamen Zustandsraumkomponenten in Estelle zusätzliche interne nachrichtenbasierte Interaktionen zwischen den Teilmodulen einer Protokollmaschine nötig.

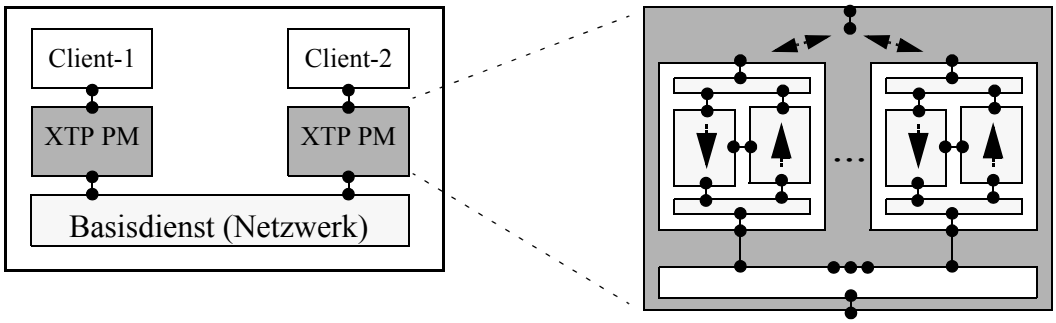


Abbildung 2-6: Die Modulinstanz- und Verbindungsstruktur der Estelle-Spezifikation

Die Estelle-Spezifikation hat mit ca. 9´000 Zeilen und insgesamt 184 Transitionsdefinitionen im Laufe ihrer Entwicklung einen für eine formale Spezifikation enormen Umfang erreicht. In Tabelle 2.2 sind einige statistische Angaben über die in der XTP-Protokollmaschine auftretenden Transitionsklauseln wiedergegeben. Insbesondere die hohe Zahl von `ANY`-Klauseln demonstriert, dass je nach Parametrierung (z. B. Maximalzahl gleichzeitig offener Verbindungen) und Systemzustand (z. B. Anzahl aktiver Kontexte) in dem System eine erhebliche Anzahl von Transitions- und Modulinstanzen existieren kann (siehe auch Abschnitt 4.4).

Aufgrund der großen Flexibilität von XTP hat das *Anwendungsprofil* der Benutzermodule großen Einfluss auf die Kommunikation zwischen den Protokollmaschinen. Entsprechend wurden in der Benchmarkspezifikation dynamisch parametrierbare, generische Benutzermodule erstellt. Diese erlauben die Simulation verschiedener Übertragungscharakteristika, wie z. B.

Tabelle 2.2: Statistik der Transitionsklauseln im XTP-Benchmark[a]

Typ	Klausel	Anzahl (abs.)	Anzahl (rel.)
Transitionen insgesamt		184	= 100 %
spontane Transitionen	–	52	28 %
Eingabe-Transitionen	`WHEN`	119	65 %
Delay-Transitionen	`DELAY`	13	7 %
Kontrollzustand	`FROM`	95	52 %
	`TO`	56	30 %
Prioritäten	`PRIORITY`	64	35 %
Any-Klauseln	`ANY`	19	10 %

a. Die Werte beziehen sich auf leere Benutzer- und Medien-Module

„Nachrichten-Ping-Pong“ oder die Übertragung großer Datenmengen („Bursts“). Die Einstellung des Benutzerprofils ist interaktiv oder über Parameterdateien möglich, so dass leicht mit verschiedenen Einstellungen experimentiert werden kann (siehe auch Abschnitt 4.3.7.3).

Wir beschließen damit dieses Kapitel und kommen im nächsten Kapitel zu der für diese Arbeit wichtigsten Grundlage, dem Estelle Implementierungs-Generator *XEC*.

3. Der Estelle-Compiler XEC

Um die Möglichkeiten einer effektiven automatischen Implementierung detailliert untersuchen, experimentell erproben und quantitativ bewerten zu können, haben wir den Estelle-Implementierungsgenerator XEC („*eXperimental Estelle Compiler*“, [ThGo98a], [The98], [The99]) entwickelt. Er bildet die Grundlage unserer Implementierungs- und Optimierungsexperimente und stellt eines der wesentlichen Ergebnisse dieser Arbeit dar.

Ursprünglich hatte die Entwicklung von XEC nur einen fragmentarischen Compiler für eine Teilmenge von Estelle zum Ziel, mit dessen Hilfe konkurrierend zu den bereits existierenden Estelle-Implementierungsgeneratoren einige einfache Estelle-Strukturen implementiert und quantitativ evaluiert werden sollten. Es zeigte sich jedoch bald, dass durch das hohe Abstraktionsniveau des zu Grunde liegenden Implementierungsansatzes in verhältnismäßig kurzer Zeit schließlich ein *vollständiger Estelle-Implementierungsgenerator* entwickelt werden konnte, der neben einer kompletten Umsetzung von Standard-Estelle auch diverse *Estelle-Erweiterungen* implementiert. Die durch das hohe Abstraktionsniveau des Implementierungsmodells erreichte Flexibilität, die Grundlage für die effiziente Anwendung der verschiedenen Implementierungs- und Optimierungsexperimente war, geht dabei nur in geringem Maße zu Lasten der Grundeffizienz des generierten Codes, sodass XEC auch im Vergleich zu den übrigen, zum Teil sehr implementierungsnah konzipierten Estelle-Implementierungsgeneratoren (speziell zum stark optimierenden EC) konkurrenzfähig ist [Dah96].

Aufbauend auf unseren Erfahrungen mit den existierenden Estelle-Implementierungsgeneratoren (siehe Abschnitt 2.1.4) haben wir die folgenden Anforderungen an ein solches Werkzeug identifiziert:

- Die Anwendung eines *leistungsorientierten Kodierungsschemas,* sowohl für den generierten Code als auch für das Laufzeitsystem, d.h. die generierte Implementierung sollte bereits ohne spezielle Optimierungsmaßnahmen eine hohe Effizienz erreichen. Dies ist nicht zuletzt auch eine wichtige Voraussetzung, um den Gewinn durch die in Kapitel 4 und 6 untersuchten Optimierungsmaßnahmen realistisch bewerten zu können.
- *Hohe Flexibilität* bezüglich der Codegenerierung und des Ausführungsmodells, um die Anwendung statischer und dynamisch-adaptiver Optimierungen zu unterstützen. Damit einher geht auch die Notwendigkeit eines *hohen Abstraktionsniveaus* sowohl des Codegenerators als auch des generierten Codes und der Laufzeitbibliothek, da nur so eine fehlerfreie Implementierung von Optimierungsmaßnahmen mit akzeptablem Aufwand möglich ist.
- *Geringer konzeptioneller Abstand* zwischen dem generierten Code und der Spezifikation. Dies erlaubt es zum einen, die Auswirkungen eines bestimmten Spezifikationsstils auf den generierten Code abzuschätzen, zum anderen erleichtert es erheblich eine mögliche manuelle Untersuchung bzw. Modifikation des generierten Codes.
- Unterstützung differenzierter *Leistungsmessungen* und *Ausführungsstatistiken* mit kontrollierbarem Overhead.

In den folgenden Abschnitten geben wir eine Einführung in Struktur und Arbeitsweise des *XEC-Toolkits* (XECTK) und ihres Kernstücks, des Estelle-Codegenerators XEC und demonstrieren dabei, wie die oben genannten Anforderungen in ihnen erfüllt werden.

Wir beginnen in Abschnitt 3.1 mit einer Übersicht über das XEC-Toolkit und seine Komponenten, in der insbesondere das Zusammenwirken der Komponenten überblicksweise dargestellt wird. In Abschnitt 3.2 gehen wir dann auf die Implementierungskonzepte des Codegenerators XEC, seiner Laufzeitbibliothek XECRT und des generierten Codes ein.

Die Abbildung statischer und dynamischer Strukturen der zu implementierenden Spezifikationen in die verwendete Zielsprache C++ sind dann Gegenstand der Abschnitte 3.3 und 3.4. Dabei illustrieren wir einige der Kernkonzepte mit realen Codefragmenten aus dem generierten Code und der Laufzeitbibliothek.

In Abschnitt 3.5 gehen wir kurz auf das zum Betrieb der Implementierung erforderliche Systemmanagement sowie die notwendige Plattformanbindung (insbesondere zur Frage der Zeitmodellierung) ein. Die im XEC-Toolkit integrierte technische Unterstützung von Ausführungsstatistiken und Ausführungszeitmessungen bis hin zu einem detaillierten und effizienten Monitoring mit Offline-Auswertung stellen wir in Abschnitt 3.6 vor.

Zuletzt geben wir in Abschnitt 3.7 einen kurzen Überblick über die im XEC-Toolkit realisierten Estelle-Erweiterungen, die wir größtenteils später an anderer Stelle jeweils motivieren und nochmals genauer vorstellen werden.

3.1. Übersicht

Im Folgenden geben wir eine erste Übersicht über Struktur und Arbeitsweise des „*eXperimental Estelle Compilers*“ *XEC*[1] und der mit ihm zusammenarbeitenden Werkzeuge. XEC selbst stellt dabei nur den Kern eines Programmsystems dar, das wir im Folgenden auch als *XEC-Toolkit* (*XECTK*) bezeichnen.[2]

Das XEC-Toolkit setzt sich primär[3] aus den folgenden Komponenten zusammen:

- PET (Compiler-Frontend)
- XEC (Codegenerator)
- XECRT (Laufzeitsystem für generierten Code)

Zusammen mit einem C++-Compiler ermöglicht das XEC-Toolkit durch die in Abb. 3-1 dargestellten Verarbeitungsschritte die automatische Generierung einer ausführbaren Implementierung („`spec`“) direkt aus einer Estelle-Spezifikation („`spec.stl`“).

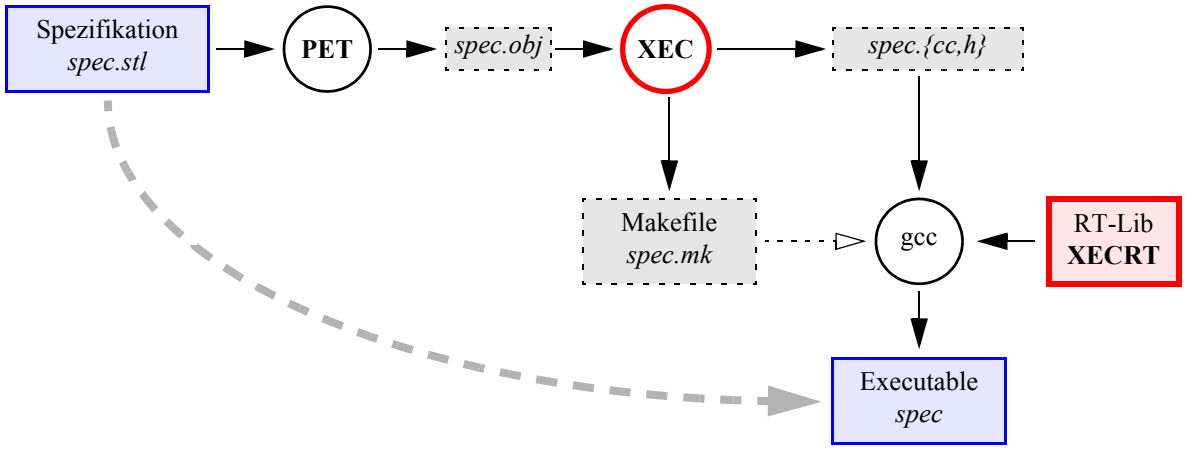


Abbildung 3-1: Generierung einer Estelle-Implementierung mit dem XEC-Toolkit

In den folgenden Abschnitten geben wir eine erste Übersicht über das Zusammenwirken der beteiligten Komponenten des XEC-Toolkits, soweit es zum Verständnis der später dargestellten Implementierungs- und Optimierungstechniken erforderlich ist.

---

1. XEC wurde anfangs unter dem Arbeitstitel „*Modular Estelle Translator*“ (MET) entwickelt.
2. In diesem Text wird an anderer Stelle gelegentlich auch das XEC-Toolkit als Ganzes (speziell aber die Kombination von XEC und XECRT) mit dem Begriff XEC bezeichnet.
3. Weitere (sekundäre) Komponenten sind die Monitoring-Tools (u.a. PATO, siehe Abschnitt 3.6.3 und Abschnitt 3.6.3) und das Java-Debugger-Frontend (siehe Abschnitt 3.6.5).

3.1.1 Der Compiler-Frontend PET

Der „*Portable Estelle Translator*“ *PET* des XEC-Toolkits ist eine erweiterte Version des gleichnamigen Tools aus dem bereits in Abschnitt 2.1.4 genannten Estelle-Implementierungswerkzeug PET/DINGO [SiSt93]. PET liest als *Compiler-Frontend* Estelle-Spezifikationen in Textform, analysiert und überprüft ihre Syntax anhand der kontextfreien und kontextsensitiven Anteile der Estelle-Grammatik und bildet intern eine zur syntaktischen Struktur der Spezifikation äquivalente baumartige[4] Grafendarstellung, deren Knoten Instanzen der *PET-Klassenbibliothek* sind.

Diese interne syntaktische Repräsentation kann dann im Sinne einer *persistenten Objektrepräsentation* in eine Datei (*PET-Objektdatei* “`*.obj`“, siehe Anhang A.1.1) geschrieben bzw. wieder aus dieser restauriert werden, um sie mit anderen Werkzeugen (z. B. XEC, s. u.) weiterverarbeiten zu können. Zudem kann die interne Repräsentation auch wieder in einen Estelle-Quelltext umgewandelt werden (siehe auch Abschnitt 7.4.4).

Ausgehend von der Urversion von PET aus dem genannten PET/DINGO-Toolkit wurde PET im Rahmen der Entwicklung von XEC in großen Teilen neu kodiert[5] (siehe Anhang A.1) und insbesondere zur Unterstützung mehrerer Estelle-Erweiterungen (siehe Abschnitt 3.7 und Anhang A.1.3) deutlich erweitert. In Tabelle 3.4 auf Seite 117 sind die zu ihrer Aktivierung erforderlichen PET-Optionen dargestellt.

3.1.2 Der Codegenerator XEC

Der „*eXperimental Estelle Compiler*“ *XEC* bildet als Codegenerator, welcher die oben beschriebenen PET-Objektdateien in objektorientierten C++-Quellcode übersetzt, den wesentlichen Kern des XEC-Toolkits (siehe Abb. 3-1 auf Seite 25).

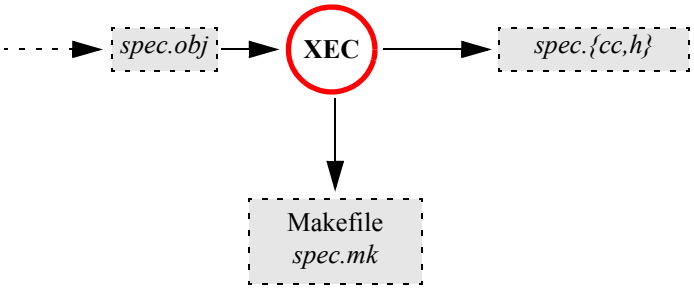


Abbildung 3-2: Codegenerierung durch XEC

4. Die Nachbildung des rekursiv definierten Syntaxbaumes des kontextfreien Anteils der Estelle-Grammatik ergibt zunächst auch in der internen Darstellung eine *echte Baumstruktur*, die jedoch durch Aufnahme von Querverweisen aus der kontextsensitiven Grammatik (z. B. Verweise von Namen auf die von ihnen referenzierten Definitionen) die Baumeigenschaft verliert.

5. Dies schließt insbesondere eine komplett neue Kommandoschnittstelle, einen neuen Scanner mit voller Text-Include-Unterstützung (siehe Anhang A.1.2) und eine verbesserte Fehlerausgabe ein.

3.1.2.1 Die Interne Struktur von XEC

Zum Laden der PET-Objektdatei und zur Generierung und Handhabung der daraus hervorgehenden baumartigen PET-Objektstruktur wird auf die PET-Klassenbibliothek zurückgegriffen. Zur Codegenerierung erzeugt XEC dann durch einen rekursiven Konstruktionsvorgang eine zu einem Teil der PET-Objektstruktur *duale* Objektstruktur (*XEC-Objektstruktur*), welche die bei der Codegenerierung wesentlichen Strukturen der Spezifikation durch Instanzen der *XEC-internen Klassenbibliothek*[6] nachbildet (siehe Abb. 3-3). Diese XEC-Objekte enthalten dabei jeweils auch interne Referenzen auf die zu Grunde liegenden PET-Objekte.

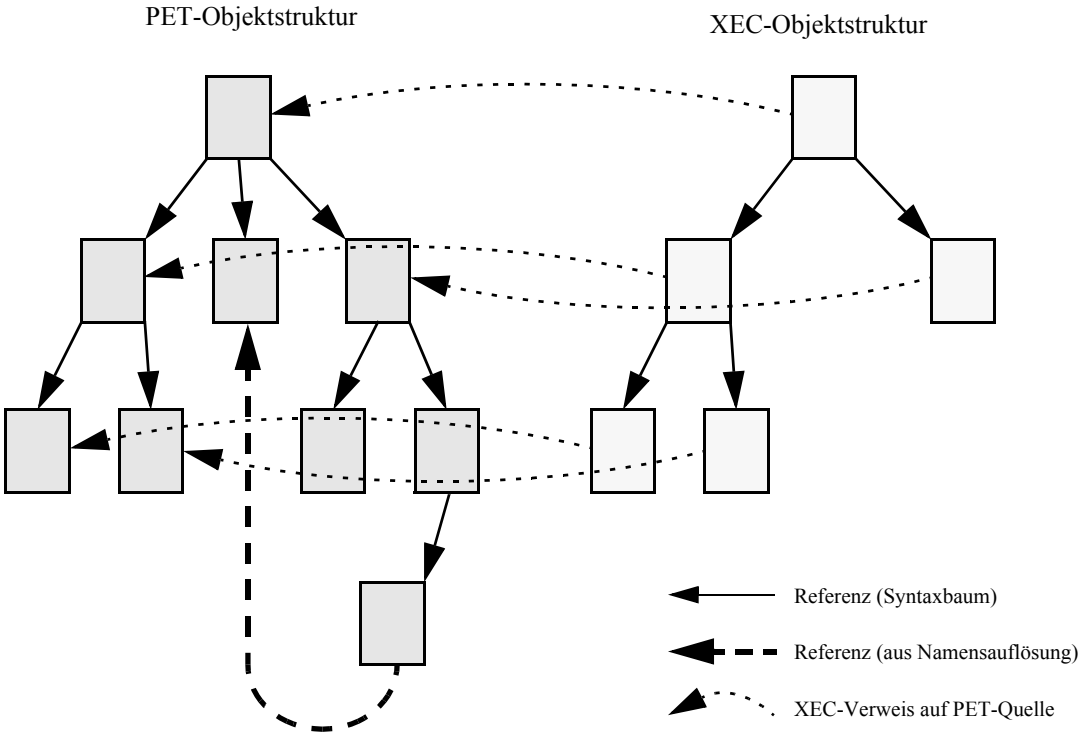


Abbildung 3-3: PET-Objektstruktur und daraus generierte XEC-Objektstruktur

Zur leichteren Unterscheidung zu Klassennamen der PET-Klassenbibliothek sind die Namen der Estelle-Syntax-bezogenen XEC-Klassen mit dem Präfix „`C`“ versehen. Die Bedeutung der einzelnen Klassen ergibt sich weitgehend aus ihren Namen (s. u.). Die Klassendefinitionen sind thematisch gruppiert in die in Tabelle A.33 auf Seite 387 aufgelisteten Quelldateien aufgeteilt, deren Namen ebenfalls durch das Präfix „`c`“ markiert sind und die so insbesondere leicht von den Dateien der Laufzeitbibliothek (Präfix „`xecrt_`“, siehe Abschnitt 3.1.3) unterschieden werden können.

Neben denjenigen syntaktischen Elementen, die in der XEC-Klassenbibliothek eine explizite Repräsentation finden, erfolgt die Behandlung aller übrigen syntaktischen Estelle-Strukturen (z. B. Typdefinitionen, Variablen, Statements, Expressions) direkt anhand der PET-Objektstruktur durch Methoden der Klasse `CDeclDef`. Zur Unterstützung insbesondere dieses Vorgangs existiert mit der Klasse `DefAssocNode` ein System zur *Rückassoziation* von PET-Objektreferenzen auf zugehörige XEC-Objektreferenzen, die u. a. für die Generierung ausreichend qualifizierter C++-Namen bei der Codegenerierung zuständig ist.

6. Die XEC-interne Klassenbibliothek enthält dazu 21 Klassendefinitionen zur Repräsentation von Estelle-Strukturen.

3.1.2.2 Funktionsprinzip der Codegenerierungsmethoden

Ausgehend von der aus der PET-Objektdatei rekonstruierten (und in sich geschlossenen) PET-Objektstruktur und der von XEC daraus abgeleiteten XEC-Objektstruktur wird dann durch Aufruf der Methode „`Compiler::translate(ostream& osH, ostream& osC)`“ der eigentliche Codegenerierungsvorgang[7] gestartet.

Dabei erzeugt XEC aus einer Estelle-Spezifikation zwei C++-Dateien:

- eine C++-Header-Datei mit Deklarationen[8] (Dateinamenserweiterung „`.h`“) und
- eine C++-Datei mit den zugehörigen Definitionen (Dateinamenserweiterung „`.cc`“).[9]

Die in den oben genannten XEC-Objekten jeweils enthaltenen Methoden zur rekursiven Generierung von C++-Quelltexten erzeugen entsprechend C++-Quellcode für diese beiden Ausgaben-Dateien. Dazu werden an diese Methoden (wie bereits auch schon an `Compiler::translate`) jeweils Referenzen auf zwei C++-Ausgabeströme (Klasse `ostream`) übergeben, über die die Codegenerierung für beide Dateien erfolgen kann. Entsprechend ergibt sich die Signatur dieser Codegenerierungsfunktionen in den meisten Fällen nach folgendem Beispiel:

Beispiel 3.3-a: Signatur der Codegenerierungsmethode für Modulkörper

```
void CModuleBody::compile (
        ostream& osH,
        ostream& osC,
        const Indent& indH,
        const Indent& indC
);
```

(Ende von Beispiel 3.3-a)

Die nach den beiden Ausgabestrom-Referenzen übergebenen Referenzen auf `Indent`-Objekte (`indH` und `indC`) dienen zur Formatierung der in die beiden Ausgabeströme erzeugten C++-Quelltexte, indem sie innerhalb des rekursiven Codeerzeugungsprozesses jeder `compile`-Methode ermöglichen, einerseits den selbst generierten Code gesteuert durch ihren Aufrufer textuell korrekt einzurücken, und andererseits bei der Generierung von darin eingeschachteltem Code durch rekursiven Aufruf einer anderen `compile`-Methode diesen gegebenenfalls relativ zur eigenen Einrückung ggf. tiefer einzurücken.[10]

---

7. Genauer betrachtet ist die Erzeugung der XEC-Objektstruktur ebenfalls Teil des Aufrufs von `Compiler::translate(...)`.

8. C++ unterscheidet wie auch C zwischen *Deklarationen*, die lediglich eine (äußere) Schnittstelle festlegen, und *Definitionen*, welche die Deklaration vervollständigen bzw. interne Aspekte hinzufügen. So ist zum Beispiel in C „`int f(int);`“ eine Funktions-Deklaration und „`int f(int n) {return n+1;}`“ eine dazu passende Funktions-Definition.

9. Die Dateinamenserweiterung „`.cc`“ wurde zur eindeutigen Unterscheidung von C++-Quelldateien und C-Quelldateien gewählt, da sie von den meisten modernen C++-Compilern korrekt erkannt wird und im Gegensatz zu der ebenfalls gebräuchlichen Dateinamenserweiterung „`.C`“ auch auf Plattformen ohne zuverlässigen Erhalt der Groß-/Kleinschreibung bei Dateinamen (z. B. MS-DOS/MS-Windows) eine eindeutige und dauerhafte Unterscheidung erlaubt.

10. In einer kommenden Version von XEC wird voraussichtlich die explizite Übergabe und Ausgabe der Einrückungsobjekte in die interne Verwaltung der Ausgabeströme integriert, soweit dies ausreichend portabel möglich ist.

Dazu enthält die Klasse `Indent` zwei essentielle Methoden:

- Einen Konstruktor, der die Erzeugung einer tieferen Einrückung bezogen auf eine gegebene Einrückung erlaubt.
- Eine Ausgabefunktion, mit deren Hilfe `Indent`-Objekte in Ausgabeströme ausgegeben werden können.

Als Beispiel für die Anwendung der Klasse `Indent` bei der rekursiven Generierung von C++-Code betrachten wir nun in Beispiel 3.3-b einen kleinen Ausschnitt aus der Methode `CModuleBody::compile`, die die Codegenerierung eines Modul-Rumpfes abwickelt.

Beispiel 3.3-b: Auszug aus Methode `CModuleBody::compile`

```
void CModuleBody::compile(      ostream& osH,
                                ostream& osC,
                                const Indent& indH,
                                const Indent& indC
) {
        Indent subindH("\t", indH);

        osH << indH << "class ";
        dump_static_name(osH, true);
        // ...
        osH << " {\n";
        osH << indH << "    public:\n";
        // ...

        /* compile Headers */
        for (   CHeader* pHeader = HeaderList.first() ;
                pHeader ;
                pHeader = pHeader->next()
        )
                pHeader->compile(osH, osC, subindH, indC);
        // ...
        osH << indH << "};\n";
}
```

(Ende von Beispiel 3.3-b)

Die lokale `Indent`-Instanz `subindH` stellt eine Einrückung dar, die um genau eine Tabulatorposition tiefer eingerückt ist als das als Parameter übergebene `indH` (also die Grundeinrückung bei Ausgaben in den Ausgabestrom `osH`). Diese wird dann u. a. für eingeschachtelte `compile`-Aufrufe verwendet. Im Beispiel 3.3-c ist ein mögliches Ergebnis dieser Codegenerierung (ebenfalls ausschnittsweise) wiedergegeben.

Beispiel 3.3-c: Auszug aus dem Codegenerierungsergebnis von Beispiel 3.3-b

```
class BODY_testbody /* ... */ {
    public:
        // ...
        class HEADER_testheader /* ... */ {
                // ...
        };
        // ...
};
```

(Ende von Beispiel 3.3-c)

Dieses Beispiel zeigt neben der Grundidee bei der Generierung des formatierten C++-Quelltexts in den beiden Ausgabedateien bereits auch ansatzweise die Aufbereitung der syntaktischen Struktur der zu übersetzenden Spezifikation in der oben bereits genannten XEC-Objektstruktur, die dual zu Teilen der PET-Objektstruktur existiert. So enthält zum Beispiel jede Instanz von `CModuleBody` (als XEC-Repräsentation eines Estelle-Modulrumpfs) mit der Komponente `HeaderList` eine Liste der darin enthaltenen Modulheader-Definitionen (genauer: ihrer XEC-Repräsentation `CHeader`).

Wir werden auf ausgewählte Konzepte der Codegenerierung von XEC und der Interaktion mit dem Laufzeitystem XECRT später in Abschnitt 3.2 genauer eingehen.

3.1.2.3 Anwendung und Steuerung von XEC

Ausgehend von einer PET-Objektdatei generiert XEC (siehe Anhang A.2) passend zum verwendeten Sprachumfang vollautomatisch die beiden o. g. C++-Quelldateien und ein Makefile, mit dem die spätere Übersetzung der C++-Quellen durch einen externen Compiler gesteuert wird (siehe Anhang A.8).

Die Codegenerierung erfolgt im einfachsten Fall durch Aufruf von XEC mit dem Namen der PET-Objektdatei als Parameter (siehe Abb. 3-4), wodurch Basisname und Verzeichnis der Quelldateien auch den Basisnamen der generierten Dateien bilden.

```
# xec ./test.obj
>> XEC (V 1.3.3)
>> reading "./test.obj"
>> creating c++-files "./test.cc" and "./test.h"
>> creating makefile "./test.mk" for goal "./test"
>> translation complete
```

Abbildung 3-4: Einfacher Übersetzungslauf von XEC

Die weitere Übersetzung erfolgt dann durch einen Aufruf des UNIX-Werkzeugs „`make`“ (für das obige Beispiel mit „`make -f test.mk`“), welches die notwendigen Aufrufe zur Übersetzung des generierten Quellcodes und der Laufzeitbibliothek steuert.

Die Übersetzung der Laufzeitbibliothek erfolgt dabei zusammen mit der Übersetzung des generierten Quellcodes, um sicherzustellen, dass alle Komponenten jeweils passend zum Satz an verwendeten Compileroptionen und dadurch aktivierten Codekomponenten übersetzt werden.

Zu diesem Zweck setzt XEC *Makefile-Templates* ein, die Zusammenstellungen von Optionen für bestimmte Anwendungen beinhalten. Die Auswahl, welches Makefile-Template eingesetzt wird, erfolgt durch den optionalen XEC-Parameter „`-m <opt>`“, wobei für <*opt*> einer der in Tabelle A.36 auf Seite 407 angegebenen, bereits vordefinierten Namen übergeben werden kann. Defaultwert ist „`-m default`“, Alternativen sind u. a. „`monitor`“, „`optimize`“ oder „`debugger`“.

Die Makefile-Templates steuern Voreinstellungen für diverse Parameter wie z. B.

- Codegenerierungs-Optionen des C++-Compilers (z. B. zur Peephole-Optimierung)[11]
- Umfang der Unterstützung für Tracing, Logging und Statistiken (siehe Abschnitt 3.6.3)
- Monitoring- und Debugger-Support (siehe Abschnitt 3.6.5)
- Nutzung von Prototypen für `PRIMITIVE-` (bzw. `EXTERNAL-`) Funktionen und Prozeduren (siehe Abschnitt 3.5.3)
- Übertragung von Inline-Code aus Pseudokommentaren (siehe Abschnitt 3.5.3)

Die einzelnen Parameter können darüber hinaus auch im generierten Makefile nachträglich bearbeitet werden. Das generierte Makefile enthält dabei nur die Definition der notwendigen Parameter, alle übrigen Definitionen (z. B. der Dateiabhängigkeiten und Regeln) werden aus einer Include-Datei der Laufzeitbibliothek bezogen. Diese greift dabei auch auf die durch ein `configure`-Script[12] bei der Installation des Toolkits gewonnenen systemspezifischen Parameter zurück (siehe Anhang A.8).

Wir beschließen damit an dieser Stelle die Ausführungen zur Arbeitsweise des Codegenerators XEC und wenden uns dem Laufzeitsystem XECRT der generierten Implementierungen zu.

3.1.3 Das Laufzeitsystem XECRT

Das Laufzeitsystem XECRT bildet die Grundlage für die Ausführung des von XEC generierten Codes (siehe Abb. 3-1 auf Seite 25). Das Laufzeitsystem unterstützt dabei neben verschiedenen Implementierungs- und Optimierungsmodellen auch diverse Zusatzfunktionen zum Logging, Debugging und Monitoring der erzeugten Implementierungen.

Die Quelldateien des Laufzeitsystems sind zur Unterscheidung von den Quellen von XEC selbst mit dem Dateinamenspräfix „`xecrt_`“ gekennzeichnet. In Tabelle A.34 auf Seite 388 sind die einzelnen Quelldateien und die darin definierten Klassen aufgelistet.

Im Wesentlichen kann man zwei Kategorien von Klassen in der XECRT-Bibliothek unterscheiden:

- Basisklassen für die von XEC generierten Klassen (zur Repräsentation von syntaktischen Elementen der Estelle-Spezifikation) und
- Klassen zur Implementierung umgebungsspezifischer Dienste

---

11. z. B. die Option “`-O2`“ für den C++-Compiler `gcc`
12. basierend auf GNU-`autoconf`

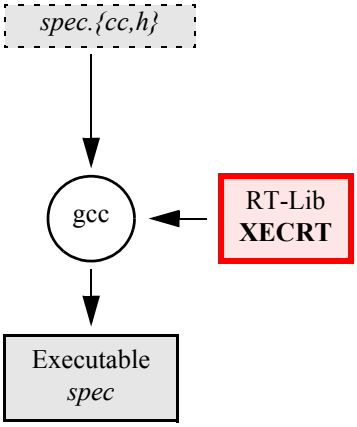


Abbildung 3-5: Integration des Laufzeitsystems XECRT in den generierten Code

Ein typisches Beispiel für die erste Kategorie ist die Klasse `Specification`, die Basisklasse der jeweils von XEC generierten und eine Spezifikation nachbildenden Klassendefinition ist. Solche Basisklassen bilden also gewissermaßen das Grundgerüst, auf dem die Implementierung der konkreten Estelle-Spezifikation aufbaut. Alle Klassen dieser Kategorie sind in den Quelldateien „`xecrt_modules.*`“ und „`xecrt_estelle.*`“ zu finden.

Die zweite Kategorie von Klassen hat dagegen keine direkte Entsprechung in der Estelle-Syntax, sondern dient dazu, die Ausführung der generierten Implementierung auf einer konkreten Hard- und Software-Plattform zu unterstützen, mit dieser zu kommunizieren und Zusatzdienste wie Logging und Debugging zu bieten. Entsprechend sind die Klassen dieser Kategorie in den Quelldateien „`xecrt_context.*`“ und „`xecrt_debugger.*`“ zu finden.

In den folgenden Abschnitten 3.2 bis 3.4 beschäftigen wir uns zunächst mit der ersten Kategorie von Laufzeitbibliotheks-Klassen und wie diese mit dem von XEC generierten Code zusammenarbeiten, um eine semantisch korrekte Implementierung der Ausgangsspezifikation zu erreichen.

Die Plattformanbindung durch die zweite Kategorie von Laufzeitbibliotheks-Klassen wird dann Gegenstand von Abschnitt 3.5 und 3.6 sein, wobei als Zielplattform primär *UNIX*-kompatible Betriebssysteme (insbesondere *Linux* und *Sun-Solaris*) im Vordergrund stehen.

3.2. Implementierungskonzepte

Wir betrachten nun die Konzepte hinter den von XEC generierten Implementierungen und ihr Zusammenwirken mit der Laufzeitbibliothek XECRT. Dabei werden wir so weit wie möglich von den Details des Codegenerators selbst abstrahieren, was – wie wir sehen werden – durch die besondere Struktur der generierten Implementierungen wesentlich unterstützt wird. Stattdessen diskutieren wir im Folgenden einige ausgewählte Aspekte des *Kodierungsschemas* des von XEC generierten Codes und sein Zusammenwirken mit der Laufzeitbibliothek. Hierbei wird die konforme Umsetzung der abstrakten Estelle-Syntax und -Semantik im Vordergrund stehen.

3.2.1 Interaktion von Laufzeitbibliothek und generiertem Code

Wie wir bereits im vorangegangenen Abschnitt gesehen haben, generiert XEC aus der in einer PET-Objektdatei enthaltenen Estelle-Spezifikation objektorientierten C++-Code. Dieser generierte Code und die ihm zu Grunde liegende Laufzeit-Klassenbibliothek XECRT interagieren dabei einerseits sehr intensiv miteinander, andererseits werden ihre Aufgaben aber auch streng durch die folgenden Grundkonzepte aufgeteilt:

(i) Der *generierte Code* spiegelt die zu Grunde liegende Spezifikation *syntaktisch* 1:1 wider, d.h. aus einer Zeile Spezifikationstext wird typischerweise jeweils eine Zeile C++-Code erzeugt.

(ii) Allein die *Laufzeitbibliothek* steuert die Ausführung der Implementierung (gemäß der impliziten Estelle-*Semantik*), d.h. sie hat vollständige Kontrolle über die semantische Interpretation der Spezifikation anhand des generierten Codes.

(iii) Entsprechend werden Optimierungen, Plattformanbindung und sonstige nicht direkt spezifikationsbezogenen Aspekte ausschließlich in der *Laufzeitbibliothek* realisiert.

Die Motivation für diese drei Grundkonzepte besteht darin, den generierten Code (und damit auch den Codegenerator) so einfach wie möglich zu halten, indem zunächst nur die *explizit* (d.h. *syntaktisch*) im Spezifikationstext angegebenen Aspekte modelliert werden.

Die Regeln zur *Ausführung der Spezifikation* (bzw. des generierten Codes als deren direktes Abbild) sind dagegen nicht Bestandteil der Estelle-Spezifikation selbst, sondern implizit als feste Regeln zu ihrer semantischen Interpretation definiert. Diese Regeln (also die *Semantik* der FDT Estelle[13]) beschreiben, wie die syntaktischen Strukturen einer Spezifikation interpretiert werden, um schließlich mögliche Zustandsfolgen einer konformen Implementierung zu erhalten.

Entsprechend trennt das XEC-Toolkit strikt zwischen

- den *explizit* im Spezifikationstext angegebenen (*syntaktischen*) Aspekten *(i)*, die – da sie individuell für die konkrete Spezifikation stehen – bei der Codegenerierung umgesetzt werden müssen, und

13. Die Semantik der FDT Estelle (beschrieben durch die Abschnitte 5 und 9 von [ISO97]) ist nicht zu verwechseln mit der Semantik einer *konkreten* Estelle-Spezifikation (eine Menge von Ausführungssequenzen): Erstere ist eine Menge von Regeln zur Bildung der letzteren anhand einer konkreten Estelle-Spezifikation, sie ist also der Meta-Ebene der konkreten Estelle-Spezifikationen zuzuordnen.

- den *impliziten* Regeln zur (*semantischen*) Interpretation dieser Spezifikation *(ii)*, die unabhängig von einer konkreten Spezifikation angegeben und daher u. U. weitgehend unabhängig von der eigentlichen Codegenerierung implementiert werden können.
- Diese Trennung von syntaktischen Aspekten im generierten Code und semantischen Aspekten in der Laufzeitbibliothek gilt umso mehr für Aspekte der *Anbindung an eine konkrete Implementierungsplattform (iii)*, da diese sogar nur indirekt aus der Semantik der Spezifikation hervorgehen. So beinhaltet die Estelle-Semantik z. B. nur einen sehr abstrakten Zeitbegriff, der auf Implementierungsebene jedoch auf ein konkretes Zeitmodell abgebildet werden muss (siehe auch Abschnitt 3.5.2). Ein solches Modell wie z. B. das Voranschreiten einer plattformspezifischen Echtzeituhr bei der Ausführung der Implementierung (mit Parametern wie *Auflösung, Genauigkeit, Ausführungsdauer von Operationen* etc.) liegt weit jenseits der syntaktischen Aspekte des Spezifikationstextes[14] und wird deshalb – wie alle impliziten semantischen Aspekte – ebenfalls ausschließlich in der Laufzeitbibliothek realisiert.

Diese Strategien unterscheidet das XEC-Toolkit wesentlich von anderen Estelle Implementierungsgeneratoren[15] wie z. B. DINGO (siehe Abschnitt 2.1.4), die im generierten Code oft auch bereits die Regeln zur Implementierung der Estelle-Semantik beinhalten. Die Laufzeitbibliotheken dienen entsprechend bei diesen Implementierungsgeneratoren nur zur Bereitstellung von *passiven* Klassen und Funktionen, die unter weitgehender Kontrolle des generierten Codes genutzt werden (siehe Abb. 3-6). Wir werden sehen, dass solche Strategien die Komplexität des Codegenerators erheblich steigern und gleichzeitig die Flexibilität bei der Auswahl und Realisierung von Implementierungstechniken ganz wesentlich reduziert wird.

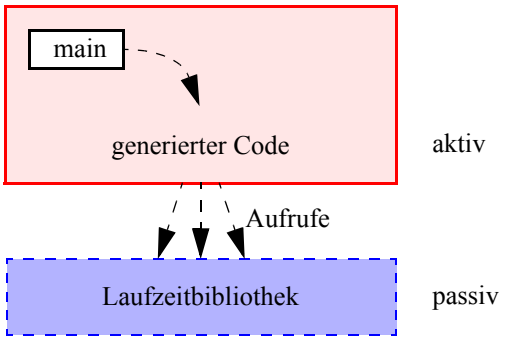


Abbildung 3-6: Aufrufhierarchie zwischen (aktivem) generiertem Code und (passiver) Laufzeitbibliothek (z. B. bei DINGO)

Beim XEC-Toolkit erfolgt die Implementierung der Estelle-Semantik dagegen ausschließlich durch die in der Laufzeitbibliothek enthaltene C++-Klassenbibliothek. Diese implementiert als Schnittstelle zum generierten Code eine Menge von (größtenteils polymorphen) Klassen und Klassen-Templates, die meist als *Basisklassen* für die vom Codegenerator anhand der Estelle-

14. Der einzige Bezug auf ein konkretes Zeitmodell im Spezifikationstext ist die `TIMESCALE`-Definition. Das dort angegebene Argument (z. B. „`TIMESCALE Seconds;`“) hat jedoch semantisch reinen Kommentarcharakter (siehe auch Abschnitt 7.4.3).

15. Der Implementierungsgenerator EC (siehe Abschnitt 2.1.4) nutzt scheinbar im Bereich der Transitionsauswahl zumindest teilweise eine zu XEC ähnliche Beziehung zwischen generiertem Code und Laufzeitbibliothek. Genauere Analysen waren jedoch aufgrund der nicht öffentlich zugänglichen Quellen von Laufzeitbibliothek und Codegenerator und dem sehr geringen Abstraktionsgrad des generierten Codes nicht möglich.

Spezifikation generierten Klassen dienen. Entsprechend gibt es sowohl bezüglich der Objektrelationen als auch bezüglich der Aufruf-Hierarchien eine *sehr starke Verflechtung* zwischen generiertem Code und der Laufzeitbibliothek, die dem Laufzeitsystem den Charakter eines *Frameworks* [FaSch97] zur Implementierung von Estelle-Spezifikationen gibt.

So geht die Kontrolle der Ausführung der Spezifikation und damit des Aufrufs der aus der Estelle-Spezifikation generierten C++-Methoden beim XEC-Toolkit durchgängig von der (aktiven) Laufzeitbibliothek aus. Insbesondere ist z. B. auch die `main`-Funktion, also der Eintrittspunkt des Programm-Kontrollflusses, ein Bestandteil der Laufzeitbibliothek (siehe Abb. 3-7).

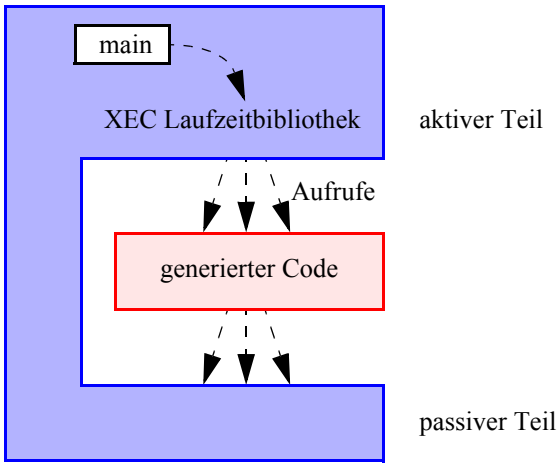


Abbildung 3-7: Aufrufhierarchie beim XEC-Toolkit zwischen (hauptsächlich aktiver) Laufzeitbibliothek und generiertem Code

Ganz offensichtlich besteht zwischen dem von XEC generierten Code und der externen Schnittstelle der XEC-Laufzeitbibliothek eine sehr *enge Verflechtung*. Entsprechend wurden der Codegenerator und die Laufzeitbibliothek in einem sukzessiven Entwicklungsprozess gemeinsam entwickelt und bilden gemäß der jeweiligen Version des XEC-Toolkits eine Einheit. Andererseits können durch die relative Einfachheit und Abstraktion dieser Schnittstelle viele Anpassungen am Ausführungs- und Optimierungsmodell der Laufzeitbibliothek unter Beibehaltung dieser gemeinsamen Schnittstelle und damit ohne Modifikationen des generierten Codes und des Codegenerators realisiert werden. Dies ist die wesentliche Grundlage der *Flexibilität* des XEC-Toolkit bezüglich der Implementierung von Optimierungsmaßnahmen.

Die Realisierung von solchen Optimierungsmaßnahmen in der Laufzeitbibliothek hat darüber hinaus noch den Vorteil, dass die dabei anzupassenden und zu implementierenden Software-Komponenten direkt auf der Ebene des Ausführungsmodells kodiert werden können. Eine Implementierung von Optimierungsmaßnahmen im generierten Code würde es dagegen erforderlich machen, dass diese Softwarekomponenten – genau wie der übrige generierte Code (siehe Beispiel 3.3-b und Beispiel 3.3-c auf Seite 30) – auf der *Meta-Ebene des Codegenerators* implementiert werden müssten, also Code zur Erzeugung des Codes auf der Ebene des Ausführungsmodells entwickelt werden muss. Eine solche Vorgehensweise wäre offensichtlich wesentlich *aufwändiger* und *fehleranfälliger* und würde gerade bei den zum Teil sehr komplexen Optimierungsmaßnahmen auch schnell an die Grenze ihrer Handhabbarkeit stoßen.[16]

16. Eine Implementierung der Spezifikation selbst auf der Meta-Ebene des Codegenerators ist dagegen naheliegenderweise unvermeidlich; sie wird jedoch im Gegensatz zu den zum Teil komplexen Optimierungsmaßnahmen durch die Einfachheit des Codegenerierungsschemas (weitgehend eine 1:1-Abbildung der Spezifikationssyntax) wesentlich erleichtert (s. u.).

Natürlich hat diese strenge Trennung auch Nachteile bezüglich der Möglichkeiten zur globalen Optimierung des generierten Codes, da die auf Ebene der Laufzeitbibliothek möglichen Optimierungsmaßnahmen vor allem organisatorische Aspekte der Ausführung der Spezifikation und die Implementierung von Basisfunktionalitäten betreffen. Wir nehmen diese Beschränkungen jedoch an dieser Stelle bewusst in Kauf zu Gunsten der Einfachheit des Codegenerators, des hohen Abstraktionsniveaus von generiertem Code und Laufzeitbibliothek und der daraus resultierenden guten Wartbarkeit und Zuverlässigkeit des Gesamtsystems.

3.2.2 Grundzüge des Kodierungsschemas

Auf der Basis des XEC zu Grunde liegenden *Kodierungsschemas* wird aus einer Estelle-Spezifikation eine C++-Implementierung gewonnen, die in vielen Aspekten wie Struktur, Datentypen, Namen und Namensräumen der Estelle-Spezifikation so weit entspricht, dass man unter Kenntnis des Kodierungsschemas z. B. leicht aus dem generierten C++-Code manuell die Ausgangs-Estelle-Spezifikation[17] zurückgewinnen könnte. Wir werden dies im Folgenden anhand einiger Beispiele illustrieren.

Die starke Entsprechung der Ausgangs-Estelle-Spezifikation und des daraus von XEC abgeleiteten C++-Codes hat natürlich in erster Linie den großen Vorteil, dass die Komplexität des Codegenerators XEC selbst deutlich verringert werden konnte. So realisiert XEC im Wesentlichen eine Codegenerierung durch schiere Abbildung von syntaktischen Elementen der Estelle-Spezifikation auf die durch das Kodierungsschema definierten *C++-Muster*, wobei Strukturen und Namen der Ausgangsspezifikation praktisch vollständig erhalten bleiben.

Diese konzeptionelle Einfachheit des Codegenerators senkt dabei nicht nur die Wahrscheinlichkeit für Fehler im Codegenerator selbst, sondern sie macht durch die resultierende *Verfolgbarkeit* des Implementierungsvorgangs und die damit verbundene Einfachheit und Reproduzierbarkeit des generierten Codes auch eine Überprüfung der „*offensichtlichen Korrektheit*“ des Codegenerators erst möglich.

Dies wird ergänzt durch das hohe *Abstraktionsniveau* des generierten C++-Codes, durch das sich syntaktische und semantische Anforderungen von Estelle häufig auch in äquivalenten Anforderungen auf C++-Ebene wieder finden. So kommt z. B. das C++-Kodierungsschema praktisch ohne explizite Typkonvertierungen aus und bildet so Aspekte wie die Typ-Kompatibilität praktisch 1:1 von Estelle auf C++ ab. Auf diese Weise wird die Korrektheit des Implementierungsgenerators in erheblichem Umfang auch nochmals durch den nachgeschalteten C++-Compiler validiert, da z. B. die Erzeugung einer falschen Typreferenz durch den Codegenerator in den meisten Fällen zu unzulässigem C++-Code führen und entsprechend vom nachgeschalteten C++-Compiler erkannt werden würde.

Diese Aspekte, die Einfachheit und Klarheit des Kodierungsschemas und die damit verbundene Überprüfbarkeit des generierten Codes anhand seiner „offensichtlichen Korrektheit“ auf der einen Seite und das hohe Abstraktionsniveau des generierten C++-Codes und die damit verbundene weitgehende Überprüfbarkeit der Konsistenz durch den nachgeschalteten C++-Compiler auf der anderen Seite, waren unter den zur Verfügung stehenden Entwicklungs-Ressourcen überhaupt die wesentlichen Grundlagen zur Entwicklung von XEC zu einem vollständigen und zuverlässigen Estelle-Implementierungs-Generator.

17. bzw. eine dazu äquivalente Spezifikation

3.2.2.1 Grundzüge der Implementierung der Modulhierarchie

Als Beispiel zum Kodierungsschema betrachten wir das Grundprinzip der Abbildung der verschachtelten statischen *Modulhierarchie* einer Estelle-Spezifikation, die von XEC gemäß des Kodierungsschemas in eine identisch strukturierte Hierarchie von *verschachtelten C++-Klassen* abgebildet wird (siehe Abb. 3-8). Dadurch kann das Namensraumsystem von Estelle (z. B. „`body B`“ in „`body A`“ in „`specification S`“) vollständig in das qualifizierte Namensraumsystem von C++ („`SPEC_s::BODY_a::BODY_b`“) übertragen werden, ohne künstliche Namenserweiterungen (z. B. zur Unterscheidung der namensgleichen aber verschiedenen Module „`SPEC_s::BODY_a::BODY_b`“ und „`SPEC_s::BODY_b`“) einführen zu müssen.[18]

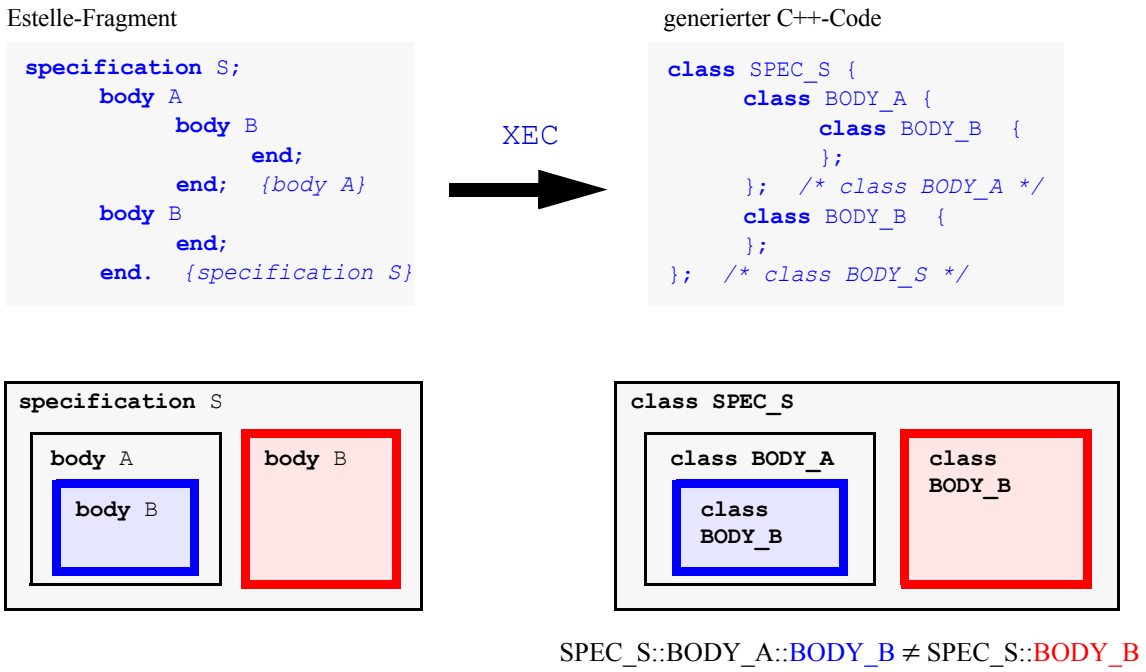


Abbildung 3-8: Prinzip der Modellierung der Spezifikationsstruktur durch XEC

Entsprechend einfach gelingt es, die Strukturen der Ausgangsspezifikation und der Implementierung zu vergleichen und somit die Implementierung der Systemarchitektur und damit auch der darin enthaltenen Komponenten zu verfolgen. Ein weiterer Vorteil dieser Abbildung ist die Möglichkeit, Estelle-Module direkt als eigenständige C++-Klassen zu definieren und somit die Estelle-Konzepte des Moduls und der Modulinstanz nahtlos in die C++-Konzepte der Klasse und ihrer Instanzen abzubilden. Wir werden diesen Aspekt in Abschnitt 3.3.2 nochmals genauer untersuchen.

3.2.2.2 Grundzüge der Implementierung von Transitionen

Als zweites charakteristisches Beispiel für das Kodierungsschema von XEC betrachten wir die Implementierung von *Transitionen*. Innerhalb einer Moduldefinition beschreiben Transitionen die Regeln, nach denen Modulinstanzen ihren Zustand[19] modifizieren können.

18. Die Namenspräfixe dienen in erster Linie der leichteren Lesbarkeit des Codes.

19. genauer: den Zustand des Gesamtsystems (z. B. über das Versenden von Interaktionen nach außen)

Transitionen bestehen im Wesentlichen aus einer Menge von jeweils optionalen *Transitionsklauseln*, die *Schaltbedingungen* (z. B. bei der `PROVIDED`-Klausel), *Schaltwirkungen* (z. B. bei der `TO`-Klausel) oder beides (z. B. bei der `WHEN`-Klausel[20]) beschreiben, und einem *Transitionsblock*, der ausschließlich Schaltwirkungen beschreibt. Entsprechend besitzt eine Transition auch keinen eigenen (dauerhaften) Zustandsraum, sondern ist aus objektorientierter Sicht eigentlich eine *Methode* (bzw. mehrere Methoden) ihrer Modulinstanz, die nach bestimmten Regeln (der Estelle-*Semantik*) aufgerufen werden.

Im Kodierungsschema von XEC werden Transitionen jedoch als *eigenständige Klassen* (innerhalb ihrer jeweiligen Modulklasse) repräsentiert und zusammen mit jeder Modulinstanz getrennt instanziiert. Die Transitionsklassen sind dabei Spezialisierungen einer polymorphen Transitions-Basisklasse aus der Laufzeitbibliothek.

Diese Abbildung erfüllt auf sehr elegante Weise verschiedene Anforderungen an das Kodierungsschema. So werden z. B. lediglich die in der Spezifikation auftretenden Transitionsklauseln durch Überschreiben der entsprechenden virtuellen Memberfunktionen in den Transitionsklassen repräsentiert (siehe „**`class`** `TRANS_test`“ in Abb. 3-9), während die nicht explizit in der Spezifikation angegebenen Transitionsklauseln auch in der Implementierung nicht explizit kodiert werden, sondern implizit auf die von der Transitions-Basisklasse geerbten Default-Methoden für die einzelnen Klauseln zurückfallen, welche wiederum die Default-Semantik der einzelnen Klauseln realisieren (siehe „**`class`** `Simple_Trans`“ in Abb. 3-9). Dies erfüllt augenscheinlich das Grundprinzip der 1:1-Kodierung der Estelle-Spezifikation auf die C++-Implementierung.

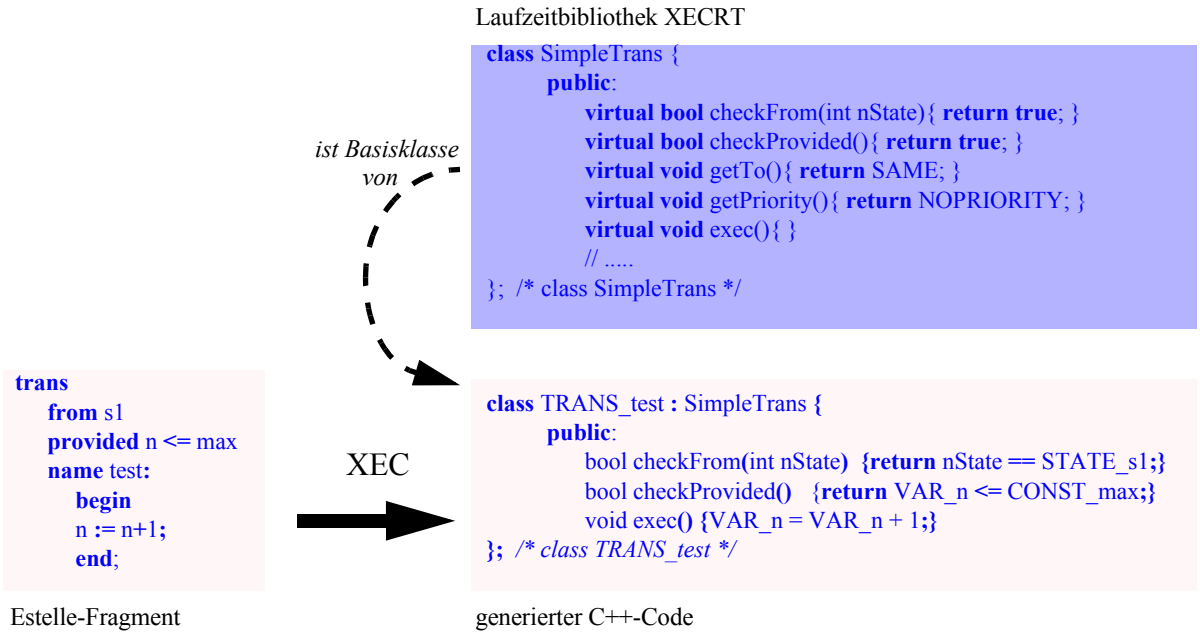


Abbildung 3-9: Prinzip der Modellierung von Estelle-Transitionen durch XEC

Anhand der Implementierung von Estelle-Transitionen und ihrer Transitionsklauseln lässt sich auch sehr gut die geforderte Trennung der Implementierung *syntaktischer Aspekte* (im generierten Code) und *semantischer Aspekte* (in der Laufzeitbibliothek) illustrieren: Für eine Transition

20. Die `WHEN`-Klausel beschreibt sowohl eine *Schaltbedingung* (das Bereitstehen einer bestimmten Interaktion an einem Interaktionspunkt), als auch eine *Schaltwirkung* (das Entfernen der Interaktion aus der zugehörigen Warteschlange während der Transitionsausführung).

werden ausschließlich Methoden zur Implementierung *einzelner Klauseln* bzw. des Transitionsblocks generiert. Wann und unter welchen Umständen diese Methoden aufgerufen werden und damit Bezug auf diese Klauseln genommen wird, liegt dagegen ausschließlich in der Hand des Laufzeitssystems. So könnte z. B. eine zur Optimierung der lokalen Transitionsauswahl eingeführte tabellengesteuerte Auswertung der `FROM`-Klauseln (siehe Abschnitt 4.3) dazu führen, dass die entsprechende Methode der betroffenen Transitions-Objekte nach einer Konfigurationsphase (zum Aufbau der Tabelle) überhaupt nicht mehr aufgerufen wird. In jedem Falle steht dies unter ausschließlicher Kontrolle des Laufzeitsystems.

Umgekehrt *abstrahiert* die vorgestellte Implementierung einer Transition und ihrer Klauseln *aus Sicht des Laufzeitsystems* auch von der spezifischen Behandlung von möglicherweise nicht explizit angegebenen Transitionsklauseln, indem ggf. durch das Beerben der Transitions-Basisklasse die nicht explizit kodierten Klausel-Methoden mit Default-Verhalten belegt werden. Dabei ermöglicht die Implementierung von Transitionen als (u. U. aktive) Objekte aber auch eine sehr *differenzierte Handhabung* der einzelnen Transitionen durch das Laufzeitsystem. So kann z. B. das Laufzeitsystem so gestaltet werden, dass ausgehend von der Transitions-Basisklasse[21] die einzelnen Transitionen dynamisch oder statisch ermittelte Informationen (z. B. statistische Informationen über die Schaltbarkeit einer Transition oder Daten aus statischen Analysen und früheren Profiling-Läufen) gewinnen bzw. bereithalten und diese zur Optimierung eingesetzt werden können.

Wir werden uns später insbesondere im Zusammenhang mit der lokalen Transitionsauswahl nochmals genauer mit diesem Punkt beschäftigen (siehe Abschnitt 4.3).

3.2.2.3 Grundzüge der Implementierung von Statements und Statement-Blocks

Als drittes und letztes Beispiel des Kodierungsschemas von XEC betrachten wir die Implementierung von Estelle- (bzw. Pascal-) Statements, wie sie z. B. im Block von Transitionen oder Funktionen bzw. Prozeduren auftreten. Solche Statement-Blocks beinhalten strukturierte oder unstrukturierte Folgen von Instruktionen, die sich in besonders hohem Maße direkt 1:1 nach C++ übertragen lassen, indem jedes einzelne Statement bzw. jede Struktur in ihr C++-Äquivalent abgebildet wird.

Im Gegensatz zu den vorgenannten Aspekten der *Modulstruktur* und *Transitionsauswahl* bzw. *-Ausführung* beinhaltet die Implementierung von *Statement-Blocks* nur in geringem Umfang spezifische Aspekte der formalen Beschreibungstechnik Estelle und ihres semantischen Modells *kommunizierender endlicher Automaten*. Stattdessen basieren die Statement-Blocks syntaktisch wie auch semantisch (bis auf wenige Ausnahmen[22]) auf dem Estelle zu Grunde liegenden Standard-Pascal und seinem *imperativ-sequentiellen* Ausführungsmodell. Entsprechend fällt auch die Kontrolle der Ausführung von abgeschlossenen Estelle-Statementblocks verhältnismäßig einfach aus und erfordert keine komplexe aktive Kontrolle durch das Laufzeitsystem, wie sie z. B. bei der Transitionsauswahl erforderlich ist.

Daher erfolgt die Implementierung von Statement-Blocks durch strikte Umsetzung jedes einzelnen (strukturierten oder unstrukturierten) Estelle-Statements in ein äquivalentes C++-Statement. Im Falle von Statements, die nur durch komplexere Anweisungen oder Anweisungsfol-

21. also ausschließlich innerhalb der Laufzeitbibliothek, d.h. ohne Anpassung des generierten Codes

22. Es gibt auch Estelle-spezifische Statements wie z. B. `ANY`-, `ALL`- oder `OUTPUT`-Statements.

gen in C++ abgebildet werden könnten, werden ggf. Prozeduren oder Makros aus der Laufzeitbibliothek benutzt, um eine kompakte und leicht verständliche Implementierung für die einzelnen Anweisungen generieren zu können.

Dies gilt insbesondere auch für die Implementierung von *Ausdrücken* (*Expressions*), wie sie u. a. innerhalb von Statement-Blocks z. B. als Parameter von Funktionsaufrufen oder Zuweisungen auftreten. Auch hier wird im Sinne der 1:1-Kodierung ein kompakter äquivalenter Ausdruck in C++ erzeugt.[23]

Die Grundidee hinter diesem Implementierungskonzept besteht in der Annahme, dass in den Statement-Blocks einer Spezifikation (im Rahmen der *Ausdrucksfähigkeit* von Estelle)[24] nur solche Anweisungssequenzen spezifiziert sind, die bei einer manuellen Implementierung der Spezifikation in gleicher Art und Weise zur Realisierung der jeweiligen Funktionalität eingesetzt würden. Entsprechend würde ein Programmierer zur Erstellung einer manuellen Implementierung letztlich weitgehend die gleichen C++-Anweisungssequenzen einsetzen, die auch XEC aus der Estelle-Spezifikation generiert hat.

Genau auf diesem Aspekt setzt der hier realisierte Ansatz einer effizienten Implementierung auf, um über den generierten C++-Code letztlich eine effiziente ausführbare Implementierung zu erhalten: Moderne C++-Compiler implementieren eine Vielzahl von Optimierungstechniken und -heuristiken, die speziell auf die von Programmierern typischerweise angewandten Kodierungstechniken, Strukturen und Idiome ausgerichtet sind. Unter der Annahme, dass der Spezifizierer unter Estelle die zur Realisierung der gestellten Aufgabe notwendigen Operationen kompakt und effizient spezifiziert, ist die naheliegendste Methode zur Generierung effizienter Implementierungen aus solchen Spezifikationen, diese *Kompaktheit und Effizienz so weit wie möglich bei der Umsetzung nach C++ zu erhalten* und die Anwendbarkeit von Compiler-Optimierungen dadurch zu unterstützen, dass möglichst solche Code-Sequenzen erzeugt werden, wie sie auch von einem Programmierer bei der manuellen Implementierung eingesetzt worden wären.

Im Gegensatz dazu würde eine Abbildung des Estelle-Codes in eine zu komplexe C++-Codestruktur zu unnötig ineffizienten Implementierungen führen. Dies gilt besonders dann, wenn aufgrund eines (im Vergleich zu einer manuellen Implementierung) „*unnatürlichen*“ Kodierungsschemas die Optimierungsheuristiken der C++-Compiler nicht greifen.[25]

---

23. Eine Sonderbehandlung erfordert hierbei das `exist`-Statement (Abschnitt 7.6.11 von [ISO97]), das nur auf Basis der gcc-Erweiterung „*Statements and Declarations in Expressions*“ vollständig innerhalb einer C++-Expression implementiert werden kann. Standard-C++-konforme Lösungen erfordern dagegen weitaus komplexere Konstrukte, die i. A. auf der Umsetzung in einer zusätzlichen Funktionsdefinition oder der vollständigen Dekomposition des Ausdrucks beruhen.

24. Wir werden später an einigen Stellen sehen, dass die Beschränkungen der Ausdrucksfähigkeit von Estelle auf Statement-Ebene gelegentlich auch zu – im Vergleich zu Handimplementierungen – komplizierteren und ineffizienteren Lösungen führen (siehe u. a. Kapitel 6).

25. Dies unterscheidet XEC z. B. vom Codegenerator EC, der zur Implementierung von Statement-Blöcken häufig heuristisch explizit „`register`“-attributierte Hilfsvariablen zur Ablage des Ergebnisses von Teilausdrücken bei Berechnungen verwendet, ohne dass dies semantisch erforderlich wäre. Der Effekt solcher Maßnahmen bezüglich der Performance der Implementierung ist jedoch hochgradig compiler- und maschinenabhängig und kann auch zu einer Verringerung der Effizienz führen, wenn z. B. dadurch die Register-Allokation des Compilers gestört wird oder die Code-Komplexität so weit steigt, dass Optimierungsheuristiken des C++-Compilers nicht mehr greifen.

Wir gehen also davon aus, dass die Optimierungs-Heuristiken des nachgeschalteten C++-Compilers um so besser greifen, je näher der aus der Estelle-Spezifikation automatisch generierte Code einer effizienten und kompakten *manuell* erstellten Implementierung kommt.

Entsprechend zielt die Codegenerierung durch XEC auf folgende *Leitlinien* ab:

- Der generierte C++-Code bildet den zu Grunde liegenden Estelle-Statementblock so eng wie möglich ab. Insbesondere sollte der generierte Code nicht komplexer sein, als der Ausgangs-Statementblock.
- Der generierte C++-Code sollte so weit wie möglich einer manuell erstellten Implementierung entsprechen, also möglichst wenige „*künstliche*“ Programmstrukturen einführen. Entsprechend sollte er für einen Programmierer leicht lesbar sein.
- Soweit Methoden, Funktionen oder Makros aus der *Laufzeitbibliothek* eingesetzt werden, sollten diese eine hohe *Grundeffizienz* bei der Lösung der gestellten Aufgabe besitzen.

Wir wollen diese Ideen an einem einfachen Beispiel illustrieren. Die Funktion `f()` im folgenden Estelle-Spezifikationsfragment berechnet offensichtlich auf nahe liegende Weise die Summe der Ganzzahlen von `1` bis zum übergebenen Parameter `n` und liefert diese Summe als Funktionsergebnis zurück.

Beispiel 3.4-a: Estelle-Spezifikationsfragment mit Funktionsdefinition

```
SPECIFICATION Test;

    FUNCTION f(n: INTEGER): INTEGER;
        VAR sum, i: INTEGER;
        BEGIN
        sum := 0;
        FOR i := 1 TO n DO
                BEGIN
                sum := sum + i;
                END;
        f := sum;
        END;

(* ... *)
END.
```

(Ende von Beispiel 3.4-a)

In C bzw. C++ würde man diese Funktion ganz analog folgendermaßen implementieren:

Beispiel 3.4-b: Manuelle Implementierung der Funktion aus Beispiel 3.4-a

```
int f(int n) {
        int i, sum = 0;

        for (i = 1 ; i <= n ; i++)
                sum = sum + i;
        return sum;
}
```

(Ende von Beispiel 3.4-b)

Betrachten wir nun die von XEC erzeugte Implementierung[26] des Blocks der oben genannten Estelle-Funktion:

Beispiel 3.4-c: Von XEC generierte C++-Implementierung der Funktion aus Beispiel 3.4-a

```
TYPE_integer SPEC_test::FUNC_f::call (
        TYPE_integer PAR_n
) {
        TYPE_integer VAR_sum;
        TYPE_integer VAR_i;
        TYPE_integer RetVal;
        VAR_sum = 0;
        FOR_BEGIN_UP(VAR_i, 1, PAR_n)
                VAR_sum = (VAR_sum + VAR_i);
        FOR_END()
        RetVal = VAR_sum;
        return RetVal;
}
```

(Ende von Beispiel 3.4-c)

Offensichtlich bildet diese Methode den Statement-Block der oben genannten Estelle-Funktion (insbesondere unter Erhaltung der Namen[27] und der Struktur) nahezu identisch nach.[28] So kann man leicht für jedes Statement und jede Definition der Estelle-Funktion ein Äquivalent in der C++-Methode finden und umgekehrt. Insbesondere entspricht sie fast vollständig der manuellen Implementierung aus Beispiel 3.4-b.

Lediglich an einer Stelle, nämlich der Zwischenspeicherung des Rückgabewertes der Funktion wird in der automatisch generierten C++-Implementierung explizit eine Hilfsvariable (`RetVal`) angelegt, auf die die Zuweisungen auf die implizite Rückgabe-Variable (`f`) der Estelle-Funktion abgebildet werden. Diese Art der Kodierung der Wert-Rückgabe ist in komplexeren Situationen erforderlich, da in Estelle (wie auch schon in Pascal) mehrere Zuweisungen auf die Rückgabe-Variable möglich sind, ohne einen Rücksprung aus der Funktion auszulösen[29], wohingegen in C++ (wie auch schon in C) ein Rückgabewert nur bei der Ausführung der `return`-Anweisung (also beim Verlassen der Funktion) übergeben werden kann. Die explizite Kodierung der Hilfsvariablen `RetVal` dient also dazu, unabhängig von der konkreten Konstellation in der Estelle-Spezifikation durch 1:1-Kodierung zu einer konformen Implementierung in C++ zukommen.

An dieser Stelle wird jedoch auch die strikte Einhaltung der 1:1-Kodierung von Statements-Blocks nochmals deutlich: Im obigen Beispiel könnte auf Grund der Tatsache, dass die Zuweisung auf die Rückgabe-Variable (`RetVal`) ausschließlich als letzte Anweisung in dem Statement-Block der Funktion auftritt, eigentlich statt der letzten beiden C++-Statements

26. Im Gegensatz zu den vorangegangenen Beispielen handelt es sich hier nicht um eine Vereinfachung, sondern tatsächlich um den Original-Quelltext, wie er von XEC erzeugt wurde.

27. Die Namens-Präfixe (wie „`TYPE_`“ oder „`PAR_`“) dienen in erster Linie der leichteren Lesbarkeit des generierten Codes (siehe Abschnitt 3.3.2).

28. Der Typ „`TYPE_integer`“ implementiert den vordefinierten Estelle-Typ „`INTEGER`“ und wird in der Laufzeitbibliothek (`xecrt_estelle.h`) normalerweise als der C/C++-Typ „`int`“ definiert. Alternative Implementierungen des Estelle-Typs „`INTEGER`“ werden in Abschnitt 3.3.3 diskutiert.

29. Der Rückgabewert der Funktion ist dann der letzte im Verlauf der Funktions-Ausführung zugewiesene Wert.

(„`RetVal := VAR_sum;  return RetVal;`“) auch vereinfachend das Statement „`return VAR_sum;`“ eingesetzt und auf die Einführung der Hilfsvariablen `RetVal` vollständig verzichtet werden. Derartige Optimierungen würden jedoch den Codegenerierungsvorgang erheblich verkomplizieren, ohne im Endeffekt einen Performance-Vorteil zu bieten: Moderne C++-Compiler sind auf Grund ihrer Datenfluss-Analyse sehr wohl in der Lage, eigenständige Optimierungen dieser Art auch nachträglich vorzunehmen, indem sie z. B. von vorne herein die Variablen `RetVal` und `VAR_sum` unifizieren und diese möglicherweise sogar bereits in einem für die Wertrückgabe der Funktion vorgesehenen Register allokieren.[30] Entsprechend hat die strikte Einhaltung des Kodierungsschemas an dieser Stelle keine negativen Auswirkungen auf die Performance der letztlich generierten Implementierung; sie erlaubt es jedoch, den Codegenerator unabhängig von solchen Fragestellungen einfach zu halten.

Ein weiterer charakteristischer Aspekt ist die Implementierung der `FOR`-Schleife durch die beiden Makros `FOR_BEGIN_UP` und `FOR_END`. Diese erlauben es einerseits, den generierten Code (und den Codegenerator) einfach und leicht verständlich zu halten, gleichzeitig jedoch für den C++-Compiler effizienten und kompakten Code zu erzeugen. Die in Beispiel 3.4-d angegebene Definition dieser beiden Makros aus der Laufzeitbibliothek (`xecrt_estelle.h`) zeigt, dass sie im Wesentlichen eine C++-`for`-Schleife realisieren, wie sie bei einer manuellen Implementierung an dieser Stelle ebenfalls eingesetzt worden wäre.

Beispiel 3.4-d: Definition der `FOR`-Schleifen-Makros in der Laufzeitbibliothek

```
#define FOR_BEGIN_UP(NAME, LO, HI)  {int nLimit = (HI);       \
                                     for (  NAME = (LO) ;     \
                                            NAME <= nLimit ; \
                                            NAME = (NAME)+1   \
                                     ) {

#define FOR_END()                   }}
```

(Ende von Beispiel 3.4-d)

Eine Besonderheit ergibt sich jedoch hier aus der Estelle- (bzw. Pascal-) Semantik der `FOR`-Schleife, die eine finale Auswertung und Fixierung des Endwertes der `FOR`-Schleife vor dem ersten Durchlauf verlangt.[31] Entsprechend wird in der Definition des Makros `FOR_BEGIN_UP` eine Hilfsvariable `nLimit` angelegt, mit der das Einfrieren des Endwertes (durch Auswertung des dritten Makro-Parameters) für die Schleifenvariable implementiert wird.[32] Zur Erhaltung des 1:1-Kodierungsschemas wird jedoch durch die Anwendung des Makros von diesem Imple-

30. Diese Optimierung wird z. B. von gcc (2.9.5) für SPARC- und IA32-Plattformen bei der Optimierungsstufe „`-O2`“ automatisch realisiert.

31. Der Unterschied wäre z. B. wahrnehmbar, wenn im Schleifenrumpf die Parameter-Variable `n` jeweils auf `i+1` gesetzt würde: Gemäß der Estelle-Semantik hätte dies keinen direkten Einfluss auf die Anzahl der Schleifendurchläufe. Würde der Wert von `n` jedoch bei jedem Schleifendurchlauf von neuem überprüft, so würde die Schleife nicht terminieren (sofern überhaupt ein Schleifendurchlauf stattfindet).

32. Da die Variable `n` im obigen Beispiel tatsächlich während des Schleifendurchlaufs gar nicht modifiziert wird, wäre die Einführung dieser Hilfsvariable in diesem Falle eigentlich nicht notwendig. Sie hat jedoch auf die Effizienz der letztlich generierten Implementierung keinen negativen Einfluss, da auch hier die oben bereits beschriebenen Überlegungen zur Optimierung durch den C++-Compiler analog gelten.

mentierungsdetail auf Codegenerierungsebene abstrahiert. Insbesondere wären mit Hilfe der beiden Makros alternative Implementierungen der FOR-Schleife durch eine einfache Modifikation der Laufzeitbibliothek und ohne Anpassungen des Codegenerators möglich.

Damit beenden wir die Einführung in die Grundzüge des Kodierungsschemas von XEC.

3.3. Statische Implementierungsstrukturen

Wir werden nun die wesentlichen Strukturen und Mechanismen zur Implementierung von Estelle-Spezifikationen durch XEC anhand einiger Original Code-Fragmente untersuchen. Wir konzentrieren uns dabei auf die Aspekte der statischen Modulhierarchie (Abschnitt 3.3.2), Datentypen und Variablen (Abschnitt 3.3.3), Funktionen und Prozeduren (Abschnitt 3.3.6), Interaktionspunkte und Nachrichtenkommunikation (Abschnitt 3.3.4) und schließlich Transitionen und Transitionsklauseln (Abschnitt 3.3.7).

Wir beginnen jedoch mit einer kurzen Einführung die die Klasse `XList` als tragende Datenstruktur der gesamten Implementierung des XEC-Toolkits.

3.3.1 Die XList-Datenstrukturen

Als Grundlage für fast alle dynamischen Datenstrukturen des von XEC generierten Codes und der Laufzeitbibliothek XECRT (wie auch des Codegenerators XEC selbst) wurde die in der Laufzeitbibliothek „`baselib.h/cc`“ implementierte Listenverwaltung `XList` entwickelt.

Sie bietet eine auf rekursiven C++-Template-Definitionen basierende *starke Aggregation* von Elementen doppelt verketteter Listen, die sich durch hohe Laufzeiteffizienz im modifizierenden und nicht-modifizierenden Zugriff, eine durchgängige strenge Typprüfung und ein sehr hohes Abstraktionsniveau in der Handhabung auszeichnet.

Zur Definition einer C++-Klasse, deren Elemente in eine solche XList-Instanz aufgenommen werden sollen, dient eine rekursive Basisklassendefinition, wie sie anhand der Beispielklasse Payload im folgenden Beispiel demonstriert wird:

Beispiel 3.5-a: Anwendung von XList: Nutzlast-Klassendefinition

```
class Payload : public XListNode<Payload> {
        // ...
};
```

(Ende von Beispiel 3.5-a)

Die Nutzlast-Klasse `Payload` hat als Basisklasse die Klasse `XListNode`, die umgekehrt wiederum als Typ-Parameter die (gerade in Definition befindliche) Klasse `Payload` erhält. Offensichtlich handelt es sich um eine rekursive Klassendefinition, die jedoch gemäß des C++-Standards [ISO98] auf Grund der Art der Verwendung des Parameter-Typs in `XListNode` zulässig ist.

Um eine Liste zur Aufnahme solcher `Payload`-Objekte anzulegen, genügt es, eine Instanz der Klasse `XList<Payload>` anzulegen:

Beispiel 3.5-b: Anwendung von XList: Instanziierung einer Liste

```
XList<Payload> myList;
```

(Ende von Beispiel 3.5-b)

Eine genauere Diskussion der Grundlagen, Operationen und Eigenschaften der `XList` und der darauf aufbauend definierten `XPtrList` bzw. `XPtrListNode` (für schwache Aggregationen und Mehrfachlistenmitgliedschaften) ist in Anhang A.5 zu finden.

3.3.2 Modulhierarchie

Wie wir bereits in Abschnitt 3.2.2.1 gesehen haben, implementiert XEC die statische wie auch die dynamische Modulstruktur einer Estelle-Spezifikation in jeweils analogen C++-Klassen- bzw. Objekthierarchien.

Die *statische Modulstruktur*, also das System von streng hierarchisch ineinander verschachtelten Modulrumpf-Definitionen, geht in Estelle immer vom Spezifikations-Modul als Wurzel des Syntax-Baumes aus. Dieses Spezifikations-Modul kann – wie jeder Estelle-Modulrumpf – in weitere Estelle-Module unterstrukturiert werden. Dabei bildet jedes Estelle-Modul einen eigenen Namensraum, sodass lokale Definitionen verschiedener Module unabhängig von möglichen Namensgleichheiten voneinander unterschieden werden können.[33]

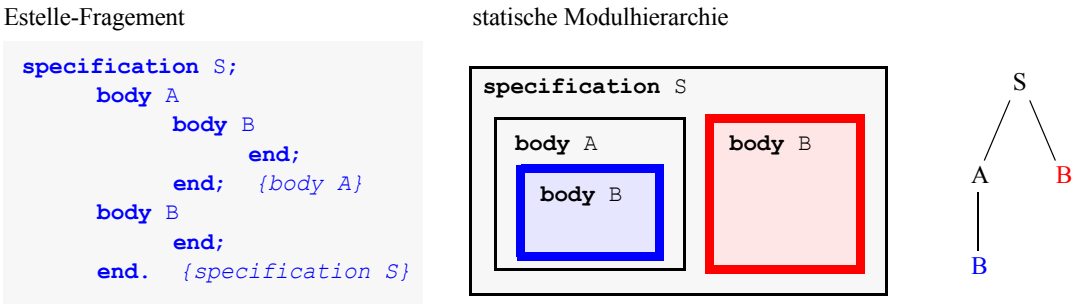


Abbildung 3-10: Statische Modulhierarchie in Estelle

Betrachten wir dazu das in Abb. 3-10 links dargestellte Estelle-Fragment einer Modulhierarchie: Wir sehen hier (einschließlich des Spezifikations-Moduls selbst) vier zum Teil ineinander geschachtelte Estelle-Module. Zwei dieser Module besitzen dabei den selben Namen „`B`“. Diese beiden gleichnamigen Module werden in Estelle ausschließlich anhand ihrer unterschiedlichen Definitions-Kontexte (das Modul „`A`“ beziehungsweise das Spezifikations-Modul „`S`“) und der damit verbundenen getrennten Namensräume voneinander unterschieden.[34]

Bis auf XEC generieren alle bekannten Estelle-Codegeneratoren (insbesondere EC und DINGO) aus spezifischen internen Regeln und der Historie des Codegenerierungsvorgangs hervorgehende *Zahlen-Erweiterungen* (wie z. B. „`b_21`“ oder „`b_17`“) zur Aufzählung und Unterscheidung der verschiedenen, aber ursprünglich namensgleichen Definitionen. Diese bei der Codegenerierung eingeführten künstlichen Namenserweiterungen erschweren die Verfolgbarkeit des Implementierungsvorgangs und verkomplizieren durch die Notwendigkeit der Assoziation von Estelle-Objekten mit den einmal zugeordneten Zahlen-Erweiterungen das Kodie-

33. Es ist jedoch zu beachten, dass im Gegensatz zum später vorgestellten C++-Namensschema in Estelle keine qualifizierten Namen vorkommen, sondern die Sichtbarkeit von Namen und Definitionen allein über die syntaktischen Regeln von Estelle festgelegt sind.

34. Standard-Estelle kennt keine *qualifizierten* Namen, die Namensauflösung erfolgt ausschließlich über implizite Sichtbarkeitsregeln zwischen den Namensräumen. Wir führen später im Zusammenhang mit der Estelle-Erweiterung *Open-Estelle* jedoch qualifizierte Namen ein (siehe auch Kapitel 7).

rungsschema und den Codegenerator. Besonders schwerwiegend ist jedoch, dass sich diese Zahlenerweiterungen aus internen Zuständen des Codegenerators ergeben und daher nur im Kontext einer konkreten Codegenerierung determiniert sind. Dies macht insbesondere die getrennte Codegenerierung und Übersetzungen von nachträglich zu einem System zusammengefügten Systemkomponenten nahezu unmöglich. Wir werden uns mit diesem Punkt später im Zusammenhang mit *Open-Estelle* noch genauer beschäftigen (siehe Kapitel 7).

Um ein Höchstmaß an Verfolgbarkeit und Determiniertheit des Implementierungsvorgangs zu erreichen, bildet das Kodierungsschema diese Estelle-Struktur auf eine analoge C++-Struktur ab, wobei (bis auf feste Typ-Präfixe[35] wie „SPEC_“, „BODY_“ oder „HEADER_“, siehe Tabelle A.35 auf Seite 406) so weit wie irgend möglich auf die Erzeugung *künstlicher* (und insbesondere *künstlich unterschiedlicher*) Namen verzichtet wird. Dazu wird jedes Modul auf eine C++-Klassendefinition abgebildet, wobei diese Klassen genau wie die Ausgangs-Estelle-Module *ineinander geschachtelt* sind (siehe auch Abb. 3-8 auf Seite 37). Dadurch ergeben sich *qualifizierte* C++-Namensräume und -Namen, die künstliche Namenserweiterungen prinzipiell unnötig machen. Stattdessen sind die qualifizierten Namen Ergebnisse der C++-Klassenstruktur und werden damit durch die syntaktische Struktur der Ausgangs-Estelle-Spezifikation unmittelbar determiniert, die ja Vorlage für die generierte C++-Struktur war.

Wir illustrieren diesen Implementierungsvorgang anhand des folgenden Beispiels. Darin ist eine zum in Abb. 3-10 angegebenen Estelle-Fragment vollständige Estelle-Spezifikation wiedergegeben, die neben den hierarchisch strukturierten Modul-Rumpf-Definitionen auch die notwendigen Modulheader-Definitionen („mod_a“ in Zeile 3 und zwei verschiedene „mod_b“ in Zeile 9 und 18) enthält.

Beispiel 3.6-a: Beispiel-Estelle-Spezifikation einer Modulhierarchie

```
 1: SPECIFICATION s;
 2:
 3:         MODULE mod_a;
 4:                 (* interface *)
 5:         END;
 6:
 7:         BODY a FOR mod_a;
 8:                 (* internal definitions *)
 9:                 MODULE mod_b;
10:                         (* interface *)
11:                 END;
12:
13:                 BODY b FOR mod_b;
14:                         (* internal definitions *)
15:                 END;
16:         END;
17:
18:         MODULE mod_b;
19:                 (* interface *)
20:         END;
21:
22:         BODY b FOR mod_b;
23:                 (* internal definitions *)
```

35. Die erzeugten Präfixe identifizieren lediglich die *Kategorie* des referenzierten Objekts und dienen in erster Linie der leichteren Lesbarkeit des generierten Codes.

```
24:         END;
25:
26: END.
```

(Ende von Beispiel 3.6-a)

Aus dieser Estelle-Spezifikation generiert XEC schließlich eine `cc`- und eine `h`-Datei, die in Beispiel 3.6-b und Beispiel 3.6-c ausschnittsweise wiedergegeben sind (siehe auch Abschnitt 3.1.2.2).[36] Die `h`-Datei beinhaltet dabei die tragenden C++-Klassendefinitionen,[37] während die `cc`-Datei im Wesentlichen nur einige dort gemachte (u.a. Methoden-) Deklarationen vervollständigt. Insbesondere enthält die `cc`-Datei normalerweise alle Implementierungen von Statement-Blocks der Ausgangsspezifikation (treten in diesem Beispiel nicht auf).

Beispiel 3.6-b: Von XEC aus Beispiel 3.6-a generierte `.h`-Datei (Auszug)

Von XEC generierter Code, aus dem lediglich einige Definitionen entfernt wurden:

```
 1: #include "xecrt.h"
 2:
 3: class SPEC_test : public Specification {
 4:     public:
 5:         class HEADER_mod_a : public Module {
 6:             public:
 7:                 inline HEADER_mod_a(Module* pParent_,
                                        unsigned uHints = 0);
 8:         };
 9:         class HEADER_mod_b : public Module {
10:             public:
11:                 inline HEADER_mod_b(Module* pParent_,
                                        unsigned uHints = 0);
12:         };
13:         SPEC_test();
14:         class BODY_a;
15:         class BODY_b;
16: }; /* class SPEC_test */
17:
18: class SPEC_test::BODY_a : public SPEC_test::HEADER_mod_a {
19:     public:
20:         class HEADER_mod_b : public Module {
21:             public:
22:                 inline HEADER_mod_b(Module* pParent_,
                                        unsigned uHints = 0);
23:         };
24:         BODY_a(Module* pParent);
25:         class BODY_b;
26: }; /* class BODY_a */
27:
28: class SPEC_test::BODY_a::BODY_b :
```

36. Der wiedergegebene Code entspricht weitestgehend den generierten Dateien, es wurden lediglich zur Vereinfachung einige an dieser Stelle unerhebliche Definitionen entfernt.

37. Die Bezeichnung als „C++-Klassen*definition*“ ist im Vergleich unseres Sprachgebrauchs bzgl. Definitionen und Deklarationen in Estelle ein Grenzfall, da zwar der Zustandsraum festgelegt wird, nicht jedoch alle Methoden.

```
                                public SPEC_test::BODY_a::HEADER_mod_b {
29:     public:
30:         BODY_b(Module* pParent);
31: }; /* class BODY_b */
32:
33: class SPEC_test::BODY_b : public SPEC_test::HEADER_mod_b {
34:     public:
35:         BODY_b(Module* pParent);
36: }; /* class BODY_b */
```

(Ende von Beispiel 3.6-b)

Die C++-Klassendefinitionen bilden dabei die bereits in Abb. 3-8 auf Seite 37 angegebene Hierarchie der Ausgangsspezifikation durch ein System ineinander verschachtelter Klassendefinitionen nach. Im Gegensatz zur Funktionsprinzip-Darstellung in Abb. 3-8 sind im von XEC tatsächlich erzeugten Code jedoch die Klassendefinitionen *nicht wirklich textuell ineinander geschachtelt*, sondern sie werden in der `.h`-Datei auf Modulebene sequentialisiert kodiert. Dazu enthält jede Klasse, die eine eingeschachtelte Modul-Klassendefinition enthalten soll, eine entsprechende *Forward-Definition* wie z. B. „`class BODY_a;`“ (Zeile 19 von Beispiel 3.6-b) innerhalb der Klassendefinitionen von „`SPEC_s`“ (Zeilen 8 bis 21). Die Klassendefinition der eingeschachtelten Klasse „`BODY_a`“ selbst wird dagegen textuell vollständig außerhalb der ihr logisch übergeordneten Klassendefinition angegeben (Zeilen 28 bis 31). Zur eindeutigen Zuordnung wird dabei der Name der untergeordneten Klasse immer vollständig qualifiziert angegeben („`class SPEC_s::BODY_a`“ in Zeile 28).

Diese *sequentialisierte Darstellung* der logisch ineinander verschachtelten C++-Klassen ist dabei äquivalent zu einer *textuell eingebetteten* Darstellung [ISO98]. Sie ermöglicht es jedoch, die syntaktische Darstellung der generierten C++-Klassenstruktur weitaus übersichtlicher und kompakter zu gestalten, als dies bei einer eingebetteten Darstellung der Fall wäre. Insbesondere wird dadurch auch die ansonsten zur strukturierten Darstellung der verschachtelten C++-Klassen notwendige akkumulierende Einrückung des generierten Codes vermieden.

Diese Möglichkeit von C++, logisch ineinander geschachtelte Klassendefinitionen syntaktisch sequentiell darzustellen vermeidet damit die bei großen und tief verschachtelten Estelle-Spezifikationen besonders hinderliche Unübersichtlichkeit der Spezifikation und ihrer Struktur, bei der immer wieder Definitionen innerhalb des selben Moduls durch eingeschobene Kindmodul-Definitionen zum Teil über tausende von Zeilen voneinander getrennt werden (siehe auch Anhang A.1.2). Wir werden im Zusammenhang mit der Estelle-Erweiterung *Open-Estelle* später in Kapitel 7 noch einmal darauf zurückkommen.

Die `.c`-Datei enthält zu diesen Klassendefinition jeweils Methoden- und Objekt-Definitionen, die ebenfalls sequentialisiert und mit voll qualifizierten Namen dargestellt sind:

Beispiel 3.6-c: Von XEC aus Beispiel 3.6-a generierte `.cc`-Datei (Auszug)

Das Beispiel zeigt von XEC generierten Code, aus dem einige Definitionen und Methoden-Implementierungen entfernt wurden (meistens markiert durch „`// ....`“).

```
1: #include "s.h"
2:
3: /* ======================================================
4:         SPEC_s
5:    ====================================================== */
6:
```

```
 7: inline SPEC_s::HEADER_mod_a::HEADER_mod_a(
                              Module* pParent_, unsigned uHints_) :
                              Module(pParent_, uHints | uHints_) {
 8: }
 9: inline SPEC_s::HEADER_mod_b::HEADER_mod_b(
                              Module* pParent_, unsigned uHints_) :
                              Module(pParent_, uHints | uHints_) {
10: }
11:
12: SPEC_s::SPEC_s() : Specification(uHints) {
13:         // .....
14: }
15: SPEC_s::HEADER_mod_a* SPEC_s::BODY_a_NEW(Module* pParent) {
16:         // .....
17: }
18:
19: SPEC_s::HEADER_mod_b* SPEC_s::BODY_b_NEW(Module* pParent) {
20:         // .....
21: }
22:
23:
24: /* ======================================================
25:         SPEC_s::BODY_a
26:    ====================================================== */
27:
28: inline SPEC_s::BODY_a::HEADER_mod_b::HEADER_mod_b(
                              Module* pParent_, unsigned uHints_) :
                              Module(pParent_, uHints | uHints_) {
29: }
30:
31: SPEC_s::BODY_a::BODY_a(Module* pParent_) :
                              HEADER_mod_a(pParent_, uHints) {
32:         // .....
33: }
34: SPEC_s::BODY_a::HEADER_mod_b* SPEC_s::BODY_a::BODY_b_NEW(
                              Module* pParent) {
35:         // .....
36: }
37:
38:
39: /* ====================================================
40:         SPEC_s::BODY_a::BODY_b
41:    ==================================================== */
42:
43: SPEC_s::BODY_a::BODY_b::BODY_b(Module* pParent_) :
                              HEADER_mod_b(pParent_, uHints) {
44:         // .....
45: }
46:
47: // ....
```

(Ende von Beispiel 3.6-c)

Wir betrachten nun einige Implementierungs-Aspekte der Moduldefinitionen anhand der beschriebenen Grundstrukturen etwas genauer.

3.3.2.1 Klassenhierarchie der Module und Modulheader

Die C++-Klassendefinitionen zur Nachbildung eines Estelle-Modulrumpfes beinhalten neben den Klassendefinitionen für Kindmodule auch Komponenten für alle übrigen modullokalen Definitionen. Dies sind im obigen Beispiel 3.6-b zunächst nur die Modulheader-Definitionen („`mod_`...“), die die externen Schnittstellen eines Moduls angeben.

Zu einem Modulheader können in Estelle mehrere verschiedene Module definiert werden, die entsprechend bezüglich ihrer *Modul-Parameter*, ihrer *exportierten Variablen* und ihrer *externen Interaktionspunkte* durch den Modulheader festgelegt sind. Insbesondere wird dieser Umstand bei der Referenzierung von Modulinstanzen durch *Modulvariablen* genutzt, indem der Typ der Modulvariablen allein durch einen Modulheader definiert wird. Entsprechend kann einer solchen Modulvariablen auch nur die Instanz eines zum Modulheader passenden Modul-Rumpfes zugewiesen werden. Da ein Zugriff auf Modulinstanzen von außen nur über diese Modulvariablen möglich ist, sind die äußeren Schnittstellen der Module syntaktisch (im Sinne ihres Ein-Ausgabe-Alphabetes) auf die Definitionen der Modulheader beschränkt. Die verschiedenen Modul-Rümpfe unterscheiden sich entsprechend nur noch bezüglich ihres konkreten *Verhaltens* im Rahmen dieser Ein-Ausgabe-Schnittstelle.

Aus einer objektorientierten Betrachtungsweise heraus bilden Modulheader offensichtlich eine Art *Signatur* oder Basisklasse für die zugehörigen Module, indem die syntaktischen Eigenschaften der externen Schnittstelle vom Modulheader auf die daraus abgeleiteten Module vererbt werden. Zugriffe auf Modulinstanzen über Modulvariablen erfolgen dann (aus dem generierten Code heraus) auf der Abstraktionsebene eben dieser Modulheader, die ja Definitionsgrundlage für die Modulvariablen sind.[38]

XEC implementiert gemäß dieser objektorientierten Betrachtungsweise Modulheader als Basisklassen der von ihnen abgeleiteten Module (siehe Zeilen 28, 43 und 53 in Beispiel 3.9-b). Entsprechend werden Modulvariablen von XEC als Referenzen[39] auf die entsprechende Modulheader-Klasse implementiert.

Die Modulheader wiederum haben als Basisklasse die Laufzeitbibliothek-Klasse `Module`, die somit für alle Module Methoden und Daten zur Verwaltung des Modulsystems und zur Steuerung der Spezifikations-Ausführung bereitstellt (siehe UML-Diagramm in Abb. 3-11). So enthält die Basisklasse `Module` u. a. eine Liste der Kindmodulinstanzen des Moduls (`XList<Module> Cildren`) und eine Liste seiner Transitions-Objekte (`XList<SimpleTrans> Transitions`), aber auch Methoden zur lokalen und globalen Transitionsauswahl (`selectTrans(...)`), zum Management der Verbindungsstruktur und der Weitergabe von Interaktionen, die Generierung von Namen für Debugging und Tracing, usw.

---

38. Der Zugriff auf Module aus der Laufzeitbibliothek heraus erfolgt dagegen ausschließlich auf der Abstraktionsebene der Modul-Basisklasse `Module` (s. u.).

39. Tatsächlich sind Modulvariablen Instanzen des XECRT-Klassen-Templates `Modvar<T>` mit der Modulheader-Klasse als Template-Parameter, die im Wesentlichen jedoch die Eigenschaften von C++-Objektreferenzen nachbilden (s. u.).

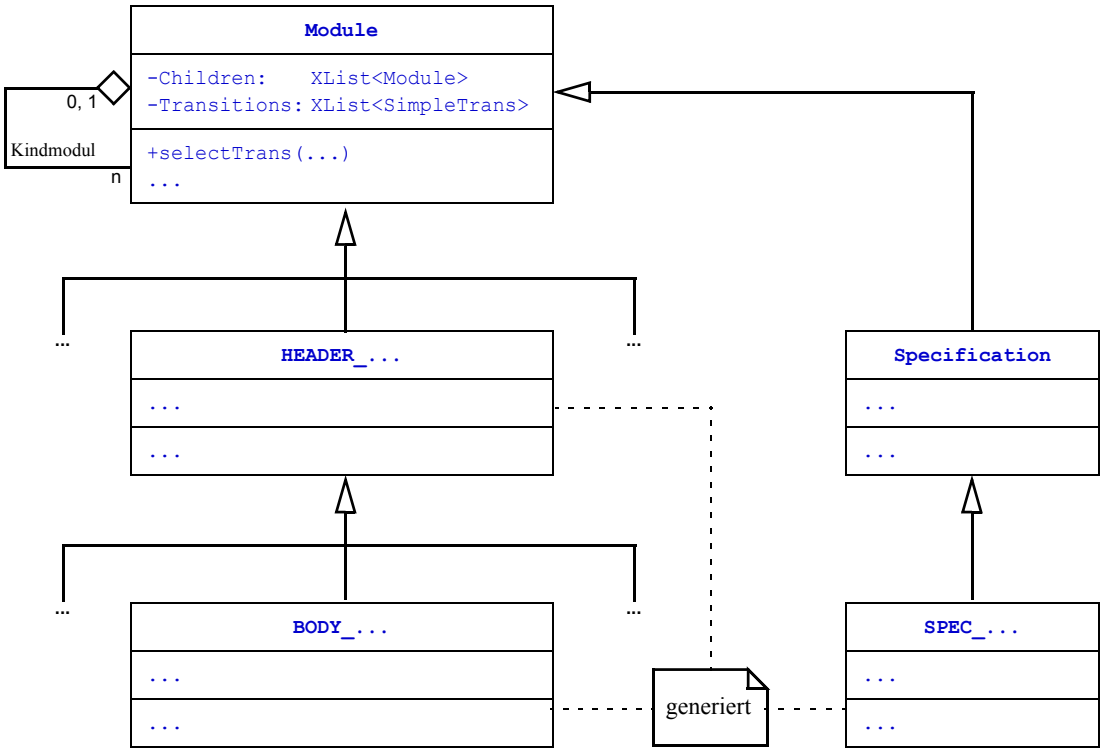


Abbildung 3-11: UML-Diagramm der Modul-Klassenzugehörigkeit

Einen Sonderfall stellt in dieser Hinsicht das Spezifikations-Modul selbst dar, da es in Standard-Estelle keinen expliziten Modulheader besitzt (siehe Zeile 1 in Beispiel 3.6-a). Aus konzeptioneller Sicht entspricht dies einem implizit angenommenen *leeren* Modulheader ohne externe Interaktionspunkte, exportierte Variablen oder Modulparameter. Dieser repräsentiert die (topologische) Geschlossenheit der Spezifikation. Implementiert wird dieser Aspekt von XEC durch die Anwendung der XECRT-Basisklasse `Specification` für das Spezifikationsmodul. Als Spezialisierung der Klasse `Module` stellt sie in gewisser Hinsicht die explizite Implementierung des (impliziten und damit festen) Headers des Spezifikations-Moduls dar (siehe Abb. 3-11). Darüber hinaus stellt sie auf Implementierungsebene einige Methoden und Datenstrukturen für das *globale Systemmanagement* der Spezifikation bereit.

Auf diese Weise werden *Management-Aspekte* des Estelle-Ausführungsmodells, die ausgehend von der Laufzeitbibliothek im Wesentlichen auf den Abstraktionen der Basisklassen `Module` und `Specification` abgewickelt werden, von *spezifikationsspezifischen Aspekten* (wie konkrete Modulschnittstellen, Variablen, Transitionen etc.) getrennt, die wiederum nur auf der Ebene des generierten Codes eine Rolle spielen. Zugriff auf diese spezifikationsspezifischen Aspekte erfolgt ausschließlich aus generiertem Code heraus, wobei die Kontrolle der Ausführung dieses generierten Codes durch polymorphe Basisklassen auf abstraktem Niveau aus der Laufzeitbibliothek heraus erfolgt.

Wir werden diese Zusammenarbeit in den folgenden Abschnitten näher illustrieren. Dazu konzentrieren wir uns zunächst auf die bei der Codegenerierung implementierten Spezifikationsspezifischen Komponenten und werden auf die Ablaufsteuerung durch die Laufzeitbibliothek später in Abschnitt 3.4 zurückkommen.

3.3.2.2 Modullokale Definitionen

Im letzten Abschnitt haben wir die Verschachtelung der Modul-Klassen und der ihnen jeweils zu Grunde liegenden Header-Klassen in eine zur Struktur der Ausgangsspezifikation analoge Hierarchie betrachtet. Dabei haben wir gesehen, dass jede einzelne Modul-Klasse weitere lokale Klassendefinitionen enthalten kann, mit denen syntaktische Unterstrukturen der Ausgangsspezifikation modelliert werden können.

Im Wesentlichen sind zu einer Modul- bzw. Spezifikations-Klasse die folgenden Unterstrukturen möglich, wobei wir die ersten beiden bereits kennen gelernt haben:

- (Kind-) Modulheader
- (Kind-) Module
- Modulvariablen
- Kanal- und Interaktionspunktdefinitionen
- Typen-, Konstanten- und Variablendefinitionen
- Kontrollzustände
- Funktions- und Prozedurdefinitionen
- Transitionen und Initialisierungstransitionen

Kontrollzustände, Variablen, Modulvariablen und Kanäle (mit ihren Queues) definieren darüber hinaus neben der Modulinstanzhierarchie die wesentlichen Komponenten des *Zustandsraums* der einzelnen Module. Wir werden im Folgenden zunächst die statische Definition der Strukturen und Zustandsraumkomponenten kennen lernen, bevor wir dann in Abschnitt 3.4 zur Implementierung der *(dynamischen) Operationen* zur Zustandsraummodifikation kommen.

3.3.3 Datentypen, Konstanten und Variablen

Estelle kennt neben den aus Pascal übernommenen Typen (vordefinierte Typen, Teilbereichstypen, strukturierte Typen, Pointer, etc.) und Konstanten (`nil`, `true`, `false`) auch den *unspezifizierten Typ* „`...`“ und die *unspezifizierte Konstante* „`ANY` *type*“. Die folgende Estelle-Spezifikation enthält einige typische Beispiele für Konstanten-, Typ- und Variablendefinitionen innerhalb eines Moduls (hier: das Spezifikationsmodul).

Beispiel 3.7-a: Beispiel-Definitionen von Estelle-Typen, -Konstanten und -Variablen

```
 1: SPECIFICATION test;
 2:
 3: TYPE     t1 = integer;
 4:          t2 = 1 .. 1000;
 5:          t3 = RECORD
 6:                   a,b,c: integer;
 7:                   d: real;
 8:                   END;
 9:          t4 = RECORD
10:                   a: integer;
11:                   CASE boolean OF
12:                           false: (b, c: integer;);
13:                           true:  (d: real;);
14:                   END;
15:          t5 = ^t4;
```

```
16:         t6 = ARRAY [1..10] OF t4;
17:         t7 = SET OF 1..10;
18:         t8 = (red, green, blue);
19:         t9 = ...;
20:
21: CONST   c1 = 10;
22:         c2 = ANY integer;
23:
24: VAR     v1: integer;
25:         v2: t3;
26:         { ... }
27:
28: END.
```

(Ende von Beispiel 3.7-a)

Die Implementierung von Datentypen, Konstanten und Variablen erfolgt so weit wie möglich direkt auf die äquivalenten C++-Typen bzw. im Falle von komplexen Typen wie Sets oder Arrays auf geeignete C++-Klassendefinitionen. Die einzelnen Definitionen werden in unserem Beispiel als Definitionen einer Modulinstanz innerhalb der entsprechenden C++-Modulklasse erzeugt:

Beispiel 3.7-b: Auszug aus dem von XEC aus Beispiel 3.7-a generierten Code (`.h`-Datei).

```
 1: class SPEC_test : public Specification {
 2:     public:
 3:         typedef TYPE_integer TYPE_t1;
 4:         typedef Subrange(1, 1000) TYPE_t2;
 5:         struct TYPE_t3 {
 6:                 TYPE_integer COMP_a;
 7:                 TYPE_integer COMP_b;
 8:                 TYPE_integer COMP_c;
 9:                 TYPE_real COMP_d;
10:         } ;
11:         struct TYPE_t4 {
12:                 TYPE_integer COMP_a;
13:                 VARIANT_PART_BEGIN
14:                 struct { /* variant: (CONST_false) */
15:                         TYPE_integer COMP_b;
16:                         TYPE_integer COMP_c;
17:                 } V1;
18:                 struct { /* variant: (CONST_true) */
19:                         TYPE_real COMP_d;
20:                 } V2;
21:                 VARIANT_PART_END
22:         } ;
23:         typedef TYPE_t4* TYPE_t5;
24:         typedef ArrayC< TYPE_t4,1,10 > TYPE_t6;
25:         typedef TypedSet< 1,10 > TYPE_t7;
26:         typedef enum {
27:                 CONST_red,
28:                 CONST_green,
29:                 CONST_blue
30:         }  TYPE_t8;
```

```
31:         typedef UNSPECIFIED_TYPE TYPE_t9;
32:         static const TYPE_integer CONST_c1 = 10;
33:         static const TYPE_integer CONST_c2 = /* ANY_CONST */ 0;
34:         TYPE_integer VAR_v1;
35:         TYPE_t3 VAR_v2;
36: }; /* class SPEC_test */
```

(Ende von Beispiel 3.7-b)

Die meisten der generierten C++-Definitionen entsprechen somit direkt dem typischerweise bei einer manuellen Implementierung vorgefundenen Code. So wird z. B. die Estelle-Definition „`TYPE t1 = integer;`“ in C++ in das äquivalente „`typedef TYPE_integer TYPE_t1;`“ umgesetzt, wobei `TYPE_integer` als Repräsentant des vordefinierten Estelle-Typs in der Laufzeitbibliothek definiert ist.[40] Auch die meisten anderen Implementierungen sind leicht nachvollziehbar.

So werden z. B. Estelle-`RECORD`s in C++-`struct`s mit analogen Komponenten umgesetzt.[41] Eine Besonderheit stellen variante `RECORD`s dar, da die in Estelle bzw. Pascal mögliche Kombination von gruppierten varianten Komponenten (`b` und `c` bzw. `d` in `t4`) und nicht varianten Komponenten (`a` in `t4`) zunächst nicht direkt als C++-`union` darstellbar ist. Durch die Einführung eines anonymen `union`s (versteckt in den Makros `VARIANT_PART_BEGIN` und `VARIANT_PART_END`, siehe Zeilen 13 bis 21 von Beispiel 3.7-b) und von künstlichen Gruppierungs-Strukturen (`V1`, `V2`, ...)[42] konnten die varianten Records jedoch angemessen abgebildet werden, wobei durch bedingte Kompilierung gesteuert werden kann, ob die Varianten auf den selben Speicherplatz abgebildet werden sollen (siehe Beispiel 3.8).

Beispiel 3.8: Auszug aus dem Laufzeitbibliotheks-Modul „`xecrt_estelle.h`“

Die folgende Definition steuert über bedingte Kompilierung abhängig vom Präprozessor-Symbol „`SEPARATE_VARIANTS`“, ob die einzelnen Feldgruppen varianter Records tatsächlich den selben Speicherplatz (durch Nutzung des anonymen Unions) oder separate Speicherbereiche belegen.

40. `TYPE_integer` wird in der aktuellen Version der XECRT als der C++-Typ `int` definiert. Dies entspricht aufgrund der Wertebereichsbeschränkung von `int` nicht ganz der Estelle-Semantik von `integer`, ist aus Performance-Gründen jedoch sinnvoll und wird daher von allen Implementierungsgeneratoren so umgesetzt. Ein Austausch gegen eine semantikkonforme C++-Klasse mit analogem Interface und (durch dynamische Datenhaltung) *unbeschränktem Wertebereich* ist jedoch leicht durch Anpassung der Laufzeitbibliothek implementierbar.

41. XEC erzeugt für `RECORD`-Implementierungen zusätzlich noch (hier ausgelassene) Methoden zur textuellen Ausgabe (`dump(...)`) des Wertes für Traces und zum Debugging.

42. Diese Strukturen gehören übrigens zu den wenigen Situationen, in denen künstlich nummerierte Namen im generierten Code auftreten (siehe Abschnitt 3.2.2). Die Nummerierung ist jedoch strukturlokal und durch die Typdefinition vollständig determiniert, bedingt also keine Nachteile bei der getrennten Übersetzung von Spezifikationsteilen (siehe Kapitel 7).

```
1: #ifdef SEPARATE_VARIANTS
2:         // variants have separated locations
3:         #define VARIANT_PART_BEGIN
4:         #define VARIANT_PART_END
5: #else
6:         // variants share the same location
7:         #define VARIANT_PART_BEGIN      union {
8:         #define VARIANT_PART_END        };
9: #endif
```

(Ende von Beispiel 3.8)

Pointer- und *Aufzählungstypen* werden von XEC auch in C++ direkt als äquivalente Typen implementiert (siehe `t5` und `t8`), wohingegen *Arrays* als Instanziierungen eines C++-Array-Klassentemplates (`Array<class elementType, int nMinIdx, int nMaxIdx>`)[43] implementiert werden, welches sowohl die effiziente Umsetzung der Adressierung[44], als auch eine optionale Index-Bereichsprüfung erlaubt.

Erheblich größerer Aufwand ist zur Implementierung von Estelle-*Sets* erforderlich, da hier neben den Sets mit bekanntem Wertebereich (implementiert durch das C++-Klassentemplate `TypedSet<int nMinIdx, int nMaxIdx>`, siehe `t7`) beim Rechnen mit Sets und bei der Parameterübergabe als Zwischenergebnisse auch Sets mit *unbekanntem Wertebereich* entstehen können. Während andere Estelle-Compiler (z. B. EC) dieses Problem durch Bereichsrestriktion der zulässigen Indexwerte[45] lösen, erzeugt das Klassensystem der XEC Laufzeitbibliothek ggf. automatisch dynamische Darstellungen der Sets ohne Bereichsbeschränkungen (siehe u. a. `class DynamicSet` in `xecrt_estelle.h`).

Die Implementierung von Estelle-*Konstanten* erfolgt direkt als konstante C++-Klassenkomponenten („`static const ...`“, siehe z. B. `c1`)[46], wodurch sie voll in das Namensraum- und Sichtbarkeitsschema integriert werden und entsprechend über ihren voll qualifizierten Namen (oder innerhalb der definierenden Klasse ggf. auch unqualifiziert) angesprochen bzw. von anderen Definitionen unterschieden werden können. Dies vermeidet Namenskollisionen, wie sie bei den in anderen Codegeneratoren üblichen Implementierungen von Konstanten durch *Präprozessormakros* entstehen und ist somit eine der Grundlagen des deterministischen Implementierungsschemas.

---

43. Im generierten Code werden statt des Klassen-Templates `Array<...>` auch `ArrayC<...>` oder `ArrayS<...>` eingesetzt; diese sind Spezialisierungen von `Array<...>`, die sich nur bezüglich der Unterstützung der Debug-Ausgabe einfacher (`ArrayS`) und komplexer (`ArrayC`) Typen unterscheiden.

44. In Estelle ist der minimale Indexwert eines Arrays parametrierbar, in C/C++ ist er fest (0). Entsprechend muss der Index umgerechnet werden, wobei die gewählte Implementierung als Klassen-Template bei konstanten Indizes bzw. dem minimalen Indexwert 0 eine Optimierung dieser Operation durch den C++-Compiler ermöglicht.

45. Dadurch ist es möglich, Sets als Bitstrukturen fester Größe darzustellen, jedoch ist die Implementierungsmethode hier unvollständig.

46. Konstante Komponenten von Klassen müssen gemäß des C++-Standards [ISO98] zwingend `static` attributiert sein, werden aber bei dieser Deklaration auch sofort *abschließend* definiert. Referenzen auf diese Konstanten können vom C++-Compiler durch ihren konstanten Wert ersetzt und somit vollständig wegoptimiert werden.

Einen Sonderfall bilden unspezifizierte Typen („`...`“) und Konstanten („`ANY` *type*“), da deren Auftreten die Estelle-Spezifikation *unvollständig* macht und sie daher eigentlich nur syntaktisch prüfbar, nicht jedoch semantisch interpretierbar ist (Abschnitt 8.2.3.3 in [ISO97]). Um dennoch in solchen Fällen soweit wie möglich zu einer ausführbaren Implementierung zu gelangen, werden diese Definitionen behelfsweise auf die Dummy-Klasse `UNSPECIFIED_TYPE` bzw. bei `ANY`-Konstanten auf einen *Dummy-Wert* (hier `0`) abgebildet. Dies genügt oftmals, um eine für den C++-Compiler prüf- und übersetzbare Implementierung zu erhalten. Es ist offensichtlich jedoch in den meisten Fällen für eine semantisch relevante Implementierung erforderlich, diese Unvollständigkeit durch manuelle Modifikationen des generierten C++-Codes zu beseitigen.

Variablendefinitionen innerhalb von Moduldefinitionen sind Teil des *Zustandsraums* der einzelnen Modulinstanzen und werden entsprechend direkt als Komponenten der Modulklasse implementiert. Dies geschieht dabei sowohl für vordefinierte primitive Typen (z. B. `v1` vom Typ `integer`) als auch für benutzerdefinierte komplexe Typen (z. B. `v2` vom Record-Typ `t3`). Wir werden später sehen, dass auch andere Variablendefinitionen, z. B. innerhalb von Transitionen, Prozeduren oder Funktionen, durch eine durchgängige Modellierung dieser Umgebungen als *Klassen* völlig analog modelliert werden konnten.

3.3.4 Kanäle, Interaktionspunkte und Nachrichtenkommunikation

Der *asynchrone Austausch von Nachrichten* ist in Estelle – neben *gemeinsamen*[47] *Variablen* – das zentrale Paradigma zur Kommunikation zwischen Modulinstanzen. Die Kommunikation erfolgt dabei über *verbundene Interaktionspunkte*, die aufbauend auf eine gemeinsame Kanaldefinition (z. B. „`ch1`“ in Beispiel 3.9-a) – aber mit entgegengesetzten Rollen – typisierte Nachrichten (hier: „`request`“, „`reply`“) ggf. mit Daten-Parametern übermitteln. Neben modulinternen Interaktionspunkten ist die Hauptanwendung die externe Kommunikation der Modulinstanzen über externe Interaktionspunkte („`to_provider`“). Diese werden, genau wie exportierte Variablen („`xdata`“), innerhalb der Modulheader-Definition („`mod_user`“) für alle abgeleiteten Module festgelegt.

Beispiel 3.9-a: Beispiel-Definitionen einer Modulschnittstelle

```
 1: SPECIFICATION test;
 2:
 3: TYPE userdata = ARRAY [1..10] OF integer;
 4:
 5: CHANNEL ch1(user, provider);
 6:         BY user:     request(d: userdata);
 7:         BY provider: reply  (d: integer);
 8:
 9: MODULE mod_user SYSTEMACTIVITY (parameter: integer);
10:         IP to_provider: ch1(user) INDIVIDUAL QUEUE;
11:         EXPORT xdata: integer;
12:         END;
13:
16: END.
```

(Ende von Beispiel 3.9-a)

47. auch „*exportierte* Variablen“

Wir haben in Abschnitt 3.3.2 bereits gesehen, dass XEC Modulheader als Basisklassen der Modulrumpf-Klassen implementiert. Wir werden nun die Grundprinzipien der Implementierung von exportierten Variablen, (externen) Interaktionspunkten und Modulparametern in diesen Modulheader-Klassen näher beleuchten.

Wir beginnen dazu mit der Kanaldefinition („`ch1`“) aus dem obigen Estelle-Beispiel. Diese beinhaltet neben der Festlegung zweier Rollennamen (hier: „`user`“ und „`provider`“) auch eine Menge von Nachrichtendefinitionen für die einzelnen Rollen, die jeweils aus einem Interaktionsnamen (z. B. „`request`“) und einer Parameterliste bestehen.

Während der Ausführung sollen dann über zwei passend verbundene Interaktionspunkte entsprechende Nachrichten zunächst zusammen mit ihren aktuellen Parametern verschickt und dann später (d.h. *asynchron*) am anderen Ende empfangen werden.

Wie für die meisten anderen Strukturen erzeugt XEC für die Kanaldefinition eine modulinterne Klassendefinition („`CHANNEL_ch1`“, siehe Zeilen 4-25 in Beispiel 3.9-b). Für jeden Interaktionstyp dieses Kanals wird u. a. eine entsprechend benannte Methode definiert (siehe „`IA_request`“ in Zeile 12 und „`IA_reply` in Zeile 22). Diese Methoden werden später zum Erzeugen und Verschicken von Nachrichten über einen Interaktionspunkt eingesetzt.

Dies ist möglich, weil diese Interaktionspunkte von XEC als Instanzen ihrer jeweiligen Kanaldefinition modelliert werden, wodurch eine elegante Kombination der Schnittstellenfestlegung in der Kanaldefinition (als Klassendefinition) und ihrer Anwendung anhand der Interaktionspunkte (als Instanzen) möglich wird. Insbesondere erlaubt die Tatsache, dass verbundene Interaktionspunkte dem gleichen Kanal angehören müssen[48], die Details der Kodierung und Typzuordnung der Nachrichten als *internen Aspekt* der jeweiligen Kanal-Klasse zu modellieren.

Wir betrachten dazu die Implementierung des externen Interaktionspunkts „`to_provider`“ in der Modulheader-Klasse „`HEADER_mod_user`“ (siehe Zeile 26-37), die lediglich aus der Komponentendefinition „`CHANNEL_ch1 IP_to_provider`“ besteht. Dadurch wird die Kanal-Klassen-Instanz „`IP_to_provider`“ über die o. g. Vererbung Teil einer jeden abgeleiteten Modulklasse bzw. Modulinstanz.

Beispiel 3.9-b: Auszug aus dem von XEC aus Beispiel 3.9-a generierten Code (`.h`-Datei).

```
 1: class SPEC_test : public Specification {
 2:     public:
 3:         typedef ArrayS< TYPE_integer,1,10 > TYPE_userdata;
 4:         class CHANNEL_ch1 : public IP {
 5:             public:
 6:                 /***** interaction IA_request *****/
 7:                 static const unsigned IA_request_ID = 1;
 8:                 struct IA_request_ARG {
 9:                         TYPE_userdata PAR_d;
10:                 };
11:                 typedef InterAction<   CHANNEL_ch1,
                                           IA_request_ID,
                                           IA_request_ARG
                                       > IA_request_TYPE;
12:                 inline void IA_request (
13:                         const TYPE_userdata& PAR_d
```

48. Sonst können sie nicht verbunden werden; dieser Aspekt wird jedoch auch bereits vom Compiler-Frontend PET statisch geprüft.

```
14:                 );
15:
16:                 /***** interaction IA_reply *****/
17:                 static const unsigned IA_reply_ID = 2;
18:                 struct IA_reply_ARG {
19:                         TYPE_integer PAR_d;
20:                 };
21:                 typedef InterAction<   CHANNEL_ch1,
                                           IA_reply_ID,
                                           IA_reply_ARG
                                       > IA_reply_TYPE;
22:                 inline void IA_reply (
23:                         const TYPE_integer PAR_d
24:                 );
25:         };
26:         class HEADER_mod_user : public Module {
27:             public:
28:                 TYPE_integer PAR_parameter;
29:                 void init (
30:                         TYPE_integer PAR_parameter
31:                 ) {
32:                         this->PAR_parameter = PAR_parameter;
33:                         execInitTrans();
34:                 };
35:                 CHANNEL_ch1 IP_to_provider;
36:                 TYPE_integer EXPVAR_xdata;
37:         };
38: }; /* class SPEC_test */
39:
40:
```

(Ende von Beispiel 3.9-b)

Wie wir später noch genauer sehen werden, genügt es zum Versenden der Interaktion „`request`“ von einer Modulinstanz zu „`mod_user`“ über diesen externen Interaktionspunkt (z. B. mit „`OUTPUT to_provider.request(x);`“), die entsprechende Methode des Interaktionspunkts aufzurufen („`IP_to_provider.IA_request(VAR_x);`“).[49]

Betrachten wir dazu nun die Implementierung dieser Methoden in der von XEC erzeugten „`.cc`“-Datei:

Beispiel 3.9-c: Auszug aus dem von XEC aus Beispiel 3.9-a generierten Code (`.cc`-Datei).

```
1: inline void SPEC_test::CHANNEL_ch1::IA_request (
2:         const SPEC_test::TYPE_userdata& PAR_d
3: ) {
4:         IA_request_TYPE* pIA = new IA_request_TYPE;
5:         pIA->param.PAR_d = PAR_d;
6:         pIA->send(this);
7: }
8:
```

49. Abhängig vom Aufrufkontext wird zum Zugriff auf die Komponente „`IP_to_provider`“ im realen Code eine ggf. notwendige Qualifizierung wie z. B. „`parent()->IP_...`“ eingesetzt (s. u.).

```
 9: inline void SPEC_test::CHANNEL_ch1::IA_reply (
10:         const TYPE_integer PAR_d
11: ) {
12:         IA_reply_TYPE* pIA = new IA_reply_TYPE;
13:         pIA->param.PAR_d = PAR_d;
14:         pIA->send(this);
15: }
```

(Ende von Beispiel 3.9-c)

Zunächst erzeugt die Sende-Methode (z. B. „`IA_request`“ ab Zeile 1) ein Heap-Objekt (hier vom Typ „`IA_request_TYPE`“, siehe Zeile 4 von Beispiel 3.9-c), das die eigentliche Nachricht darstellt und sie entsprechend bis zu ihrem endgültigem Empfang zwischenspeichert. Dieses Nachrichtenobjekt enthält dazu (in der Komponente „`param`“) für jeden Nachrichtenparameter eine entsprechende Unterkomponente gleichen Namens (hier: „`param.PAR_d`“), in der der Wert des Parameters konserviert wird (siehe Zeile 5). Schließlich wird das Nachrichtenobjekt durch Aufruf der Methode „`pIA->send(this)`“ in den Nachrichtentransportmechanismus dieses Interaktionspunkts eingeklinkt und über diesen weitergeleitet.

Um die Erzeugung dieses Nachrichtenobjekts am Interaktionspunkt typsicher, gleichzeitig auf globaler Transportebene aber abstrakt zu gestalten, sind alle Nachrichtenobjekte indirekt Instanzen von der XECRT-Klasse „`InterActionAbstract`“ (siehe Abb. 3-12). Diese Klasse realisiert zusammen mit der XECRT-Klasse „`IP`“ (als Basisklasse für alle Kanäle und damit auch für alle Interaktionspunkte, siehe z. B. Zeile 4 in Beispiel 3.9-b) das abstrakte Management von Verbindungen und Nachrichtentransport. Insbesondere können aufgrund der `XListNode`-Abstammung „`InterActionAbstract`“-Objekte in entsprechende Listen aufgenommen werden, die dann als z. B. Nachrichtenwarteschlangen fungieren.

Die typspezifische Anwendung dieser Basisklasse erfolgt für jeden Nachrichtentyp einer Kanaldefinition (z. B. „`request`“) durch die Definition einer Template-Klasse (z. B. „`IA_request_TYPE`“ in Zeile 11 von Beispiel 3.9-b) aus dem Klassen-Template „`Interaction`“ durch Bindung der Template-Parameter „`Header`“, „`uId`“ und „`Param`“ an die Modulheader-Klasse (hier: „`CHANNEL_ch1`“), die Nachrichten-ID (hier: „`IA_request_ID`“) und die Parameter-Struktur (hier: „`IA_request_ARG`“).

Die Nachrichten-ID (siehe z. B. Zeile 7 oder 17 in Beispiel 3.9-b) ist dabei die fortlaufende Nummer des Nachrichtentyps[50] innerhalb der Kanaldefinition und dient zur Typidentifikation der Nachricht beim Empfänger (wo sie ja nach dem Transport zunächst als abstrakte Instanz von „`InterActionAbstract`“ ankommt, s. u.). Die Parameter-Struktur (siehe z. B. Zeile 8-10 in Beispiel 3.9-b) enthält für jeden Nachrichtenparameter eine entsprechende Komponente und dient, wie wir bereits gesehen haben, zur Aufnahme der Parameterwerte während des Transports.

Ein interessantes Detail ergibt sich bei genauerer Betrachtung der Parameterlisten der Methoden „`IA_request`“ und „`IA_reply`“: Während bei der ersten Methode der Parameter (vom Typ „`TYPE_integer`“) *by-value* übergeben wird, wird bei der zweiten Methode der Parameter (vom Typ „`SPEC_test::TYPE_userdata`“) *by-reference* übergeben (genauer: „`const SPEC_test::TYPE_userdata &`“). Diese Unterscheidung wird an dieser Stelle immer zwischen primitiven Typen und komplexen Typen getroffen und soll den Kopier-Overhead bei der Parameterübergabe by-value an die Sende-Methode für komplexe Typen vermeiden.[51] Dies ist möglich und sinnvoll, da in der Methode ohnehin durch Zuweisung auf die

50. Alternativ zur expliziten Zuweisung wäre auch die Definition eines `enum`-Typs möglich.

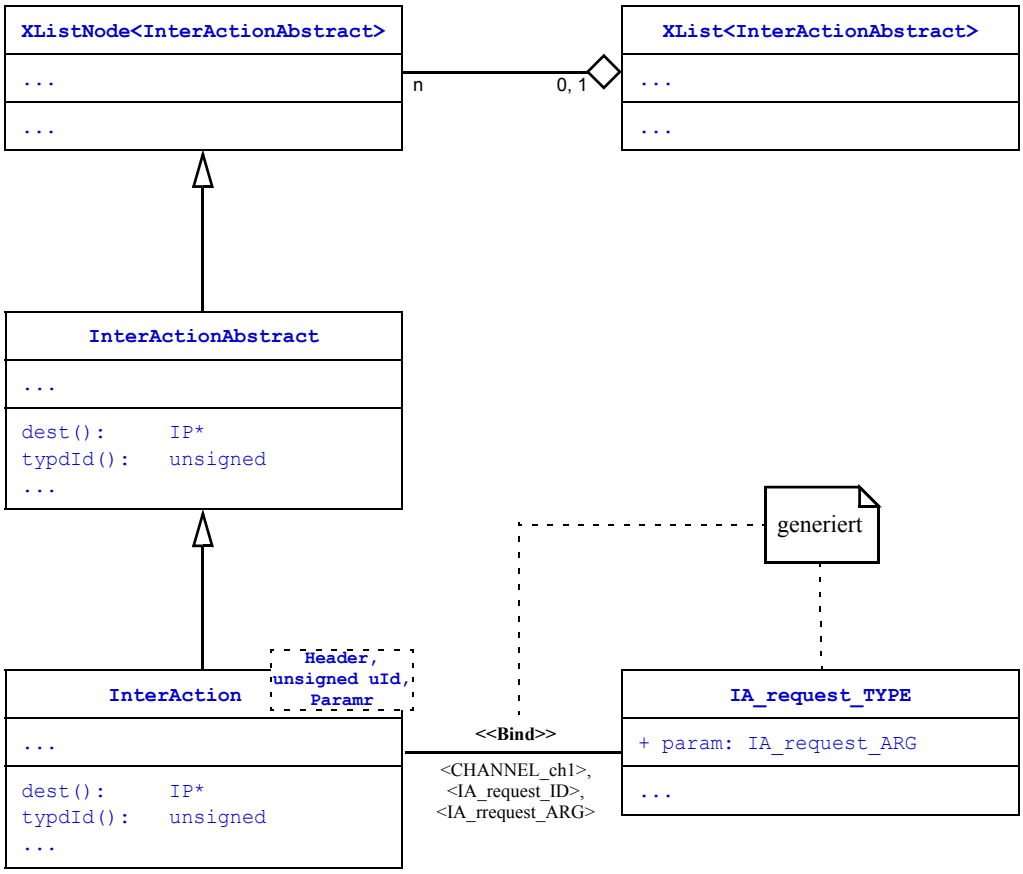


Abbildung 3-12: UML-Diagramm der Interaktionsklassen am Beispiel der Int. „`request`“

`param`-Komponente des Interaktionsobjekts eine Kopie zur Aufbewahrung für den Weitertransport der Nachricht angefertigt werden muss (siehe auch Untersuchungen zur effizienten Datenübertragung in Abschnitt 6).

Wir beschließen damit zunächst das Thema Nachrichtenkommunikation und komplettieren die Vorstellung der Implementierungsmechanismen von Modulschnittstellen mit exportierten Variablen und Modulparametern.

3.3.5 Modulparameter und exportierte Variablen

Exportierte Variablen (siehe z. B. „`xdata`“ in Zeile 11 von Beispiel 3.9-a) werden analog zu modullokalen Variablen (siehe Abschnitt 3.3.2.2) direkt als Komponenten der Modulheader-Klasse definiert (siehe z. B. „`EXPVAR_xdata`“ in Zeile 36 in Abschnitt 3.9-b). Dadurch sind sie sowohl auf der Abstraktionsebene des Modulheaders[52] zugreifbar, als auch innerhalb aller

51. Es könnten zur Vereinfachung auch alle Parametertypen *by-reference* übergeben werden, woraufhin der C++-Compiler bei einem nicht-„LValue“ als aktuellem Parameter (also z. B. einer `int`-Konstante) automatisch ein temporäres referenzierbares Datenobjekt erzeugen würde. Die vorgestellte Diskriminierung vermeidet diesen (wenn auch nur geringen) Overhead.

52. Auf Modulinstanzen wird von „außen“ ausschließlich mit der Abstraktion der Modulheader-Schnittstelle zugegriffen.

abgeleiteten Modulinstanzen als lokale Komponente. Dies ermöglicht zunächst auch, exportierte Variablen genau wie lokale Variablen zu verwenden, außer dass die exportierten Variablen auch von der Vatermodulinstanz modifiziert und gelesen werden können.[53]

Modulparameter (siehe z. B. „`parameter`“ in Zeile 9 von Beispiel 3.9-a) werden zunächst analog zu exportierten Variablen implementiert (siehe z. B. „`PAR_parameter`“ in Zeile 28 in Beispiel 3.9-b), wobei jedoch bereits auf Estelle-Ebene nach Abschluss der Modulinstanziierung kein Zugriff mehr von außen möglich ist. Während der Modulinstanziierung selbst findet die Übergabe der aktuellen Parameter durch Aufruf einer generierten Methode „`init`“ mit adäquater Parameterliste (siehe z. B. Zeile 29-34 in Beispiel 3.9-b) statt. Diese überträgt die Parameterwerte auf die Komponentenvariablen[54] und ruft dann als Abschluss der Modulinitialisierung mit „`Module::execInitTrans()`“ indirekt eine der Initialisierungstransitionen[55] auf (siehe Abschnitt 3.3.7.9).

Konzeptionell scheint es zunächst so, als ob die Übergabe der Initialisierungsparameter statt wie hier in einer separaten Methode eher mit einer Konstruktor-Methode erfolgen sollte. Bei genauerer Betrachtung krankt diese Vorgehensweise jedoch daran, dass in Estelle die Parametrierung bereits mit dem Modulheader (also einer Art Basisklasse) fixiert wird, anstatt die Festlegung der Parameterliste den verschiedenen Modulrümpfen mit ihren unterschiedlichen Anforderungen zu überlassen.[56] Dadurch müssten in unserem Implementierungsschema alle Modulrumpf-Klassen, die den gleichen Header besitzen, explizit jeweils eine Konstruktor-Methode mit der gleichen Parameterliste definieren, um dann diese Parameter lediglich wiederum explizit an die Konstruktor-Methode der Header-Klasse als ihre Basisklasse weiterzureichen. Offensichtlich ist unter diesen Bedingungen die von den Header-Klassen implizit an die Modulrumpf-Klassen vererbte `init`-Methode wesentlich eleganter. Sie ermöglicht aber auch eine Optimierung der Instanziierung von Modulen, die wir in Abschnitt 3.5.4 vorstellen werden.

3.3.6 Funktionen und Prozeduren

Auf den ersten Blick scheint die Abbildung von Funktionen und Prozeduren von Estelle (bzw. Pascal) nach C++ durch entsprechende C++-Methoden mit analogen Parameterlisten und Rückgabetypen möglich zu sein. Als problematisch erweist sich dabei jedoch die Möglichkeit von Estelle, innerhalb jeder Funktion (analog auch für Prozeduren[57]) wiederum interne Funktionen definieren und aufrufen zu können (siehe Funktion „`f`“ in Prozedur „`p`“ in Beispiel 3.10-a). Bei ihrer Ausführung können diese internen Funktionen insbesondere auf lokale Variablen und Parameter ihrer übergeordneten Funktionen (genauer: ihrer Aufrufaktivierungen) zugreifen. Dies

53. Wir werden später sehen, dass diese Ununterscheidbarkeit externer und interner Zugriffe auch erhebliche Nachteile bei der Optimierung der Transitionsauswahl hat (siehe Abschnitt 4.2.2.2).

54. Die Optimierungen der Parameterübergabe erfolgen analog zu denen der Erzeugung von Nachrichten (s. o.).

55. Estelle erlaubt die Definition mehrerer Initialisierungstransitionen, aus denen eine (schaltbare) indeterministisch ausgewählt wird.

56. Da bei der Modulinstanziierung der konkrete Modultyp noch explizit ist, wären unterschiedliche Parameterlisten technisch leicht realisierbar und zudem bei heterogenen Modulrumpfdefinitionen in vielen Fällen auch sehr nützlich.

57. Wir werden uns zur begrifflichen Vereinfachung im Folgenden – soweit keine explizite Differenzierung erfolgt – mit dem Begriff „Funktionen“ auf „Funktionen oder Prozeduren“ beziehen.

wird zusätzlich dadurch verkompliziert, dass auf allen Aufrufebenen direkte oder indirekte Aufrufrekursionen möglich sind, durch die dann beim Aufruf einer internen Funktion mehrere Aktivierungen ihrer umgebenden Funktion und damit beim vorgenannten Variablenzugriff auch mehrere mögliche Zugriffsziele existieren, wobei es eindeutige Regeln zu deren Auflösung gibt (siehe auch Abschnitt 6.2.3 von Annex C in [ISO97]).

Beispiel 3.10-a: Beispiel-Definitionen einer verschachtelten Funktionsdefinition

```
 1: SPECIFICATION test;
 2:
 3: PROCEDURE p(a: integer; VAR b: integer);
 4:         VAR x: integer;
 5:
 6:         FUNCTION f(c:integer): integer;
 7:                 BEGIN
 8:                 f := c+x;
 9:                 END;
10:
11:         BEGIN
12:         x := 5;
13:         b := f(a);
14:         END;
15:
16: END.
```

(Ende von Beispiel 3.10-a)

Gerade dieses Problem der Rekursion auf den verschiedenen Aufrufstufen macht auf Maschinen-Implementierungsebene komplexe Datenstrukturen und ggf. Suchoperationen zum Zugriff auf die einzelnen Stack-Frames der verschiedenen Aktivierungen eingeschachtelter Funktionen erforderlich. Dies war der wesentliche Grund, warum die Sprachen C und C++ das Konzept der funktionslokalen Funktionsdefinitionen nicht beinhalten. Entsprechend existiert auch keine einfache direkte Abbildung derartig verschachtelter Funktionsdefinitionen auf diese Zielsprachen.

Zur Lösung der Problematik implementiert XEC Funktionen und Prozeduren als jeweils eigenständige Klassendefinitionen innerhalb der Klasse ihres Definitionskontexts (siehe „PROC_p“ in Zeile 3-20 und „FUNC_f“ in Zeile 8-15 von Beispiel 3.10-b). Jede dieser Klassen enthält als Komponenten die lokalen Variablen der Funktion bzw. Prozedur und zudem einen Zeiger „pEnv“ (siehe Zeilen 5 und 10) auf die relevante[58] Instanz der umgebenden Klasse. Dieser Zeiger-Wert wird bei der Instanziierung der Klasse als Konstruktor-Parameter übergeben.

Beispiel 3.10-b: Auszug aus dem von XEC aus Beispiel 3.10-a generierten Code (.h-Datei).

```
1: class SPEC_test : public Specification {
2:     public:
3:         class PROC_p {
4:             public:
5:                 SPEC_test* pEnv;
6:                 PROC_p(SPEC_test* p) {pEnv = p;}
7:                         TYPE_integer VAR_x;
8:                 class FUNC_f {
9:                     public:
```

58. Dies ist in einem Aufruf-Stack-Modell zunächst die jeweils oberste.

```
10:                         PROC_p* pEnv;
11:                         FUNC_f(PROC_p* p) {pEnv = p;}
12:                         TYPE_integer call (
13:                                 TYPE_integer PAR_c
14:                         );
15:                 };
16:                 void call (
17:                         TYPE_integer PAR_a,
18:                         TYPE_integer& PAR_b
19:                 );
20:         };
21: }; /* class SPEC_test */
```

(Ende von Beispiel 3.10-b)

Als Implementierung der eigentlichen Funktion dient die Methode „`call`“ der jeweiligen Klasse (siehe Zeilen 12-14 und 16-16 in Beispiel 3.10-b sowie Beispiel 3.10-c), deren Parameterliste und Implementierung der Ausgangsspezifikation entspricht (siehe auch Abschnitt 3.2.2.3).

Beispiel 3.10-c: Auszug aus dem von XEC aus Beispiel 3.10-a generierten Code (`.cc`-Datei).

```
 1: TYPE_integer SPEC_test::PROC_p::FUNC_f::call (
 2:         TYPE_integer PAR_c
 3: ) {
 4:         TYPE_integer RetVal;
 5:         RetVal = (PAR_c + pEnv->VAR_x);
 6:         return RetVal;
 7: }
 8:
 9: void SPEC_test::PROC_p::call (
10:         TYPE_integer PAR_a,
11:         TYPE_integer& PAR_b
12: ) {
13:         VAR_x = 5;
14:         PAR_b = (FUNC_f(this)).call(PAR_a);
15: }
```

(Ende von Beispiel 3.10-c)

Die Instanziierung dieser Klasse erfolgt jeweils durch „`FUNC_f(this)`“ als temporäres Stack-Objekt ausschließlich für den Zeitraum des Methodenaufrufs von „`call`“. Dabei wird als Konstruktor-Parameter ein Zeiger auf den passenden Kontext übergeben. Dies ist im obigen Beispiel für den Aufruf von „`f`“ aus „`p`“ heraus gerade der Instanzpointer auf „`p`“, also „`this`“. Es ergibt sich damit der Aufruf „`FUNC_f(this).call(...)`“ (siehe Zeile 14 von Beispiel 3.10-c). Ein direkt rekursiver Aufruf von „f“ aus sich selbst heraus würde entsprechend durch „`FUNC_f(pEnv).call(...)`“ erfolgen.

Der übergebene Kontext-Zeiger wird vom Konstruktor dann in der internen Funktions-Objekt-Komponente „`pEnv`“ abgelegt, welche dann während der Ausführung des eigentlichen Funktionsaufrufs „`call(...)`“ zum Zugriff auf Komponenten der relevanten Umgebungsinstanz eingesetzt wird. So greift in Zeile 5 von Beispiel 3.10-c „`f`“ auf die Variable „`x`“ von „`p`“ durch „`pEnv->VAR_x`“ zu. Ein Zugriff über mehrere Stufen würde eine mehrstufige Dereferenzierung erfordern. Hätte also das Spezifikationsmodul im obigen Beispiel 3.10-a eine lokale Variable „`y`“, so könnte aus „`f`“ heraus mit „`pEnv->pEnv->VAR_y`“ zugegriffen werden.

Sowohl die o. g. temporäre Instanziierung eines Objekts beim Aufruf, als auch die ggf. mehrstufige Dereferenzierung beim Zugriff haben bei praxisrelevanten statischen Verschachtelungstiefen eine mit einem direkten Funktionsaufruf vergleichbare Performance. So entspricht die Instanziierung des Funktions-Objekts auf dem Stack auf Maschinenebene lediglich der Zuweisung der „`pEnv`"-Variable als Stack-Wert. Aber auch die Tiefe einer möglichen Aufrufrekursion spielt beim Zugriff keine Rolle, da die mehrstufige Dereferenzierung nach statischen Regeln und daher mit festem Aufwand erfolgt. Somit bietet die vorgestellte Implementierungsmethode eine günstige Mischung aus Abstraktion und Effizienz bei der Modellierung verschachtelter Funktionen und Prozeduren.[59]

3.3.7 Transitionen und Transitionsklauseln

Im Folgenden betrachten wir die Implementierung von *Transitionen*, dem tragenden Element der Ausführung von Estelle-Spezifikationen und somit auch späterer Optimierungsmaßnahmen.

Wie wir bereits in Abschnitt 3.2.2.2 gesehen haben, implementiert XEC Estelle-Transitionen als Objekte, also Instanzen von entsprechenden C++-Klassen. Dazu werden die einzelnen Transitionen als Teil der lokalen Definitionen innerhalb ihrer jeweiligen Modul-Klasse als *lokale Klassendefinitionen* implementiert. Diese Klassendefinitionen basieren – abhängig vom Typ der zu Grunde liegenden Transition – auf einer der drei XECRT-Basisklassen `SimpleTrans`, `InputTrans` oder `DelayedTrans`, wobei die letzteren beiden Basisklassen selbst bereits Spezialisierungen der Klasse `SimpleTrans` darstellen (siehe UML-Diagramm in Abb. 3-13).

Die Basisklasse `InputTrans` wird dabei zur Implementierung von Transitionen mit einer `WHEN`-Klausel eingesetzt. Analog dient `DelayedTrans` als Basisklasse für Transitionen mit `DELAY`-Klausel. Alle übrigen Transitionen ohne `WHEN`- und `DELAY`-Klausel[60] haben `SimpleTrans` als (unmittelbare) Basisklasse.

Die Interaktion zwischen den konkreten Transitionsklassen (bzw. ihren Instanzen) und ihren jeweiligen Basisklassen geschieht dabei jeweils durch

(i) das Überschreiben (und spätere Aufrufen) von *virtuellen Methoden* der Basisklassen, oder durch

(ii) die initiale Übergabe von *Parametern* an den Konstruktor der entsprechenden Basisklasse.

Wie wir später noch genauer sehen werden, dient gerade dieser letztere Mechanismus – entgegen dem in Abb. 3-9 auf Seite 38 dargestellten Grundprinzip – auch zur Implementierung einiger Transitionsklauseln. Dies stellt im Vergleich zur dort vorgestellten strikten Implementie-

59. Offensichtlich könnte bei nicht unterstrukturierten Funktionen und Prozeduren eine unmittelbare Implementierung als C++-Funktion erfolgen. Diese Optimierung wird jedoch erst in einer künftigen Version von XEC umgesetzt werden.

60. `DELAY`- und `WHEN`-Klausel schließen sich gegenseitig aus und können daher nicht gemeinsam auftreten.

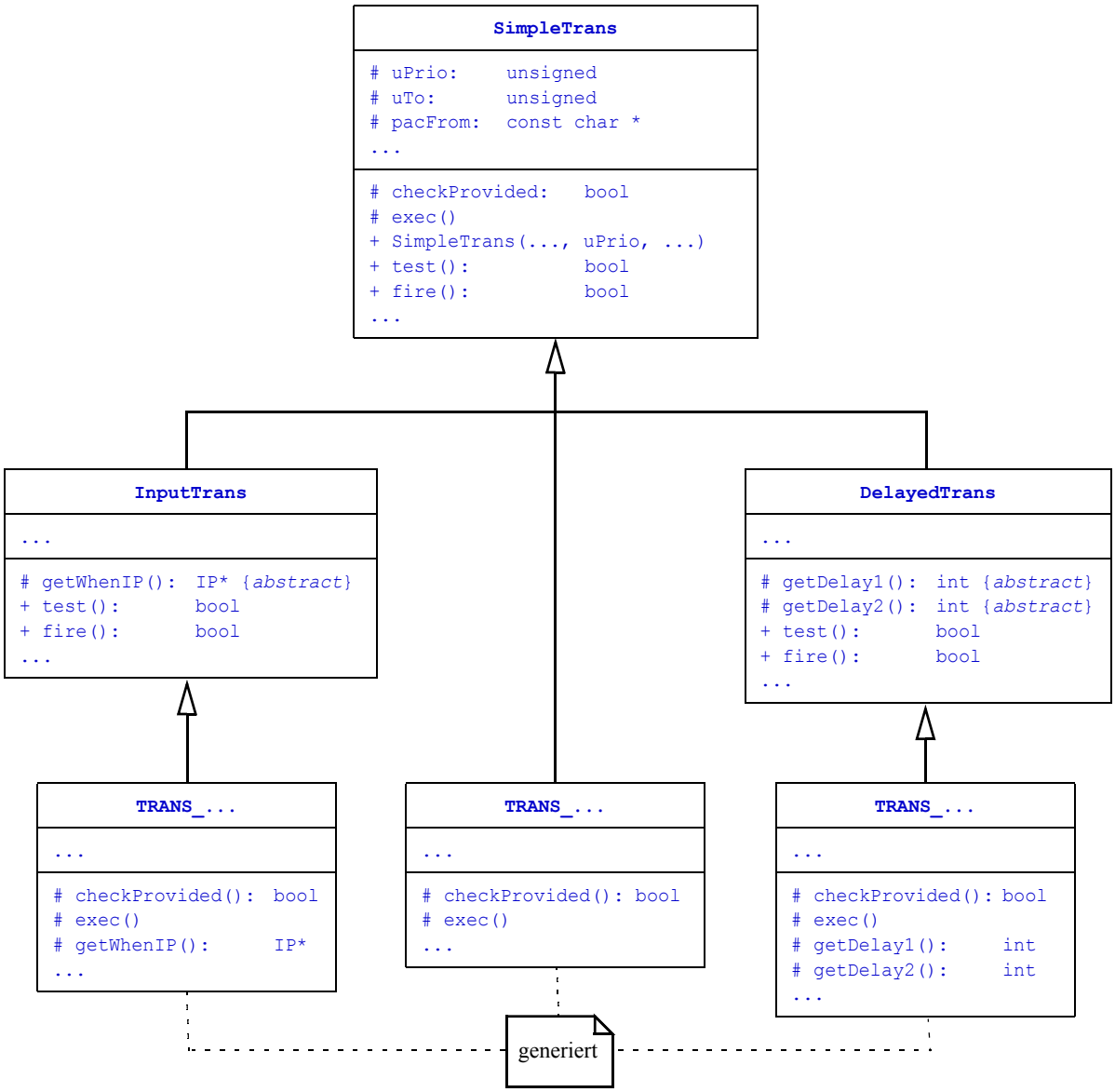


Abbildung 3-13: UML-Diagramm der Transitions-Basisklassen von XEC

rung von Transitionsklauseln durch das Überschreiben virtueller Funktionen[61] ein Zugeständnis an die Effizienz des Übersetzungsvorgangs durch den C++-Compiler[62] und der resultierenden Implementierung dar. [63]

61. Die in Abschnitt 3.2.2.2 als Grundprinzip vorgestellte Implementierungsmethode war in der Tat bei einer früheren Version von XEC und seiner Laufzeitbibliothek im praktischen Einsatz; sie wurde jedoch im Laufe der Zeit durch die hier vorgestellte, effizientere (wenn auch konzeptionell unelegantere) Lösung ersetzt.

62. Gerade bei sehr großen Spezifikationen (wie zum Beispiel der von XTP 4.0) vergrößert eine hohe Anzahl von virtuellen Methoden den Speicher- und Zeitbedarf des C++-Compilers gcc überproportional.

63. Der Aufruf virtueller Methoden bedingt im Vergleich zu statisch gebundenen Methoden Performance-Nachteile auf Grund der zusätzlichen Dereferenzierungsstufe und den damit verbundenen Problemen beim Instruction-Pipelining auf Maschinenebene.

Wir werden nun für die einzelnen Transitions-Komponenten die jeweiligen Implementierungsmethoden auf Basis dieser Transitionsklassen darstellen.

3.3.7.1 Transitions- und Klassennamen

Estelle-Transitionen können in ihrem TRANSITIONS-BLOCK unmittelbar vor dem STATEMENT-PART einen Transitionsnamen festlegen. Wird ein solcher Name angegeben, so dient er mit dem Präfix „TRANS_“ (für normale Transitionen) bzw. „INIT_“ (für Initialisierungstransitionen, siehe Abschnitt 3.3.7.9) auch als Name der entsprechenden Transitionsklasse. Wird jedoch kein solcher Name angegeben, so wird durch Anfügen einer fortlaufenden Nummer an das Präfix ein künstlicher Name erzeugt (z. B. „TRANS_5“).[64] Das Nummerierungsschema ist dabei modullokal (in jedem Modul aufsteigend, beginnend mit 1).

Diese künstliche Namensgebung bei anonymen Transitionen stellt eine unvermeidbare Ausnahme von der Übernahme der Namensstruktur aus der Estelle-Spezifikation in die generierte Implementierung dar. Sie bildet jedoch aufgrund des einfachen, modullokalen Nummerierungsschemas keine wesentliche Beeinträchtigung.

Da die generierten Namen der Transitionen und Transitionsklassen zudem beim Logging, Tracing und Debugging der Spezifikation zum Einsatz kommen, empfiehlt sich ohnehin die Vermeidung anonymer Transitionen durch eine durchgehende explizite Namensgebung auf Spezifikationsebene.

3.3.7.2 Kontrollzustände und TO-Klausel

Die TO-Transitionsklausel beschreibt einen Teil der Schaltwirkung der Transitionen und dient zur Angabe eines neuen Kontrollzustandes für die Modulinstanz. Die TO-Klausel hat als Parameter entweder den Namen eines der zuvor definierten Kontrollzustände des Moduls oder das Schlüsselwort „SAME“, durch das der Verbleib im aktuellen Kontrollzustand explizit gemacht wird. Wird keine TO-Klausel explizit angegeben, so entspricht dies der Klausel „TO SAME“.

Beispiel 3.11-a: Beispiel-Definition einer Transition mit TO-Klausel (Auszug)

```
 1: SPECIFICATION test SYSTEMACTIVITY;
 2:
 3: STATE s1, s2;
 4:
 5: { ... }
 6:
 7: TRANS
 8:         TO s2
 9:         NAME to_s2:
10:         BEGIN
```

64. Es entsteht dadurch kein Namenskonflikt mit *benannten* Transitionen, da ein entsprechender Transitionsname aus einer Zahl bestehen müsste, was gemäß Abschnitt 6.1.3, Annex C von [ISO97] unzulässig ist.

```
11:         END;
12:
13: END.
```

(Ende von Beispiel 3.11-a)

Das obige Estelle-Fragment[65] zeigt eine Transition (namens „to_s2“) mit einer TO-Klausel (siehe Zeile 8), die bei der Ausführung der Transition einen Wechsel des Kontrollzustandes des umgebenden Moduls in den Zustand „s2“ vorschreibt. Dieser Klausel liegt eine Kontrollzustandsraum-Definition (STATE-DEFINITION-PART, siehe Zeile 3) zu Grunde, die entsprechende Kontrollzustände („s1“ und „s2“) definiert.

Zur Implementierung einer solchen Moduldefinition muss neben der Transitionsklasse auch eine geeignete Umsetzung der Kontrollzustandsraum-Definition erfolgen. XEC erzeugt zu diesem Zweck in jedem Modul einen modulklassenlokalen Aufzählungstyp namens „State“, der den einzelnen Kontrollzuständen jeweils Konstanten mit gleichem Namen und dem Präfix „STATE_“ zuordnet (siehe Zeile 3 bis 7 von Beispiel 3.11-b).

Zusätzlich werden in diesem Aufzählungstyp immer auch die beiden Konstanten „STATE_0“ (als präinitialer Zustand bzw. für Module ohne expliziten Kontrollzustand) und „STATES“ (als intern genutzter Indikator für die Anzahl der Kontrollzustände) definiert. Bei Modulen ohne explizite Kontrollzustände (also ohne STATE-DEFINITION-PART) beschränkt sich der Aufzählungstyp „State“ auf gerade diese beiden Werte.

Beispiel 3.11-b: Auszug aus dem von XEC aus Beispiel 3.11-a generierten Code (.h-Datei).

```
 1: class SPEC_test : public Specification {
 2:     public:
 3:         enum State {
 4:                 STATE_0,
 5:                 STATE_s1,
 6:                 STATE_s2,
 7:                 STATES };
 8:         // ...
 9:         class TRANS_to_s2 : public SimpleTrans {
10:             public:
11:                 static const unsigned uTo = STATE_s2;
12:                 // ...
13:                 TRANS_to_s2(Module* pModule)
                         : SimpleTrans( pModule, uHints,
                                        uPriority, pacFrom, uTo) {
14:                 };
15:                 // ...
16:         }; /* TRANS_to_s2 */
17:         // ...
18: }; /* class SPEC_test */
```

(Ende von Beispiel 3.11-b)

65. Zur Vervollständigung des Estelle-Fragmentes ist zumindest noch eine Initialisierungstransition erforderlich.

Die Implementierung der o. g. Transition selbst erfolgt durch die modullokale Klassendefinition „TRANS_to_s2“, in der u. a. eine Konstante namens „uTo“ mit dem Wert „STATE_s2“ aus dem obigen Aufzählungstyp angelegt wird (Zeile 11). Zur Implementierung der „TO SAME“-Klausel wäre diese Konstante auf den Wert 0 gesetzt worden. Die Konstante wird dann als Teil der Konstruktor-Definition dieser Klasse „TRANS_to_s2“ als Parameter an den Konstruktor ihrer Basisklasse „SimpleTrans“ übergeben (Zeile 13).

Dieser Konstruktor der Klasse „SimpleTrans“ wiederum speichert den derart übergebenen Wert schließlich in der Komponentenvariablen „unsigned SimpleTrans::uTo“, wo sie für die Dauer der Existenz des Transitionsobjekts als Wert zur Verfügung steht (siehe auch Abb. 3-13).

3.3.7.3 Die FROM-Klausel

Im Gegensatz zur TO-Klausel, die nur einen Zahlenwert als Parameter hat, können an die FROM-Klausel mehrere Kontrollzustände als Parameter übergeben werden. Dabei können auch über den Umweg von STATESETs (STATE-SET-DEFINITION-PART) mehrere Kontrollzustände über einen Parameter übergeben werden.

Eine FROM-Klausel definiert dabei im Sinne einer Schaltbedingung die Teilmenge der Kontrollzustände, unter denen die Transition bereit sein kann. Wird für eine Transition keine explizite FROM-Klausel angegeben, so bestehen in dieser Hinsicht keine Einschränkungen der Schaltbereitschaft, dies entspricht also einer FROM-Klausel, die als Parameter alle Kontrollzustände des Moduls enthält.

Beispiel 3.12-a: Beispiel-Definitionen einer Transition mit FROM-Klausel (Auszug)

```
 1: SPECIFICATION test SYSTEMACTIVITY;
 2:
 3: STATE s1, s2, s3, s4;
 4:
 5: { ... }
 6:
 7: TRANS
 8:         FROM s1, s3
 9:         NAME from_s1_s3:
10:         BEGIN
11:         END;
12:
13: END.
```

(Ende von Beispiel 3.12-a)

Zur Auswertung der FROM-Klausel ist also zur Laufzeit für jede Transition zu prüfen, ob der konkrete Kontrollzustand des Moduls in der jeweiligen Kontrollzustandsmenge der FROM-Klausel enthalten ist. Hinsichtlich der Anzahl der von einer FROM-Klausel referenzierten Kontrollzustände ist eine sequentielle Suche in einer Liste oder mit einem fest kodierten Ausdruck wie „uState == STATE_s1 || uState == STATE_s2 || ...“ bezüglich der erforderlichen Suchoperationen letztlich mit einem Aufwand *O(n)* bzw. *O(log(n))* (bei binären Suchverfahren) verbunden.

Als effizienteste Implementierungsmethode ergibt sich mit *O(1)* jedoch eine direkte Indizierung eines Arrays von booleschen Werten über den (endlichen) Wertebereich des Modul-Kontrollzustands.[66]

Zu diesem Zweck definiert XEC für jede Transition mit `FROM`-Klausel ein solches Array, in dem für jeden Kontrollzustandswert angegeben ist, ob der entsprechende Kontrollzustand von der `FROM`-Klausel abgedeckt wird oder nicht. Dieses Array wird dann später per Referenz analog zur Implementierung der `TO`-Klausel (siehe Abschnitt 3.3.7.2) an die Basisklasse „`SimpleTrans`“ übergeben und dort im Verlauf der Spezifikationsausführung wiederholt anhand der jeweils aktuellen Kontrollzustände ausgewertet.

Zur kompakten und laufzeitneutralen Implementierung dieses Arrays wurde dabei auf die explizite Definition eines konventionell vorinitialisierten C-Arrays[67] verzichtet, und es wurde stattdessen für diese Anwendung die kompakteste Form einer Array-Konstanten-Definition in C bzw. C++ gewählt: Character-String-Literale (STRING LITERALS [ISO98]).

Character-String-Literale (z. B. `"abc"`) haben den Vorteil, dass sie sehr kompakt dargestellt und in der Klassendefinitions-Hierarchie direkt inline vollständig definiert werden können. Zur Kodierung der Zustands-Menge einer `FROM`-Klausel erzeugt XEC ein solches Character-String-Literal, das für jeden Kontrollzustand (und zusätzlich für den Dummy-Wert „`STATE_0`“ mit dem Wert `0`) ein Zeichen enthält. Zur Kodierung eines *enthaltenen* Kontrollzustands wird der Zeichen-Wert `1` eingesetzt, zur Kodierung eines *nicht enthaltenen* Kontrollzustands entsprechend der Zeichen-Wert `0`.[68] Dargestellt werden diese Werte durch die oktalen Escape-Sequenzen '`\1`' und '`\0`'.

Als Beispiel betrachten wir das String-Literal in Zeile 14 von Beispiel 3.12-b, in dem die `FROM`-Klausel „`FROM s1, s3`“ aus Beispiel 3.15-a implementiert wird. Dieses String-Literal enthält an den Index-Positionen `1` (= `STATE_s1`) und `3` (= `STATE_s3`) jeweils einen nicht-Null-Wert und ansonsten ausschließlich Null-Werte.

Beispiel 3.12-b: Auszug aus dem von XEC aus Beispiel 3.12-a generierten Code (`.h`-Datei).

```
 1: class SPEC_test : public Specification {
 2:     public:
 3:         // ...
 4:         enum State {
 5:                 STATE_0,
 6:                 STATE_s1,
 7:                 STATE_s2,
 8:                 STATE_s3,
 9:                 STATE_s4,
10:                 STATES };
11:         // ...
12:         class TRANS_from_s1_s3 : public SimpleTrans {
```

66. Die konstante Aufwandabschätzung *O(1)* setzt bei *realen* Implementierungen natürlich auch einen konstanten Aufwand beim Speicherzugriff voraus, was jedoch nur beim Direktzugriff auf (bezüglich der Hauptspeicher- und Cachegröße) *kleinen* Datenobjekten der Fall ist. Vor dem Hintergrund der praktisch zu erwartenden Anzahl von Kontrollzuständen kann man die Vorbedingung jedoch als annähernd gegeben annehmen.

67. z. B. „`static const bool aFrom[STATES] = {false, true, ....}`“

68. also der Wert des C-Ausdrucks „`((char) 0)`“ bzw. „`((char) 1)`“, nicht zu verwechseln mit den Ziffer '`0`' bzw. '`1`'

```
13:             public:
14:                 static const char* const pacFrom = "\0\1\0\1\0";
15:                 TRANS_from_s1_s3(Module* pModule)
                        : SimpleTrans( pModule, uHints,
                                       uPriority, pacFrom, uTo) {
16:                 };
17:         }; /* TRANS_from_s1_s3 */
18: }; /* class SPEC_test */
```

(Ende von Beispiel 3.12-b)

Um festzustellen, ob ein konkreter Kontrollzustand in der Zustands-Menge der FROM-Klausel enthalten ist, genügt eine direkte Indexierung wie im Ausdruck „pacFrom[uState]“, wobei der Rückgabewert in C++ direkt als Wahrheitswert (0 oder 1) interpretiert werden kann. Diese Auswertung erfolgt durch die jeweilige Basisklasse der Transitionsklasse, indem ein Pointer auf diesen String an den Basisklassen-Konstruktor übergeben wird (siehe Zeile 15). Diese speichert den derart übergebenen Wert schließlich in der Komponentenvariablen „const char* SimpleTrans::pacFrom“, wo sie für die Dauer der Existenz des Transitionsobjekts für eine wiederholte Auswertung anhand des aktuellen Kontrollzustands zur Verfügung steht (siehe auch Abb. 3-13).

Transitionen, die keine explizite FROM-Klausel besitzen, übergeben anstatt eines String-Literals (das ja nur aus '\1'-Zeichen bestehen würde) einen NULL-Pointer an ihre Basisklasse, die in diesem Sonderfall die Anforderungen der FROM-Klausel auch ohne Index-Zugriff immer als erfüllt betrachtet.

3.3.7.4 Die PRIORITY-Klausel

Die Implementierung von PRIORITY-Klauseln erfolgt völlig analog zu den TO-Klauseln (siehe Abschnitt 3.3.7.2), wobei als Parameter anstatt eines Aufzählungswertes eine nicht negative Integer-Konstante übergeben wird, deren Wert schließlich in der Komponente „unsigned uPrio“ der Transitions-Basisklasse SimpleTrans abgelegt wird und dort für spätere Transitionsauswahlvorgänge zur Verfügung steht.

Da kleinere PRIORITY-Werte höhere Prioritäten bedeuten und Transitionen ohne explizite PRIORITY-Klausel die geringste Priorität besitzen, wird für letztere der Prioritätswert „PRIORITY_NONE“ an die Basisklasse übergeben, welcher von der Laufzeitbibliothek als größtmöglicher unsigned-Wert (UINT_MAX aus „limits.h“) definiert ist.[69]

3.3.7.5 Die PROVIDED-Klausel

Im Gegensatz zu den zuvor diskutierten Klauseln kann die PROVIDED-Klausel im Allgemeinen nicht als einfacher Konstruktor-Parameter implementiert werden, da die PROVIDED-Klausel einen booleschen Ausdruck enthält, der auf Komponenten des erweiterten Zustandsraums des jeweiligen Moduls Zugriff nehmen kann (siehe x in Zeile 6 von Beispiel 3.13-a). Dieser Ausdruck muss im Rahmen des Tests der Transition auf Schaltbereitschaft evaluiert werden, da die Erfülltheit des derart angegebenen Ausdrucks eine Schaltbedingung darstellt.

69. Dies genügt zur Einhaltung der Estelle-Semantik innerhalb des vom Compiler unterstützten Zahlenbereichs für Ganzzahlen (C-integer).

Beispiel 3.13-a: Beispiel-Definitionen einer Transition mit `PROVIDED`-Klausel

```
 1: SPECIFICATION test SYSTEMACTIVITY;
 2:
 3: VAR x: integer;
 4:
 5: TRANS
 6:         PROVIDED x < 5
 7:         BEGIN
 8:         END;
 9:
10: END.
11:
```

(Ende von Beispiel 3.13-a)

Entsprechend implementiert XEC die `PROVIDED`-Klausel in der jeweiligen Transitionsklasse durch das Überschreiben der virtuellen Methode „`void checkProvided()`“ aus der Basisklasse `SimpleTrans` (siehe Zeile 7 in Beispiel 3.13-b).

Beispiel 3.13-b: Auszug aus dem von XEC aus Beispiel 3.13-a generierten Code (`.h`-Datei).

```
 1: class SPEC_test : public Specification {
 2:     public:
 3:         // ...
 4:         class TRANS_1 : public SimpleTrans {
 5:             public:
 6:                 // ...
 7:                 bool checkProvided();
 8:         }; /* TRANS_1 */
 9: }; /* class SPEC_test */
10:
```

(Ende von Beispiel 3.13-b)

Diese für die jeweilige Transition und den erweiterten Zustandsraum des Moduls spezifische Methode implementiert dann den booleschen Ausdruck der `PROVIDED`-Klausel, d. h. der Aufruf dieser Methode liefert direkt das Ergebnis der Auswertung der `PROVIDED`-Bedingung als Wahrheitswert (siehe Beispiel 3.16-c).

Beispiel 3.13-c: Auszug aus dem von XEC aus Beispiel 3.13-a generierten Code (`.cc`-Datei).

```
1: bool SPEC_test::TRANS_1::checkProvided() {
2:         return (parent()->VAR_x < 5);
3: }
4:
```

(Ende von Beispiel 3.13-c)

3.3.7.6 Die WHEN-Klausel

Die WHEN-Klausel referenziert einen (Modul-externen oder -internen) Interaktionspunkt und einen festen Nachrichtentyp. Sie definiert dadurch als *Schaltbedingung* für die jeweilige Transition, dass in der mit dem Interaktionspunkt *assoziierten* Warteschlange

(i) eine Nachricht des angegebenen Typs

(ii) für diesen Interaktionspunkt

(iii) als vorderstes Element

ansteht. Sie definiert gleichzeitig aber auch einen Teil der *Schaltwirkung*, indem die derart empfangene Nachricht nach Ausführung der Transition aus der Warteschlange entfernt wird.

Bei *einfachen* WHEN-Klauseln (siehe z. B. Zeile 15 in Beispiel 3.14-a) könnte die genannte Funktionalität prinzipiell nach dem bereits früher vorgestellten Muster durch geeignete Konstruktor-Parameter (Referenz auf das Interaktionspunkt-Objekt und Nachrichtentyp-ID, siehe Abschnitt 3.3.4) implementiert werden, da beide über die Existenzdauer der Modulinstanz konstant sind.

WHEN-Klauseln können jedoch auch auf dynamisch wechselnde Interaktionspunkte zugreifen, wenn statt eines festen Interaktionspunkts ein (ein- oder mehrdimensionales) *Interaktionspunkt-Array* angegeben wird (siehe Zeile 22 in Beispiel 3.14-a). In diesem Fall kann als Index-Parameter in dieses Interaktionspunkt-Array ein beliebiger, dynamisch ausgewerteter Ausdruck (bzw. bei einem mehrdimensionalen Array eine Folge von Ausdrücken) eingesetzt werden.[70]

Beispiel 3.14-a: Beispiel-Definitionen einer Transition mit WHEN-Klausel (Auszug)

```
 1: SPECIFICATION test SYSTEMACTIVITY;
 2: DEFAULT INDIVIDUAL QUEUE;
 3:
 4: CHANNEL ch1(user, provider);
 5:         BY provider: reply  (d: integer);
 6:
 7: IP to_provider_array: ARRAY [1..10] OF ch1(user);
 8:
 9: VAR x: integer;
10:
11: TRANS
12:         WHEN to_provider_array[x].reply
13:         PROVIDED d < 5
14:         BEGIN
15:         END;
16:
17: END.
```

(Ende von Beispiel 3.14-a)

70. Der beschriebene Mechanismus war vermutlich ursprünglich zum Zugriff auf Interaktionspunkt-Arrays über Transitionen mit ANY-Variablen (siehe Abschnitt 3.3.7.8) vorgesehen, welche bezüglich der einzelnen Transitionsinstanz als *Konstanten* zu betrachten sind. Inwieweit eine bezüglich des erweiterten Zustandraums eines Moduls definierte und damit gegebenenfalls dynamisch wechselnde Zuordnung von Transitionen zu Interaktionspunkten konzeptionell erforderlich und wünschenswert ist, bleibt vor dem Hintergrund der dadurch verursachten Komplikationen (s. u.) unklar.

Deshalb erfolgt die Implementierung der WHEN-Klausel zweistufig auf Grundlage der für Eingabetransitionen vorgesehenen Basisklasse InputTrans (s. o.): Während die (konstante) Nachrichtentyp-ID als (zusätzlicher) Konstruktor-Parameter der Basisklasse übergeben wird (siehe Zeile 26 von Beispiel 3.14-b), erfolgt die Bezugnahme auf einen konkreten Interaktionspunkt durch Überschreiben der Methode „getWhenIP()“, die direkt einen Zeiger auf den referenzierten Interaktionspunkt zurückliefert (siehe Zeile 23).

Der von „getWhenIP()“ zurück gelieferte Zeiger wird bei der Prüfung der Erfülltheit der WHEN-Klausel eingesetzt und dient gegebenenfalls zur Ermittlung der durch die WHEN-Klausel empfangenen Nachricht. Der Zeiger auf diese Nachricht wird entsprechend zwischen der (erfolgreichen) Prüfung der WHEN-Klausel und dem möglichen späteren Schalten der Transition in der Komponente „InterActionAbstract* pIaAbs“ der Basisklasse InputTrans abgelegt.[71]

Dieser zwischengespeicherte Zeiger-Wert dient neben dem internen Management der Basisklasse InputTrans auch auf der Ebene der abgeleiteten Transitionsklasse zum Zugriff auf die gegebenenfalls in der Interaktion enthaltenen Parameter. Dazu ist jedoch zunächst eine Konvertierung vom abstrakten Interaktionstyp InterActionAbstract zu dem bei der Erzeugung der Interaktion ursprünglich erzeugten konkreten Interaktionstyp erforderlich (siehe auch Abb. 3-12 auf Seite 61). Dies wird realisiert durch die Transitionsklassen-Methode „param()“, die den als Nutzlast in der Nachricht enthaltenen Parametertyp „ParamType“ zurückliefert (siehe Zeile 24 von Beispiel 3.14-b). Dazu ist an dieser Stelle eine Typkonvertierung von der Basisklasse InterActionAbstract auf den abgeleiteten konkreten Interaktionstyp erforderlich. Die Korrektheit dieser Konvertierung basiert auf der Korrektheit der typsicheren Verbindungsstruktur von Estelle (siehe auch Abschnitt 3.3.4).

Beispiel 3.14-b: Auszug aus dem von XEC aus Beispiel 3.14-a generierten Code (.h-Datei).

```
 1: class SPEC_test : public Specification {
 2:     public:
 3:         // ...
 4:         TYPE_integer VAR_x;
 5:         class CHANNEL_ch1 : public IP {
 6:             public:
 7:                 /***** interaction IA_reply *****/
 8:                 static const unsigned IA_reply_ID = 1;
 9:                 struct IA_reply_ARG {
10:                         TYPE_integer PAR_d;
11:                 };
12:                 typedef InterAction<CHANNEL_ch1, IA_reply_ID,
```

71. Im Zusammenhang mit der Indizierung von Interaktionspunkt-Arrays zeigt sich eine *semantische Unterspezifikation* von Estelle: Naheliegenderweise muss sich die WHEN-Klausel sowohl bei der Prüfung der Schaltbereitschaft, als auch beim Feuern der Transition auf den selben Interaktionspunkt und damit auf die selbe Nachricht beziehen. Da zwischen der Auswahl einer Transition und ihrem Feuern keine (bezüglich der Transitionsausführung) wirksamen lokalen Änderungen des Zustandsraums der Modulinstanz auftreten können, ist es bezüglich der oben genannten Anforderungen an die WHEN-Klausel hinreichend, wenn die Auflösung des von der WHEN-Klausel referenzierten Interaktionspunkts zu einem *identischen Ergebnis* kommt, also die Index-Ausdrücke zur Indizierung von Interaktionspunkt-Arrays ein konstantes Ergebnis liefern. Dies ist jedoch auf Grund einiger Schwächen in der semantischen Definition von Estelle im Zusammenhang mit indeterministischen Ausdrücken nicht eindeutig sichergestellt (siehe auch Diskussion zu indeterministischen Ausdrücken im Zusammenhang mit „PROVIDED OTHERWISE“ Klauseln in [The95]). Die von XEC erzeugte Implementierung umgeht diese Problematik durch den oben beschriebenen Mechanismus.

```
                                        IA_reply_ARG> IA_reply_TYPE;
13:                 inline void IA_reply (
14:                         const TYPE_integer PAR_d
15:                 );
16:         };
17:         ArrayC< CHANNEL_ch1,1,10 > IP_to_provider_array;
18:
19:         class TRANS_1 : public InputTrans {
20:             public:
21:                 // ...
22:                 typedef SPEC_test::CHANNEL_ch1::IA_reply_TYPE
                                                        IaType;
23:                 IP* getWhenIp() {
                            return &parent()->
                            IP_to_provider_array[parent()->VAR_x];}
24:                 IaType::ParamType& param() const {
                            return ((IaType*) pIaAbs)->param;}
25:                 bool checkProvided();
26:                 TRANS_1(Module* pModule) : InputTrans(
                            pModule, uHints, uPriority,
                            pacFrom, uTo, IaType::uId)
27:                 {};
28:         }; /* TRANS_1 */
29: }; /* class SPEC_test */
```

(Ende von Beispiel 3.14-b)

Zuletzt betrachten wir anhand des vorgestellten Beispiels noch das Zusammenwirken verschiedener Transitionsklauseln[72] und die Anwendung der Methode „`param()`“. So beinhaltet die in Beispiel 3.14-a dargestellte Transition neben der `WHEN`-Klausel auch eine `PROVIDED`-Klausel (Zeile 13), die mit der Variablen „`d`“ Zugriff auf einen Interaktionsparameter aus der `WHEN`-Klausel nimmt. Offensichtlich kann die `PROVIDED`-Klausel nur geprüft werden, nachdem die `WHEN`-Klausel erfolgreich geprüft wurde und die zu empfangende Nachricht (und damit die Nachrichten-Parameter) feststehen. Durch den Einsatz der Methode „`param()`“ ergibt sich schließlich die in Beispiel 3.14-c dargestellte Implementierung der `PROVIDED`-Klausel.[73]

Beispiel 3.14-c: Auszug aus dem von XEC aus Beispiel 3.14-a generierten Code (`.cc`-Datei).

```
1: bool SPEC_test::TRANS_1::checkProvided() {
2:         return (param().PAR_d < 5);
3: }
```

(Ende von Beispiel 3.14-c)

72. Siehe auch Abschnitt 9.6.2 von [ISO97].

73. Die in Abschnitt 9.6.2 von [ISO97] diskutierte unterschiedliche *Bindung* aufgrund der jeweiligen Reihenfolge von WHEN- und PROVIDED-Klauseln wirkt sich hier nicht mehr aus, da die Variablenbindungen hier bereits festgelegt sind und durch entsprechende explizite Referenzen aufgelöst werden.

3.3.7.7 Die DELAY-Klausel

Über die DELAY-Klausel werden zwei Zeitintervalle definiert, in denen die jeweilige Transition schaltbar bzw. optional schaltbar ist. Dazu ist zunächst die spezifikationsweite Festlegung einer Zeitskala erforderlich, die auf informeller Ebene[74] eine Einheit für die im Zusammenhang mit DELAY-Klauseln (einheitenlos) definierten Zeitintervalle angibt (siehe Zeile 2 von Beispiel 3.15-a).

Beispiel 3.15-a: Beispiel-Definitionen einer Transition mit DELAY-Klausel (Auszug)

```
 1: SPECIFICATION test SYSTEMACTIVITY;
 2: TIMESCALE Seconds;
 3:
 4: VAR x: integer;
 5:
 6: TRANS
 7:         DELAY (2*x, 3*x)
 8:         BEGIN
 9:         END;
10:
11: END.
```

(Ende von Beispiel 3.15-a)

Die Implementierung der DELAY-Klausel basiert auf dem Überschreiben der beiden Methoden „`int getDelay1()`“ und „`int getDelay2()`“ (siehe Zeilen 10 und 11 von Beispiel 3.15-b) der für DELAY-Transitionen vorgesehenen Basisklasse `DelayedTrans` (s. o.). Diese beiden Methoden liefern das Ergebnis der Auswertung der beiden DELAY-Parameter, wobei durch Multiplikation mit der in der Spezifikations-Klasse definierten Konstanten `TIMESCALE` (siehe Zeile 3) bereits eine Abbildung von der spezifikationsspezifischen Zeitskala in die implementierungsspezifische Zeitskala erfolgt.

Beispiel 3.15-b: Auszug aus dem von XEC aus Beispiel 3.15-a generierten Code (`.h`-Datei).

```
1: class SPEC_test : public Specification {
2:     public:
3:         static const unsigned TIMESCALE = Timescale::SECONDS;
4:          // ...

5:         TYPE_integer VAR_x;
6:
7:         class TRANS_1 : public DelayedTrans {
8:             public:
9:                 // ...
```

74. Estelle besitzt nur eine schwache Zeitsemantik (siehe Abschnitt 5.3.5 von [ISO97]), die im Wesentlichen lediglich das systemweit einheitliche Voranschreiten der Zeit mit der Ausführung der Spezifikation fordert (siehe auch Abschnitt 3.5.2).

```
10:                 int getDelay1() {
                        return ((2 * parent()->VAR_x)) * TIMESCALE;}
11:                 int getDelay2() {
                        return ((3 * parent()->VAR_x)) * TIMESCALE;}
12:         }; /* TRANS_1 */
13: }; /* class SPEC_test */
```

(Ende von Beispiel 3.15-b)

Die Definition dieser Konstanten `TIMESCALE` basiert dabei auf einer zur `TIMESCALE`-Definition[75] der Ausgangspezifikation gleichnamigen Konstantendefinition in der Laufzeitbibliothek-Klasse `Timescale` (`xecrt_estelle.h`). Diese gibt für gebräuchliche `TIMESCALE`-Werte („`microsecond`[`s`]“, „`millisecond`[`s`]“, „`second`[`s`]“, „`minute`[`s`]“) Umrechnungsfaktoren für die zur internen Zeitmessung eingesetzten Einheit *Mikrosekunden* an und kann bei Bedarf um weitere Einheiten erweitert werden.[76]

Entsprechend ist das Ergebnis des Aufrufs der oben beschriebenen beiden Methoden seitens des Laufzeitsystems bereits von der konkreten Zeitskala der Spezifikation abstrahiert. Wir werden später in Kapitel 7 im Zusammenhang mit *Open-Estelle* noch auf den Nutzen dieser Abstraktion bei der Systemkomposition zurückkommen. Bezüglich der Laufzeit-Performance ergibt sich zudem bei den meist als *Konstanten* formulierten Parametern der `DELAY`-Klauseln durch die vorgestellte Implementierungsmethode der Vorteil, dass die Multiplikation mit dem `TIMESCALE`-Faktor im Allgemeinen zum Zeitpunkt der Übersetzung durch den C++-Compiler bereits aufgelöst werden kann, sodass die Berechnung des Rückgabewertes zur Laufzeit gänzlich vermieden werden kann.

Bei den syntaktischen *Varianten* der `DELAY`-Klausel werden der jeweiligen Semantik entsprechende Implementierungen der beiden Funktionen gewählt:

- Eine einstellige `DELAY`-Klausel („`DELAY(x)`“, entspricht „`DELAY(x,x)`“) wird durch identische Definition der beiden Funktionen implementiert.[77]
- Bei einer `DELAY`-Klausel mit „`*`“-Parameter („`DELAY(x,*)`“) beschreibt der zweite Parameter einen unendlich langen Zeitraum; entsprechend liefert die Funktion „`getDelay2()`“ dann statt eines berechneten Wertes die spezielle Laufzeitbibliothek-Konstante `Timer::INFINIT_SPAN` (definiert als `INT_MAX` aus `limits.h`) zurück, die vom Laufzeitsystem entsprechend behandelt wird.

75. optionale TIME-OPTIONS in einer SPECIFICATION, siehe Abschnitt 7.2.1 ff. von [ISO97]

76. PET und XEC reichen den Namen uninterpretiert durch, es müssen also zur Einführung neuer Timescales nur die entsprechenden Konstantendefinitionen (z. B. „`static const unsigned MILLISECONDS = 1000`“ in der angegebenen Klasse der Laufzeitbibliothek ergänzt werden.

77. Durch den Hint `SimpleTrans::HINT_SAMEDELAY` (siehe Abschnitt 3.4.4) kann später zum Ausführungszeitpunkt der redundante Aufruf der Methode `DelayedTrans::getDelay2()` vermieden werden. Dies umgeht auch unerwünschte Auswirkungen der unter bestimmten Bedingungen kritischen Definition zur Auflösung einstelliger `DELAY`-Klauseln in Estelle (siehe auch Abschnitt 3.3.7.6). Falls nämlich ein indeterministischer Ausdruck angegeben wurde, wird dieser bei der syntaktischen Expansion dupliziert und könnte u. U. unterschiedliche Werte liefern.

3.3.7.8 Die ANY-Klausel

In einer ANY-Klausel einer Transition werden eine oder mehrere Variablen[78] mit jeweils endlichem, diskretem Wertebereich definiert, auf die innerhalb der Transition lesend zugegriffen werden kann (siehe Zeile 10 in Beispiel 3.16-a). Semantisch entspricht eine Transition mit ANY-Klausel (im Sinne einer syntaktischen Expansion) einer Menge von Transitionen, die jeweils für jede Wertekombination der ANY-Variablen entsprechende Konstantendefinitionen gleichen Namens enthalten (im Beispiel also zehn verschiedene Transitionen mit entsprechenden Werten für die Variable „i“).

Beispiel 3.16-a: Beispiel-Definitionen einer Transition mit ANY-Klausel (Auszug)

```
 1: SPECIFICATION test SYSTEMACTIVITY;
 2:
 3: VAR events: ARRAY[1..10] OF integer;
 4:
 5: TRANS
 6:         BEGIN
 7:         END;
 8:
 9: TRANS
10:         ANY i: 1..10 DO
11:         PROVIDED events[i] > 0
12:         BEGIN
13:         END;
14:
15: END.
```

(Ende von Beispiel 3.16-a)

Die bereits beschriebene Implementierung von Transitionen als Instanzen transitionsspezifischer Klassen ermöglicht es XEC, ANY-Klauseln durch entsprechende Mehrfachinstanziierungen dieser Klassen zu implementieren. Wir werden später sehen, dass diese Technik eine individuelle Analyse und Optimierung der verschiedenen Transitionsinstanzen ermöglicht, wodurch deren jeweilige charakteristische Eigenschaften im Verlauf der Ausführung der Implementierung[79] individuell berücksichtigt werden können (siehe Abschnitt 4.3).

Zu diesem Zweck definiert jede Transitionsklasse (sowohl mit als auch ohne ANY-Klausel) eine Methode „`make_instances(..., XList<SimpleTrans>& lst)`“, die entsprechend die jeweiligen Transitionsinstanzen erzeugt und an eine per Referenz als Parameter „`lst`“ übergebene Liste von Transitionen anfügt.

78. Tatsächlich definieren ANY-Klauseln von Transitionen Lauf-*Variablen* anstatt -*Konstanten* (siehe Abschnitt 7.5.9.2 von [ISO97]). Entsprechend können diese z. B. auch *nicht* als Parameter für PRIORITY-Klauseln dienen. Den Variablen darf jedoch kein Wert zugewiesen werden, so dass bei der syntaktischen Expansion schließlich ohne Einschränkungen auch entsprechende Konstantendefinitionen eingesetzt werden können (s. u.).

79. Ein Beispiel ist das jeweilige Kommunikationsaufkommen über verschiedene Interaktionspunkte eines Arrays und damit über verschiedene Instanzen möglicher Eingabe-Transitionen mit ANY-Klausel zur Mehrfachinstanziierung auf alle Array-Felder.

Bei Transitionen ohne ANY-Klausel (siehe Zeile 7 von Beispiel 3.16-b) genügt dabei die einfache Instanziierung der Transitionsklasse und das Anfügen des entstehenden Transitions-Objekts an die oben genannte Liste („appendTo(&lst)“).

Beispiel 3.16-b: Auszug aus dem von XEC aus Beispiel 3.16-a generierten Code (.h-Datei).

```
 1: class SPEC_test : public Specification {
 2:     public:
 3:         // ...
 4:         ArrayS< TYPE_integer,1,10 > VAR_events;
 5:         class TRANS_1 : public SimpleTrans {
 6:             public:
 7:                 inline static void make_instances(
                       Module* pModule,
                       XList<SimpleTrans>& lst)
 8:                           {(new TRANS_1(pModule))->appendTo(&lst);}
 9:                 // ...
10:         }; /* TRANS_1 */
11:         class TRANS_2 : public SimpleTrans {
12:             public:
13:                 BYTE_Subrange(1, 10) ANYPAR_i;
14:                 static void make_instances(
                          Module* pModule,
                          XList<SimpleTrans>& lst);
15:                 // ...
16:         }; /* TRANS_2 */
17: }; /* class SPEC_test */
```

(Ende von Beispiel 3.16-b)

Bei Transitionen *mit* ANY-Klausel wird für die ANY-Variable eine entsprechende Schleife[80] erzeugt (siehe Zeile 2 von Beispiel 3.16-c), in der dann

- die entsprechenden Instanzen der Transitionsklasse erzeugt (Zeile 3) werden,
- die jeweiligen Werte der ANY-Variablen für die weitere Verwendung in der Transition in gleichnamige Objektkomponenten abgelegt werden (Zeile 5) und
- schließlich die nunmehr fertig gestellte Transitionsinstanz an die oben genannte Liste angefügt wird (Zeile 6).

Beispiel 3.16-c: Auszug aus dem von XEC aus Beispiel 3.16-a generierten Code (.cc-Datei).

```
 1: void SPEC_test::TRANS_2::make_instances(
           Module* pModule,
           XList<SimpleTrans>& lst
    ) {
 2:         for (   BYTE_Subrange(1, 10) ANYPAR_i = 1 ;
                    ANYPAR_i <= 10 ;
                    ANYPAR_i = FUNC_succ(ANYPAR_i)
           ) {
 3:                 TRANS_2* pTrans = new TRANS_2(pModule);
 4:
```

80. bzw. bei mehreren ANY-Variablen entsprechend viele ineinander geschachtelte Schleifen

```
 5:                 pTrans->ANYPAR_i = ANYPAR_i;
 6:                 pTrans->appendTo(&lst);
 7:         }
 8: }
 9:
10: SPEC_test::SPEC_test() : Specification(uHints) {
11:         // ...
12:         TRANS_1::make_instances(this, Transitions);
13:         TRANS_2::make_instances(this, Transitions);
14:         setupDone();
15: }
16:
```

(Ende von Beispiel 3.16-c)

Da die Transitions-Methoden „`make_instances(...)`“ jeweils `static` attributiert sind,[81] ist ein Aufruf dieser Methoden seitens des Laufzeitsystems natürlich nicht durch Überschreiben einer virtuellen Methode dieser Transitionsklassen möglich. Stattdessen werden die Aufrufe für alle Transitionen einer Modul-Klasse (im obigen Beispiel die Spezifikationsklasse selbst) in ihrem vom Codegenerator erzeugten Konstruktor getätigt (siehe Zeile 10 bis 15). Als Ziel-Liste für die Transitions-Objekte wird dabei direkt die Komponente `Transitions` der Basisklasse `Module` (siehe auch Beispiel 3.23) angegeben, die während der späteren Ausführung als Basis für die Transitionsauswahl dienen wird.

Entsprechend werden über den beschriebenen Mechanismus bei der Instanziierung einer Modulklasse automatisch auch die Instanzen der jeweiligen Transitionen erzeugt, wobei mögliche `ANY`-Klauseln der Transitionen durch entsprechende Mehrfachinstanziierungen aufgelöst werden.

3.3.7.9 Initialisierungstransitionen

Initialisierungstransitionen werden in Estelle syntaktisch weitgehend analog zu regulären Transitionen definiert (siehe Abschnitt 7.5.10 von [ISO97]). Sie unterscheiden sich lediglich darin, dass sie mit dem Schlüsselwort „`INITIALIZE`“ statt mit „`TRANS`“ beginnen und nur `PROVIDED`- und `TO`-Klauseln besitzen dürfen.[82]

Konzeptionell und semantisch haben sie jedoch wenig mit regulären Transitionen gemeinsam, da sie weder atomar ausgeführt werden (sie werden noch während der Ausführung des `INIT`-Statements zur Instanziierung ihres Moduls durch dessen Vatermodul ausgeführt), noch semantisch einen echten Zustandsübergang der Modulinstanz gemäß der *next-state*-Relation definieren. Stattdessen dienen sie zur lokalen Initialisierung des Modulzustandes vom *präinitialen* in den *initialen* Zustand.

Eigentlich spielen Initialisierungstransitionen eher die Rolle von imperativ vom Vatermodul aufgerufenen Initialisierungs-Prozeduren, die zusätzlich bei Modulen mit explizitem Kontrollzustand durch eine `TO`-Klausel den initialen Kontrollzustand festlegen.[83]

---

81. Sie dienen ja erst zur Erzeugung der Transitionsinstanzen.

82. Konzeptionell wäre auch eine `ANY`-Klausel akzeptabel, sie ist jedoch in der Positivliste von Abschnitt 7.5.10.2 von [ISO97] nicht explizit enthalten.

Nichtsdestotrotz können für ein Modul *mehrere* Initialisierungstransitionen angegeben werden, zwischen denen (sofern sie jeweils bezüglich ihrer `PROVIDED`-Klausel schaltbar[84] sind) eine indeterministische Auswahl erfolgt.

Auf Ebene des generierten Codes behandelt XEC Initialisierungstransitionen völlig analog zu regulären Transitionen. Der Unterschied besteht neben dem Namens-Präfix „`INIT_`“ statt „`TRANS_`“ (siehe Abschnitt 3.3.7.1) nur darin, dass beim Aufruf der Methode „`make_instances(...)`“ bei Initialisierungstransitionen nicht die Liste `Transitions` (siehe Abschnitt 3.3.7.8 und Beispiel 3.16-b) als Ziel angegeben wird, sondern die Liste `InitTransitions`, die ebenfalls Komponente der Basisklasse `Module` ist (siehe Zeile 3 von Beispiel 3.17). Aus dieser Liste wird dann zum Abschluss der Modulinitialisierung[85] unter Einsatz der auch für reguläre Transitionen benutzten Auswahlmechanismen eine Initialisierungstransition ausgewählt und sofort ausgeführt.

Beispiel 3.17: Auszug aus dem von XEC zu einer Initialisierungstransition generierten Code.

```
1: SPEC_test::SPEC_test() : Specification(uHints) {
2:         // ...
3:         INIT_1::make_instances(this, InitTransitions);
4:         TRANS_2::make_instances(this, Transitions);
5:         setupDone();
6: }
```

(Ende von Beispiel 3.17)

In diesem Zusammenhang fällt ein Mangel in der semantischen Definition von Initialisierungstransitionen in [ISO97] auf: Ist keine der angegebenen Initialisierungstransitionen eines Moduls bei seiner Instanziierung schaltbar (auf Grund nicht erfüllter `PROVIDED`-Bedingungen), so verbleibt die Modulinstanz zunächst in einem *präinitialen Zustand*. Dies bedeutet speziell bei Modulen mit explizitem Kontrollzustand, dass *kein eindeutiger Kontrollzustand* zugeordnet ist. Diese Situation erscheint wenig sinnvoll, wird jedoch im Estelle-Standard nicht diskutiert oder als Fehler eingestuft.[86]

XEC ordnet als Behelfslösung in einem solchen Fall der Modulinstanz zunächst den internen Kontrollzustand „`STATE_0`“ zu (siehe Abschnitt 3.3.7.2), der zu diesem Zweck als zusätzlicher, nicht explizit auf Spezifikationsebene zugänglicher Kontrollzustand dient. Entsprechend sind in diesem Zustand auch alle *expliziten* `FROM`-Klauseln nicht erfüllt (siehe Abschnitt 3.3.7.3). Transitionen ohne explizite `FROM`-Klausel sind von dieser Beschränkung nicht betroffen, und so kann später durch Feuern einer Transition mit expliziter `TO`-Klausel (aber ohne explizite `FROM`-Klausel) in einen regulären Kontrollzustand gewechselt werden.

---

83. Im Block von Initialisierungstransitionen können im Gegensatz zu den in dieser Hinsicht beschränkten Estelle-Prozeduren auch Estelle-spezifische Operationen wie z. B. das `OUTPUT`-Statement ausgeführt werden.

84. Eine bedingte Schaltbarkeit einer Initialisierungstransition ist nur anhand möglicher Initialisierungsparameter des Moduls sinnvoll, da nur diese zum Zeitpunkt der Auswahl der Initialisierungstransition eine individuelle Eigenschaft der Modulinstanz bedingen können.

85. Dies geschieht in der in Zeile 5 aufgerufenen Laufzeitbibliotheks-Methode „`setupDone()`“.

86. Der Estelle-Standard macht in diesem Zusammenhang lediglich die Vorgabe „*Only one transition-block is executed when the module is initialized, [...].*“ (Abschnitt 7.5.10.3 von [ISO97]). Diese Anforderung ist jedoch im Falle einer leeren Menge schaltbarer Initialisierungstransitionen nicht im Sinne von „*exactly one* ...“ erfüllbar.

3.3.7.10 Der Transitionsblock

Der Transitionsblock entspricht im Grunde einer parameterlosen Prozedur, die im Verlauf des Feuerns der Transition ausgeführt wird.

Beispiel 3.18-a: Beispiel-Definition einer Transition mit Block (Auszug)

```
1: SPECIFICATION test SYSTEMACTIVITY;
2:
3: VAR     x: integer;
4:
5: TRANS
6:     BEGIN
7:     x := x+1;
8:     END;
9: END.
```

(Ende von Beispiel 3.18-a)

Entsprechend wird der Transitionsblock nahezu identisch zu den in Abschnitt 3.3.6 vorgestellten Prozeduren implementiert. Die Unterschiede bestehen letztlich nur darin, dass zum Aufruf keine Prozedur-Instanz erzeugt werden muss (dazu dient die Transitionsklassen-Instanz, die ja bereits existiert) und die Aufruf-Methode durch Überschreiben der virtuellen Basisklassen-Methode „`void SimpleTrans::exec()`“ implementiert wird (siehe Zeile 8 von Beispiel 3.18-b bzw. Beispiel 3.18-c).

Beispiel 3.18-b: Auszug aus dem von XEC aus Beispiel 3.18-a generierten Code (`.h`-Datei).

```
1: class SPEC_test : public Specification {
2:     public:
3:         TYPE_integer VAR_x;
4:         // ...
5:         class TRANS_1 : public SimpleTrans {
6:             public:
7:                 // ...
8:                 void exec();
9:         }; /* TRANS_1 */
10: }; /* class SPEC_test */
```

(Ende von Beispiel 3.18-b)

Beispiel 3.18-c: Auszug aus dem von XEC aus Beispiel 3.18-a generierten Code (`.cc`-Datei).

```
1: void SPEC_test::TRANS_1::exec() {
2:         parent()->VAR_x = (parent()->VAR_x + 1);
3: }
```

(Ende von Beispiel 3.18-c)

3.4. Dynamische Implementierungsstrukturen

Die Estelle-Semantik basiert auf einem *Server-Modell* (siehe auch Abschnitt 4.2.1.1), dem als atomare Operationen die Auswahl einer Menge zu schaltender Transitionen und ihre Ausführung[87] zu Grunde liegt. Dazu wird das System in eine feste Menge von Subsystemen unterteilt, die nebenläufig zueinander jeweils Folgen von Transitionsauswahl- und -Ausführungszyklen durchlaufen.

Innerhalb eines (Sub-) Systems kann die Auswahl einer Menge schaltbarer Transitionen bzw. Module in vier aufeinander aufbauende Ebenen gegliedert werden:

- der Test einer einzelnen *Transitionsklausel* auf *Erfülltheit*[88] (Abschnitt 3.3.7),
- der Test einer einzelnen *Transition* auf *Schaltbereitschaft*[89] (Abschnitt 3.4.1),
- der Test einer *Modulinstanz* auf *Schaltbarkeit*,[90] also die lokale Auswahl einer *schaltbaren*[91] *Transition* innerhalb einer Modulinstanz bzw. der Nachweis der Nichtexistenz einer solchen Transition (Abschnitt 3.4.2) und
- die globale Auswahl einer Menge *schaltbarer Transitionen* bzw. *Module* aus einem Modulinstanz-Teilbaum bis hinauf zu einem ganzen *(Sub-) System* (Abschnitt 3.4.3).

In den folgenden Abschnitten betrachten wir aufbauend auf dem bereits vorgestellten Test einzelner Transitionsklauseln auf Erfülltheit die Implementierungsansätze zu diesen verschiedenen Ebenen auf Basis des Laufzeitsystems XECRT.

3.4.1 Testen und Ausführen einzelner Transitionen

Wie wir bereits in Abschnitt 3.3.7 gesehen haben, werden Transitionen von XEC als Instanzen von Klassen definiert, die durch Parametrierung des Basisklassen-Konstruktors oder durch Überschreiben virtueller Methoden die einzelnen *Transitionsklauseln* implementieren. Der Test der einzelnen Transitionen beziehungsweise das Schalten der Transitionen erfolgt dann durch entsprechende Methodenaufrufe aus diesen Basisklassen heraus. Wir stellen nun die dafür wesentlichen Komponenten und Methoden dieser Basisklassen vor.

---

87. Wir benutzen für das *Ausführen* einer *Transition* auch gleichberechtigt die in der deutschsprachigen Literatur verbreiteten Begriffe „*Feuern*“ („fire“) und „*Schalten*“. Analog kommen diese Begriffe auch für die *Modulinstanz* zur Anwendung, die die ausgeführte Transition enthält.

88. „*satisfied*“, siehe Abschnitt 9.6.2 von [ISO97]

89. Eine Transition ist *(schalt-) bereit* („*enabled*“), wenn ihre `WHEN`-, `PROVIDED`- und `FROM`-Klauseln erfüllt sind (siehe Abschnitt 9.6.2 von [ISO97]).

90. Wir bezeichnen in Erweiterung der Begriffsbildung in [ISO97] genau solche *Module*, die eine schaltbare Transition anbieten, als *schaltbar*.

91. Eine Transition ist *schaltbar* („*fireable*“), wenn sie (1.) bereit ist, (2.) eine mögliche `DELAY`-Klausel erfüllt ist (also die Bereitschaft ausreichend lange unterbrechungsfrei bestanden hat) und sie (3.) unter den übrigen Transitionen ihrer Modulinstanz mit diesen Eigenschaften maximale Priorität hat (siehe Abschnitt 9.6.2 von [ISO97]).

3.4.1.1 Klasse „SimpleTrans“

Die XECRT-Klasse SimpleTrans dient als unmittelbare Basisklasse für die generierten Transitionsklassen zu Transitionen *ohne* WHEN- oder DELAY-Klauseln, sie stellt jedoch auch als Basisklasse für die später diskutierten Klassen InputTrans und DelayedTrans die Grundlage für alle übrigen Transitionsklassen dar.

Zu diesem Zweck enthält sie zunächst einige Komponenten zur Ablage von Konstruktor-Parametern zur Repräsentation der Transitionsklauseln PRIORITY, TO und FROM (siehe Zeilen 4 und 6 von Beispiel 3.19-a sowie Abschnitt 3.3.7). Auf diese Komponenten wird später mit den Methoden „priority()“, „to()“ und „bool checkFrom(unsigned nState)“ lesend zugegriffen (siehe Zeilen 17 bis 19), wobei letztere die Auswertung der als char-Array kodierten FROM-Klausel (siehe Abschnitt 3.3.7.3) leistet.

Beispiel 3.19-a: Auszug aus der Klasse SimpleTrans (xecrt_modules.h)

```
 1: class SimpleTrans : public XListNode<SimpleTrans> {
 2:         // ...
 3:     private:
 4:         unsigned uPrio, uTo;
 5:         Module* pModule;
 6:         const char* pacFrom;
 7:         bool bReady;
 8:
 9:     protected:
10:         virtual bool checkProvided()    {return true;}
11:         virtual void exec()             {}
12:
13:     public:
14:         // ...
15:         virtual bool test();
16:         virtual void fire();
17:         unsigned priority() const       {return uPrio;}
18:         unsigned to() const             {return uTo;}
19:         bool checkFrom(unsigned uState)
                            {return !pacFrom || pacFrom[uState];}
20: };
```

(Ende von Beispiel 3.19-a)

Die virtuellen Methoden „checkProvided()“ und „exec()“ (Zeilen 10 und 11) repräsentieren die PROVIDED-Klausel und den Transitionsblock und werden entsprechend von den generierten Transitionsklassen gegebenenfalls überschrieben (siehe Abschnitt 3.3.7.3 bzw. Abschnitt 3.3.7.10).

Das Testen und Feuern einer Transition wird durch die Methoden „test()“ und „fire()“ implementiert (Zeilen 15 und 16). Diese beiden Methoden sind virtual-attributiert, da sie von den Subklassen InputTrans und DelayedTrans gemäß deren speziellen Anforderungen zur Implementierung der WHEN- bzw. DELAY-Klausel überschrieben werden.

Die Methode „SimpleTrans::test()“ berücksichtigt dagegen lediglich die Transitionsklauseln FROM und PROVIDED, da bei einfachen Transitionen nur diese eine Schaltbedingung implizieren[92], die anhand einer einzelnen Transition überprüfbar ist.[93] Sie ruft dazu die Metho-

de „checkFrom(unsigned)“ mit dem gegenwärtigen Kontrollzustand der Modulinstanz als Parameter und die Methode „checkProvided()“ auf und liefert das Ergebnis der Konjunktion der beiden booleschen Ausdrücke zurück (siehe Zeilen 2 bis 4 von Beispiel 3.19-b).[94]

Dies implementiert den Test der beiden Transitionsklauseln insbesondere unabhängig von ihrem expliziten Auftreten in der Spezifikation: So führt zum Beispiel das Auslassen einer expliziten PROVIDED-Klausel dazu, dass in der Transitionsklasse die Methode „SimpleTrans::checkProvided()“ nicht überschrieben wird und entsprechend letztere mit ihrem Default-Ergebnis „true“ (siehe Zeile 10 von Beispiel 3.19-a) beim Transitionstest ausgewertet wird.

Beispiel 3.19-b: Auszug aus der Klasse SimpleTrans (xecrt_modules.cc)

```
 1: bool SimpleTrans::test() {
 2:         bReady = checkFrom(module()->state()) &&
 3:                  checkProvided();
 4:         return bReady;
 5: }
 6:
 7: void SimpleTrans::fire() {
 8:         exec();
 9:         if (uTo)
10:                 pModule->state(uTo);
11:         bReady = false;
12: }
13:
```

(Ende von Beispiel 3.19-b)

Ähnlich einfach verläuft das *Schalten* einer Transition durch die Methode „SimpleTrans::fire()“ (siehe Zeilen 7 bis 12 von Beispiel 3.19-b): Nach Ausführung des Transitionsblocks wird bei Transitionen mit expliziter TO-Klausel (d.h. $uTo \neq 0$) lediglich noch der entsprechende Kontrollzustand gesetzt, da lediglich diese Transitionsklausel bei einfachen Transitionen eine Schaltwirkung besitzen kann.

3.4.1.2 Klasse „InputTrans“

Transitionsklassen zu Transitionen, die eine WHEN-Klausel besitzen, haben als Basisklasse InputTrans, welche wiederum eine Subklasse der zuvor beschriebenen Klasse SimpleTrans ist (siehe Abb. 3-13 auf Seite 66).

92. Eine TO-Klausel impliziert keine Schaltbedingung und eine ANY-Klausel wurde bereits durch entsprechende mehrfach-Instanziierung der Transitionsklasse aufgelöst.

93. Die PRIORITY-Klausel ist nur oberhalb der Ebene des Tests einzelner Transitionen relevant.

94. Das Ergebnis der Auswertung wird insbesondere auch in der Transitions-Komponente „bReady“ zwischengespeichert, die im Wesentlichen nur für das Debugger-Interface (siehe Abschnitt 3.6.5) und einige spezielle Optimierungsmethoden erforderlich ist.

Zu diesem Zweck enthält sie eine Komponente „`unsigned uIaId`“ (Zeile 3 von Beispiel 3.20-a), in der die als Konstruktor-Parameter von ihrer Transitionsklasse übergebene Interaktions-ID abgelegt wird. Diese identifiziert den Nachrichtentyp, der aus dem von der `WHEN`-Klausel referenzierten Interaktionspunkt empfangen werden soll (siehe Abschnitt 3.3.7.6).

Beispiel 3.20-a: Auszug aus der Klasse `InputTrans` (`xecrt_modules.h`)

```
 1: class InputTrans : public SimpleTrans {
 2:     private:
 3:         unsigned uIaId;
 4:     protected:
 5:         // ...
 6:         virtual IP* getWhenIp() = 0;
 7:         InterActionAbstract* pIaAbs;
 8:     public:
 9:         bool test();            /* override */
10:         void fire();            /* override */
11: };
```

(Ende von Beispiel 3.20-a)

Diesen Interaktionspunkt ermittelt sie durch Aufruf der abstrakten virtuellen Funktion „`getWhenIp()`“, die von der generierten Transitionsklasse zwingend überschrieben wird.[95]

Zur Zwischenspeicherung einer Referenz auf die bei einem erfolgreichen Transitionstest ermittelte (d.h. zu empfangende) Interaktion zwischen dem eigentlichen Test und dem späteren Schalten der Transition dient die Komponente „`pIaAbs`“ (Zeile 7).

Der eigentliche Transitionstest und das Schalten der Transition wird durch Überschreiben der beiden Methoden „`test()`“ und „`fire()`“ der Basisklasse `SimpleTrans` realisiert, die die entsprechenden Funktionen der Basisklasse für die Implementierung der Schaltbedingung und der Schaltwirkung der `WHEN`-Klausel erweitert.

Die *Schaltbedingung* der `WHEN`-Klausel besteht darin, dass in der Warteschlange zu dem referenzierten Interaktionspunkt als erstes Element eine Nachricht an diesen Interaktionspunkt[96] steht, deren Typ dem in der `WHEN`-Klausel referenzierten Nachrichtentyp entspricht.

Entsprechend ermittelt die Methode „`InputTrans::test()`“ zunächst den gegenwärtig[97] referenzierten Interaktionspunkt (Aufruf von „`getWhenIp()`“ siehe Zeile 2 von Beispiel 3.20-b) und prüft durch Aufruf der Methode „`peek(uIaId)`“, ob über diesen Interaktionspunkt eine Nachricht des geforderten Typs empfangen werden kann. Ist dies möglich, so wird ein Zeiger auf die entsprechende Nachricht zurückgeliefert, andernfalls ist der Rückgabewert `NULL`.

---

95. Die `WHEN`-Klausel ist an dieser Stelle in der Ausgangstransition nicht optional, da diese Basisklasse nur bei Transitionen mit `WHEN`-Klausel eingesetzt wird.

96. Auch wenn durch „`COMMON QUEUE`“-Attributierungen sich mehrere Interaktionspunkte eine Warteschlange teilen können, so kann jede einzelne Nachricht jedoch (bei unveränderter Verbindungsstruktur) nur durch *genau einen Interaktionspunkt* empfangen werden. Entsprechend ist beim Empfang die Überprüfung des korrekten Ziel-Interaktionspunkts erforderlich.

97. Wie wir bereits oben diskutiert haben, kann bei Interaktionspunkt-Arrays der referenzierte Interaktionspunkt während der Ausführung der Spezifikation geändert werden.

Beispiel 3.20-b: Auszug aus der Klasse `InputTrans` (`xecrt_modules.cc`)

```
 1: bool InputTrans::test() {
 2:         pIaAbs = getWhenIp()->peek(uIaId);
 3:         bReady = pIaAbs && SimpleTrans::test();
 4:         return bReady;
 5: }
 6:
 7: void InputTrans::fire() {
 8:         SimpleTrans::fire();
 9:         delete pIaAbs;
10:         pIaAbs = NULL;
11: }
```

(Ende von Beispiel 3.20-b)

Das Ergebnis dieser Abfrage wird der oben beschriebenen Komponente `pIaAbs` zugewiesen und anschließend werden – sofern eine passende Nachricht gefunden wurde – die übrigen Klauseln durch Aufruf der ursprünglichen Methode „`SimpleTrans::test()`“ evaluiert (Zeile 3).

An dieser Stelle ist es wichtig, dass der Aufruf von „`SimpleTrans::test()`“ nur erfolgt, wenn die Erfülltheit der `WHEN`-Klausel bestätigt ist und die Komponente `pIaAbs` auf die entsprechende Nachricht zeigt, da durch eine entsprechende `PROVIDED`-Klausel ein Zugriff auf die Parameter dieser Nachricht erfolgen kann. Dieser erfolgt gerade über die `InputTrans`-Komponente `pIaAbs`, welche über die in der Transitionsklasse generierte Methode „`param()`“ zum Zugriff auf Interaktionsparameter genutzt wird (siehe auch Zeile 24 von Beispiel 3.14-b und Beispiel 3.14-c auf Seite 75).

Die Schaltwirkung einer `WHEN`-Klausel besteht darin, dass die empfangene Nachricht nach Ausführung des Transitionsblocks aus ihrer Warteschlange entfernt wird. Entsprechend wird in der Methode „`InputTrans::fire()`“ zunächst die ursprüngliche Methode „`SimpleTrans::fire()`“ (Zeile 8 von Beispiel 3.20-b) aufgerufen und anschließend die durch `pIaAbs` referenzierte Nachricht mit `delete` freigegeben, wodurch sie unter anderem (als „`XListNode<InterActionAbstract>`“) aus der sie enthaltenden Nachrichtenwarteschlange (eine „`XList<InterActionAbstract>`“) automatisch entfernt wird (siehe Abschnitt 3.3.4 und Anhang A.5).

3.4.1.3 Klasse „`DelayedTrans`“

Transitionsklassen zu Transitionen, die eine `DELAY`-Klausel besitzen, haben als Basisklasse `DelayedTrans`, welche ebenfalls eine Subklasse der oben beschriebenen Klasse `SimpleTrans` ist (siehe Abb. 3-13 auf Seite 66).

Beispiel 3.21-a: Auszug aus der Klasse `DelayedTrans` (`xecrt_modules.h`)

```
1: class DelayedTrans : public SimpleTrans {
2:     private:
3:         // ...
4:         Timer timer;
5:         int getDelay();
6:     protected:
```

```
 7:          virtual int getDelay1() = 0;
 8:          virtual int getDelay2() = 0;
 9:      public:
10:          // ...
11:          bool test();              /* override */
12:          void fire();              /* override */
13: };
```

(Ende von Beispiel 3.21-a)

Zu diesem Zweck enthält sie eine Komponente „`Timer timer`“ (Zeile 4 von Beispiel 3.21-a), welche als Schnittstelle zu den im Modul `xecrt_context` plattformspezifisch implementierten Zeitmesssystemen dient. Die Klasse `Timer` (siehe Beispiel 3.21-b) stellt dabei Countdown-Timer bereit, die gestartet mit einem (zunächst semantisch nicht näher spezifizierten) Zeit-Parameter[98] (durch „`start(t)`“, siehe Zeile 6) vom initialen Zustand *STOPPED* in den Zustand *RUNNING* wechseln, wo sie für den angegebenen Zeitraum verbleiben, um dann schließlich in den Zustand *EXPIRED* zu wechseln. Zur Abfrage des Erreichens der gleichnamigen Zustände existieren die Methoden „`stopped()`“, „`running()`“ und „`expired()`“(Zeilen 8 und 9). Von jedem Zustand aus kann jederzeit durch Aufruf der Methode „`stop()`“ zurück in den Zustand *STOPPED* gewechselt werden. Es ergibt sich damit der in Abb. 3-14 dargestellte Zustandsgraf.[99]

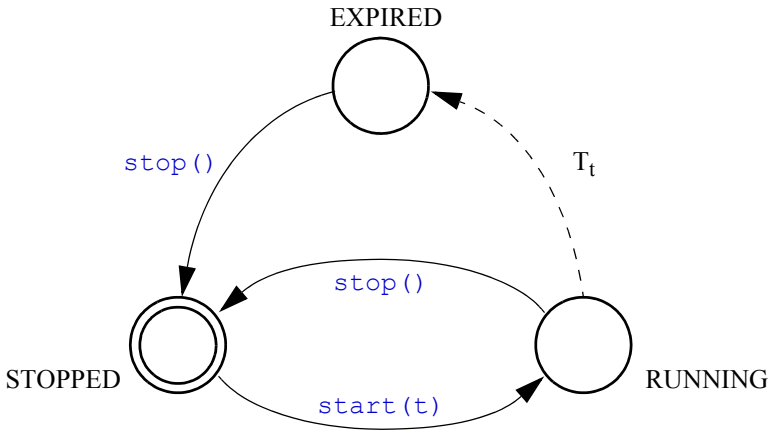


Abbildung 3-14: Zustandsgraf der Instanzen von `Timer` (vereinfacht)

Beispiel 3.21-b: Auszug aus der Klasse `Timer` (`xecrt_context.h`)

```
1: class Timer : public XListNode<Timer> {
2:     public:
           // ....
3:         typedef int Span;
4:         static const Span INFINIT_SPAN  = INT_MAX;
5:
```

98. Intern werden gegenwärtig alle Zeitangaben auf die Einheit Millisekunden bezogen.

99. Neben der Abfrage des Timers durch wiederholtes Zustandsabfragen (*Polling*) ist auch ein Überschreiben der virtuellen Methode „`Timer::handle_expiration()`“ möglich, welche beim Übergang vom Zustand *RUNNING* zum Zustand *EXPIRED* vom Timer-Managementsystem automatisch aufgerufen wird. Wir werden dies später im Zusammenhang mit einigen Optimierungsverfahren einsetzen (siehe Abschnitt 4.2.2).

```
 6:         void  start(Span nDelay);
 7:         void  stop();
 8:         bool  stopped() const;
 9:         bool  running() const;
10:         bool  expired() const;
11:     protected:
12:         virtual void handle_expiration() {};
13: };
```

(Ende von Beispiel 3.21-b)

Als Schnittstelle zu dem für die DELAY-Klausel in der Transitionsklasse generierten Code stehen die beiden abstrakten virtuellen Funktionen „getDelay1()“ und „getDelay2()“ (siehe Zeilen 7 und 8 in Beispiel 3.21-a) bereit, die bei ihrem Aufruf die Auswertungsergebnisse der jeweiligen Parameter der DELAY-Klausel zurückliefern (siehe Abschnitt 3.3.7.7).

Aus den Ergebnissen dieser beiden Methodenaufrufe berechnet die Methode „getDelay()“ eine konkrete Delay-Dauer, indem sie zunächst die Anforderung der Estelle-Semantik prüft, dass der Wert des ersten Parameters nicht größer als der des zweiten Parameters sein darf[100] und im Falle einer Verletzung dieser Bedingung den Wert -1 zur Invalidierung der DELAY-Klausel zurückliefert (Zeile 4 von Beispiel 3.21-c).[101]

Andernfalls liefert sie mit dem Wert des ersten Parameters den kleinsten möglichen Delay-Wert aus dem durch die Estelle-Semantik indeterministisch vorgegebenen Intervall vom Wert des ersten Parameters bis zu dem des zweiten Parameters.[102]

Der eigentliche Test der Transition auf Schaltbereitschaft[103] erfolgt auch in der Klasse DelayedTrans durch Überschreiben der Methode „test()“. Dabei wird gemäß der Estelle-Semantik aufbauend auf der o. g. Timer-Komponente ein Zustandsautomat für die Transition implementiert, der letztlich folgende Semantik hat: Die DELAY-Klausel ist erfüllt, wenn die Transition mindestens für den in der DELAY-Klausel angegebenen Zeitraum (s. o.) ununterbrochen[104] bereit war (d.h. FROM- und PROVIDED-Klausel erfüllt waren) und nicht geschaltet wurde.

Dazu werden zunächst diese beiden Klauseln durch Aufruf der ursprünglichen Methode „SimpleTrans::test()“ evaluiert (Zeile 10 von Beispiel 3.21-c). Sind sie nicht erfüllt, so wird der Timer-Automat in den Zustand *STOPPED* versetzt (Zeile 12). Andernfalls wird unterschieden, ob die Transition gerade bereit wurde (also der Timer sich noch im Zustand STOPPED befindet, siehe Zeile 15) oder früher bereits gestartet wurde (Zeile 29). Im letzteren Fall ist die DELAY-Klausel genau dann erfüllt, wenn der Timer abgelaufen ist (Zeile 30). Wurde die

100. Siehe Abschnitt 9.6.4 von [ISO97].

101. Verzichtet man auf die Überprüfung dieser Anforderungen, so muss insgesamt nur einer der beiden Parameter-Ausdrücke evaluiert werden.

102. Da wir hier eine Implementierung von Estelle-Spezifikationen entwickeln, können alle auf semantischer Ebene indeterministischen Verhaltensweisen durch eine beliebige konkrete Verhaltensweise implementiert werden.

103. Im Gegensatz zur Begriffsbildung in Abschnitt 9.6.2 von [ISO97] beinhaltet der Test auf *Schaltbereitschaft* von DELAY-Transitionen hier bereits auch die Erfülltheit der DELAY-Klausel.

104. Wir werden später im Zusammenhang mit der transitionslokalen Auswahloptimierung noch näher darauf eingehen, zu welchen Zeitpunkten diese Bedingungen geprüft werden müssen und welchen Einfluss dies auf die Systemperformance hat (siehe Abschnitt 4.3.3).

Transition dagegen gerade bereit, so ist ausgehend von der von „`getDelay()`“ zurück gelieferten Zeitspanne eine Fallunterscheidung (*<0*, *=0*, *>0*) zu treffen, die im üblichen Fall einer positiven (*>0*) Wartezeit zum Starten des Timers führt (Zeile 25).

Beispiel 3.21-c: Auszug aus der Klasse `DelayedTrans` (`xecrt_modules.cc`)

```
 1: int DelayedTrans::getDelay() {
 2:         int nDelay1 = getDelay1();
 3:         int nDelay2 = getDelay2();
 4:         if (nDelay2 < nDelay1)
 5:                 return -1;
 6:         return nDelay1;       // from [nDelay1 .. nDelay2]
 7: }
 8:
 9: bool DelayedTrans::test() {
10:         bReady = SimpleTrans::test();
11:         if (!bReady) {
12:                 timer.stop();
13:                 return false;
14:         }
15:         else if (timer.stopped()) {
16:                 int nDelay = getDelay();
17:
18:                 if (nDelay == 0)
19:                         return true;
20:                 else if (nDelay < 0) {
21:                         bReady = false;
22:                         return false;
23:                 }
24:                 else {
25:                         timer.start(nDelay);
26:                         return false;
27:                 }
28:         }
29:         else
30:                 return timer.expired();
31: }
32:
33: void DelayedTrans::fire() {
34:         SimpleTrans::fire();
35:         timer.stop();
36: }
```

(Ende von Beispiel 3.21-c)

Das Schalten einer Transition mit `DELAY`-Klausel gestaltet sich dagegen wesentlich einfacher, da neben aus „`SimpleTrans::fire()`“ bekannten Operationen lediglich das Zurücksetzen des Timers erforderlich ist.[105]

105. Dies stellt sicher, dass bei fortgesetzt bereiten Transitionen im Verlauf des nächsten Auswahldurchlaufs der Timer erneut über den Mechanismus in Zeile 16 bis 27 gestartet wird.

3.4.2 Modullokale Transitionsauswahl

Die Aufgabe der modullokalen Transitionsauswahl besteht darin, aus der Menge aller Transitionsinstanzen einer gegebenen Modulinstanz eine *schaltbare Transition* auszuwählen oder nachzuweisen, dass keine solche Transition existiert. Wir werden später sehen, dass dieser Auswahlvorgang ein erhebliches Optimierungspotenzial besitzt (siehe Abschnitt 4.3). Zunächst beschäftigen wir uns jedoch mit einer *Referenzimplementierung* dieses Auswahlvorgangs.

In den vorangegangenen Abschnitten haben wir gesehen, wie anhand einer einzelnen Transition die Überprüfung der Schaltbereitschaft bezüglich der `FROM`-, `PROVIDED`-, `WHEN`- und `DELAY`-Klauseln durch Aufruf der virtuellen Methode „`test()`“ der Klasse `SimpleTrans` bzw. der von ihr abgeleiteten Klassen erfolgen kann. Entsprechend definiert sich über dieses Prädikat die Teilmenge der *schaltbereiten Transitionsinstanzen* einer Modulinstanz.

Die modullokale Transitionsauswahl wertet nun mit der `PRIORITY`-Klausel die letzte noch nicht berücksichtigte Transitionsklausel aus, indem sie aus der genannten Teilmenge eine Transition mit *maximaler Priorität* auswählt. Stehen mehrere Kandidaten gleicher (maximaler) Priorität bereit, so kann einer davon *beliebig* ausgewählt werden.[106]

Der modullokale Transitionsauswahlprozess hat weiterhin gemäß der Estelle-Semantik (siehe Abschnitte 9.6.4 und 9.6.5 von [ISO97]) den Seiteneffekt, dass für alle `DELAY`-Transitionsinstanzen eine eventuelle Änderung ihrer Schaltbereitschaft (bezüglich der `FROM`- und `PROVIDED`-Klauseln) durch einen Start bzw. Abbruch des mit der Transition assoziierten *Delay-Timers* berücksichtigt wird. Dies geschieht zum Beispiel gerade durch Aufruf der in Abschnitt 3.4.1.3 vorgestellten „`test()`“-Methode der Klasse `DelayedTrans`.

Als Referenzimplementierung dieser modullokalen Transitionsauswahl dient die in Beispiel 3.22-a dargestellte Methode „`basic_select(...)`“. Dabei wird die als Parameter übergebene Liste von Transitionen sequentiell durchlaufen, für jede Transition die Methode „`test()`“ aufgerufen[107], und bei positivem Ergebnis die Transition *vorläufig* ausgewählt (d. h. in `pSelected` abgelegt), wenn sie die erste solche Transition ist oder sie eine höhere Priorität als die zuletzt vorläufig ausgewählte Transition hat. Entsprechend referenziert `pSelected` nach Durchlaufen aller Transitionsinstanzen eine Referenz auf eine der gesuchten schaltbereiten Transitionen maximaler Priorität oder den Wert `NULL`, wenn keine solche Transition existiert.[108]

Beispiel 3.22-a: Auszug aus der Methode `basic_select` (`xecrt_modules.cc`)

```
1: SimpleTrans* /*...::*/ basic_select(XList<SimpleTrans>& lst) {
2:         SimpleTrans* pSelected = NULL;
3:         for (SimpleTrans* pTrans = lst.first() ;
                pTrans ;
                pTrans = pTrans->next()
           ) {
4:               if (pTrans->test() &&
5:                   ( (!pSelected) ||
```

106. Auf semantischer Ebene wird dies als *indeterministische* Auswahl formalisiert.

107. Der Aufruf von `test()` ist zur semantikkonformen Steuerung der Timer-Bedingungen bei allen `DELAY`-Transitionen zwingend erforderlich (siehe auch Abschnitt 4.3.3).

108. Diese Auswahlmethode ist zwischen den Transitionen gleicher Priorität offensichtlich nicht aus sich heraus *fair*, da immer die erste geeignete Transition aus der Liste gewählt wird.

```
 6:                       (pTrans->priority() < pSelected->priority())
 7:                   )
 8:               )
 9:                       pSelected = pTrans;
10:         }
11:         return pSelected;
12: }
```

(Ende von Beispiel 3.22-a)

Um den Auswahlprozess für spätere Optimierungen dynamisch flexibel gestalten zu können, wird zur modullokalen Transitionsauswahl keine feste Methode der Klasse `Module` eingesetzt, sondern eine von dieser referenzierte Instanz der Basisklasse `LocalTransSelector`. Diese definiert im Wesentlichen die virtuelle Funktion „`select()`“, über die durch Überschreiben in einer Spezialisierung von `LocalTransSelector` verschiedene Auswahlmethoden implementiert werden können.

Beispiel 3.22-b: Auszug aus der Klasse `LocalTransSelector` (`xecrt_modules.h`)

```
1: class LocalTransSelector {
2:         // ...
3:     public:
4:         virtual SimpleTrans* select();
5:         static SimpleTrans* basic_select(XList<SimpleTrans>& lst);
6: };
```

(Ende von Beispiel 3.22-b)

Die Basisklasse `LocalTransSelect` implementiert diese Methode im Sinne einer *Referenzimplementierung* durch Aufruf der oben genannten statischen Methode „`basic_select(...)`“. Darüber hinaus wird diese Methode auch zur Auswahl einer Initialisierungstransition durch Anwendung auf die Liste `Module::InitTransitions` eingesetzt (siehe Abschnitt 3.3.7.9).

Die Festlegung eines (Default-) Auswahlverfahrens erfolgt über die Kommandozeilenoption “`-localselect <name>`“ der generierten Implementierung, wobei mit “`-localselect straight`“ z. B. das beschriebene Referenzauswahlverfahren aktiviert wird. Wir werden später in Abschnitt 4.3 noch verschiedene andere Verfahren zur modullokalen Transitionsauswahl auf Basis der Klasse `LocalTransSelect` kennen lernen.

3.4.3 Globale Auswahl

Ziel der *globalen (Transitions- bzw. Modul-) Auswahl innerhalb eines (Sub-) Systems* ist es, auf Basis der Modulinstanz-Hierarchie und der oben beschriebenen lokalen Transitionsauswahl jeweils eine Menge schaltbarer Transitionen bzw. Module[109] *auszuwählen*, die dann im Wechsel mit der Auswahl *ausgeführt* werden. Die Auswahl innerhalb eines Subsystems erfolgt dabei durch rekursive modullokale Transitionsauswahl, wobei abhängig von der Modulattributierung

109. Die Begriffe „schaltbare Transition“ und „schaltbare Modul (-Instanz)“ fallen auf Ebene der globalen Auswahl letztlich zusammen, da jede Modulinstanz nur höchstens eine schaltbare Transition (auswählen und) anbieten kann und genau darüber sich die *Modulinstanz* als *schaltbar* qualifiziert.

immer nur höchstens ein Kindmodul ausgewählt wird (ACTIVITY) oder alle Kindmodule in die Auswahl miteinbezogen werden (PROCESS). Da jedes Modul lokal höchstens eine Transition anbieten kann, und zudem auf Grund der Vater-Sohn-Priorität bei Modulen mit einer schaltbaren Transition die oben genannte Rekursion abbricht, können nur durch PROCESS-attributierte Module mehrere Transitionen in einem Auswahl-Vorgang ausgewählt werden.

Eine Referenz-Implementierung dieses Auswahlvorgangs besteht also darin, für jedes Modul zunächst eine lokale Transitionsauswahl durchzuführen und bei deren erfolglosem Ausgang rekursiv die Suche auf die Kindmodule fortzusetzen. Dabei wird abhängig von der Modulattributierung beim ersten Kindmodul mit einer erfolgreichen Transitionsauswahl die rekursive Suche abgebrochen (ACTIVITY), oder es werden alle Kindmodule getestet (PROCESS). Im letzteren Fall ist das Ergebnis der Auswahl unter den Kindmodulen eine Teilmenge dieser Kindmodule, im ersteren Fall kann es höchstens ein Kindmodul sein.

Zur Darstellung des Auswahl-Zustandes zwischen dem globalen Auswahlvorgang (durch Methode „Module::select()“) und dem Schalten der Transitionen (durch Methode „Module::fire()“) dienen dabei die beiden Komponenten „SimpleTrans* pSelectedTrans“ (für eine lokal ausgewählte Transition) und die Liste SelectedChildren, in die ausgewählte Kindmodule vorübergehend aus der Kindmodul-Liste Children verlegt werden (siehe Zeile 11 von Beispiel 3.23). Ist das Vatermodul ACTIVITY-attributiert, so wird der Suchprozess beim ersten erfolgreich getesteten Kindmodul abgebrochen[110] (siehe Zeile 13), andernfalls werden alle Module getestet und gegebenenfalls in die Liste SelectedChildren verlegt. Beim rekursiven Schalten der ausgewählten Kindmodulinstanzen werden diese dann später wieder zurück in die Liste Children transferiert (siehe Zeile 34).

Beispiel 3.23: Auszug aus der Klasse Module (xecrt_modules.h)

```
 1: bool Module::select() {
 2:         // now do local transition selection ...
 3:         pSelectedTrans = pLocalTransSelector->select();
 4:         if (!pSelectedTrans) {
 5:                 /* evaluate childmodules */
 6:                 tracing.notifyStartChildSelect(this);
 7:                 Module *pNextChild;
 8:                 for ( Module* pChild = Children.first() ;
                          pChild ;
                          pChild = pNextChild
                   ) {
 9:                         pNextChild = pChild->next();
10:
11:                         if (pChild->select()) {
12:                                 SelectedChildren.append(pChild);
13:                                 if (!(uClass & PROCESS))
14:                                         // just the first one
15:                                         break;
16:                         }
17:                 }
18:         }
19:         return !pSelectedTrans && SelectedChildren.isEmpty();
```

110. Diese Auswahlmethode ist zwischen Kindmodulen ACTIVITY-attributierter Module offensichtlich nicht aus sich selbst heraus *fair*, da immer das erste geeignete Modul aus der Liste gewählt wird.

```
20: }
21:
22: void Module::fire() {
23:         // fire selected transitions or children
24:         if (pSelectedTrans) {
25:                 pSelectedTrans->fire();
26:                 pSelectedTrans = NULL;
27:         }
28:         else {
29:                 for ( Module* pChild = SelectedChildren.first() ;
30:                       pChild ;
31:                       pChild = SelectedChildren.first()
32:                 ) {
33:                         pChild->fire();
34:                         Children.insert(pChild);
35:                 }
36:         }
37: }
```

(Ende von Beispiel 3.23)

Die globale Auswahl *zwischen* Subsystemen erfolgt auf semantischer Ebene indeterministisch und damit auf Implementierungsebene beliebig. Eine einfache Standardimplementierungsmethode für sequentielle Implementierungen besteht darin, den beschriebenen globalen Auswahlprozess unabhängig von der konkreten Attributierung ebenfalls rekursiv vom Spezifikationsmodul aus zu betreiben. Da Subsysteme semantisch nebenläufig arbeiten, kann dabei die Auswahl oberhalb der Subsystem-Ebene (also für nicht attributierte Modulinstanzen) z. B. auch analog zu einer hinzu gedachten `PROCESS`- oder `ACTIVITY`-Attributierung erfolgen.

In der Tat bietet die o. g. Referenzimplementierung auch für nicht attributierte Module eine korrekte Modulauswahl: Da bei nicht attributierten Modulinstanzen die Komponente `uClass` das Bit-Flag `PROCESS` nicht enthält, erfolgt zwischen Subsystemen bei der oben beschriebenen Referenzauswahlmethode eine Synchronisierung gemäß einer `ACTIVITY`-Attributierung.[111]

Wir werden später anhand komplexerer Verfahren zeigen, dass der globale Auswahlvorgang ein erhebliches Optimierungspotenzial besitzt (siehe Abschnitt 4.2).

3.4.4 Optimierungsparameter aus der Codegenerierung

Wir haben in den vorangegangenen Abschnitten das Zusammenwirken des *generierten Codes* als Spiegelbild der in der Spezifikation vorgefundenen Definitionen und der *vordefinierten Laufzeitbibliothek* zur Steuerung der Ausführung des spezifizierten Systems auf Basis des generierten Codes kennen gelernt. Durch die damit verbundene konsequente Übertragung von Ausführungs- und Optimierungsmechanismen in die Laufzeitbibliothek wurden sowohl der Codegenerator als auch die Implementierung von Optimierungsmechanismen wesentlich vereinfacht. Dies wird jedoch erkauft durch eine gewisse Unflexibilität dieser Mechanismen bezüglich einer Anpassung an die spezifischen Gegebenheiten in der jeweiligen Spezifikation. So ab-

111. Die *lokale* Transitionsauswahl verläuft bei diesen nicht attributierten Modulinstanzen natürlich immer erfolglos, da diese (abgesehen von Initialisierungstransitionen) keine Transitionen enthalten können.

strahiert das Interface zwischen generiertem Code und Laufzeitbibliothek z. B. bzgl. des Vorhandenseins von bestimmten Transitionsklauseln fast vollständig[112], sodass ihre Auswertung in der Laufzeitbibliothek nach einem festen Schema erfolgen kann (siehe Abschnitt 3.4.1).

Einige Optimierungsmaßnahmen erfordern jedoch die Kenntnis *spezifischer Eigenschaften der Spezifikation*, die über dieses einfache und abstrakte Interface hinausgehen. Um derartige Zusatzinformationen, die für die korrekte Implementierung im Grunde nicht erforderlich sind, übertragen zu können, wurde das Interface zwischen generiertem Code und Laufzeitbibliothek an einigen Stellen erweitert. Ein Beispiel für diese Erweiterungen sind die sog. *Hints*.

Hints sind Attribute, die vom Codegenerator den Klassendefinitionen für konkrete Module und Transitionen zugeordnet und als Konstruktor-Parameter[113] an die jeweiligen Basisklassen der Laufzeitbibliothek übergeben werden. Diese Attribute zeigen jeweils eine spezielle Eigenschaft des Moduls bzw. der Transition an und werden nach den jeweiligen Bedürfnissen der Laufzeitbibliothek in den Codegenerator integriert. Die Semantik dieser Attribute ist dabei jeweils so gewählt, dass ein nicht-Setzen eines Attributs in jedem Fall zu einer semantisch korrekten Implementierung führt. Umgekehrt zeigt das Setzen eines Attributs eine spezifische Eigenschaft der Spezifikation an, die von der Laufzeitbibliothek zur Optimierung bestimmter Mechanismen eingesetzt werden kann.

So haben wir z. B. in Abschnitt 3.3.7.6 mit der Methode „`getWhenIP()`“ eine wesentliche Komponente der Schnittstelle zwischen einer Eingabe-Transition und der Laufzeitbibliothek kennen gelernt (siehe auch Beispiel 3.14-b auf Seite 74). Diese Methode liefert für einen konkreten Systemzustand den jeweils von der Transition referenzierten Interaktionspunkt. Da dieser gemäß der jeweiligen Index-Werte möglicher Interaktionspunkt-Arrays über den Systemzustand variieren kann, muss diese Methode gemäß des ursprünglichen abstrakten Interfaces bei jedem Test der `WHEN`-Klausel erneut aufgerufen werden.

Dieser Methoden-Aufruf kann dabei mit erheblichem zeitlichem Aufwand verbunden sein, wenn zur Berechnung der Index-Werte gemäß der Estelle-Spezifikation komplexe Ausdrücke evaluiert werden müssen. Andererseits wird häufig von einer `WHEN`-Klausel immer auf den selben Interaktionspunkt Bezug genommen, sodass in der Laufzeitbibliothek nach einer erstmaligen Auswertung der o. g. Methode das Ergebnis zwischengespeichert und wieder verwendet werden kann. Dies ist genau dann der Fall, wenn

- es sich um einen einfachen Interaktionspunkt handelt (kein Array) oder
- alle Index-Ausdrücke beim Zugriff auf ein Interaktionspunkt-Array bzgl. des jeweiligen Transitions-Objekts konstant sind.

Letzteres ist auf Grund der gewählten Implementierungsmethode der `ANY`-Klausel durch mehrfach-Instanziierung der jeweiligen Transitionsklasse insbesondere für solche Index-Ausdrücke der Fall, die nur von `ANY`-Parametern abhängen. Damit ist in der Praxis in typischen Protokollspezifikationen sehr oft[114] einer dieser beiden Punkte erfüllt und somit eine entsprechende Optimierung der Laufzeitbibliothek möglich. Wie wir zudem später sehen werden, kann über die

112. Eine Ausnahme bilden die `WHEN`- und `DELAY`-Klauseln, die jeweils durch spezielle Transitions-Basisklassen implementiert werden.

113. als OR-verknüpfte Bitflags in „`unsigned uHints`“, siehe z. B. Beispiel 3.6-b auf Seite 48

114. z. B. in der untersuchten Estelle-Spezifikation von XTP 4.0 für alle Eingabe-Transitionen (siehe Tabelle 3.3)

Einsparung des reinen Aufrufs der o. g. Methode hinaus das Wissen um die Invarianz des referenzierten Interaktionspunkts auch zu einigen modellbasierten Optimierungen eingesetzt werden (siehe Abschnitt 4.3.6).

Falls der Codegenerator die Invarianz des referenzierten Interaktionspunkts im obigen Sinne *positiv festgestellt* hat[115], wird der in der Laufzeitbibliothek definierte Bitflag-Wert „`HINT_FIXED_IP`“ im oben genannten Konstruktor-Parameter der Eingabe-Transitionsklasse gesetzt.

Analog zu „`HINT_FIXED_IP`“ werden bei der Codegenerierung noch einige weitere Hints unterstützt, die verschiedene Eigenschaften von Transitionen oder Modulen (bzw. deren Modulheadern) anzeigen (siehe Tabelle 3.3). So zeigt z. B. „`HINT_SAMEDELAY`“ an, dass bei einer Transition mit `DELAY`-Klausel die beiden Parameter-Ausdrücke (bzgl. Schaltbarkeit und optionaler Schaltbarkeit) identisch sind, und „`HINT_NOEXPVAR`“ zeigt für einen Modulheader[116] an, dass er keine exportierten Variablen enthält. Wir werden später sehen, dass gerade letzteres eine wichtige Eigenschaft bzgl. modellbasierter Optimierungen ist.

Tabelle 3.3: Hints in XEC (V 1.3.2) und ihre Häufigkeit im XTP-Benchmark

Bezug	Name	Bedeutung	Häufigkeit[a]
Transition	`HINT_NOPROV`	keine `PROVIDED`-Klausel in einer Transition	61 von 184
	`HINT_SAMEDELAY`	beide Parameter einer `DELAY`-Klausel identisch	13 von 13
	`HINT_FIXED_IP`	invarianter IP-Bezug einer `WHEN`-Klausel	119 von 119
Modulheader[b]	`HINT_NOEXPVAR`	keine nach außen exportierten Variablen	4 von 10
	`HINT_NOEXPIP`	keine externen Interaktionspunkte	1 von 10
Modul	`HINT_NOINTVAR`	keine lokalen Variablen	7 von 10
	`HINT_NOINTIP`	keine lokalen Interaktionspunkte	8 von 10
	`HINT_NODELAY`	keine Delaytransitionen	8 von 10
	`HINT_NOPRIO`	alle Prioritäten identisch	6 von 10

a. Relative Häufigkeit des Auftretens in der XTP-Benchmarkspezifikation (Abschnitt 2.3.4.2) bezogen auf alle Module, Modulheader bzw. Transitionen (mit leeren Benutzer- und Medien-Modulen).
b. Einschließlich des implizit definierten Systemheaders.

115. Da im Gegensatz zum Setzen das nicht-Setzen eines Flags bzgl. einer korrekten Implementierung immer unschädlich ist, werden die Flags nur bei einem erfolgreichen (positiven) Nachweis der mit ihnen assoziierten Eigenschaft gesetzt. (Es ist dabei zu beachten, dass die effektive Ergebnis-Invarianz eines Estelle-Ausdrucks prinzipiell nicht berechenbar ist.)

116. und damit über den Vererbungsmechanismus der XEC-Implementierung auch für alle zugehörigen Module, siehe Abschnitt 3.3.2.1

In Tabelle 3.3 ist neben den Transitions-Hints auch der Prozentsatz ihres Auftretens in der XTP-Benchmarkspezifikation angegeben (siehe Abschnitt 2.3.4). Bemerkenswert ist dabei, dass „HINT_FIXED_IP“ und „HINT_SAMEDELAY“ bei allen Transitionen gesetzt sind. Bei den Modulheadern[117] exportieren nur 40% keine Variablen („HINT_NOEXPVAR“), aber bis auf das Systemmodul selbst haben alle Module externe Interaktionspunkte. Weiterhin haben nur 20% der Module DELAY-Transitionen (nicht „HINT_NODELAY“) und 60% der Module haben Transitionen mit einheitlichen[118] Prioritäten („HINT_NOPRIO“). Wir werden später in Kapitel 4 auf die Anwendungen dieser Informationen bei der dynamischen Optimierung zurückkommen.

117. In der Spezifikation besitzt jede Moduldefinition ihre eigene Headerdefinition.

118. genauer: keiner expliziten

3.5. Systemmanagement und Plattformanbindung

Neben den bisher diskutierten Mechanismen zur Auswahl und Ausführung von Transitionen müssen zum Betrieb der generierten Implementierung noch einige zentrale Managementaufgaben erbracht werden. Diese beinhalten neben der Steuerung des Systemlebenszyklus und der darin iterierten globalen Auswahl- und Ausführungsphasen auch Aspekte der Plattformanbindung wie z. B. die Umsetzung des Estelle-Zeitmodells.

3.5.1 Systemsteuerung

Wie bereits in Abschnitt 3.2.1 dargestellt, wird die Ausführung der von XEC erzeugten Implementierung vollständig von Routinen des Laufzeitsystems initiiert und gesteuert. Ausgangspunkt ist die in Beispiel 3.24-a wiedergegebene Funktion `main()` des XEC-Laufzeitsystems.

Beispiel 3.24-a: Die `main`-Funktion des Laufzeitsystems (`xecrt_context.cc`)

```
 1: int main(int, char**argv) {
 2:         if (!scanArgs(argv))
 3:                 return 1;
 4:
 5:         Specification* pSpec = startup();
 6:         run(pSpec);
 7:         shutdown(pSpec); /
 8:
 9:         return 0;
10: }
```

(Ende von Beispiel 3.24-a)

Nach der Auswertung der Kommandozeilenargumente in der Funktion `scanArgs()` (siehe auch Anhang A.2.1) wird durch Aufruf der Funktion `startup()` eine Instanz des Systemmoduls erzeugt. Die wichtigste in dieser Funktion durchgeführte Aktion ist dabei der indirekte Aufruf der vom Codegenerator erzeugten Funktion `createSpecificationInstance()` (siehe Beispiel 3.24-b), die als Bindeglied zwischen der (spezifikationsunspezifischen) Laufzeitbibliothek und dem aus der Estelle-Spezifikation erzeugten Klassensystem fungiert, indem sie eine explizite Bezugnahme auf die konkrete Spezifikationsklasse[119] (im Beispiel `SPEC_xtp40`) enthält. Sie liefert letztlich eine neu erzeugte Instanz dieser Spezifikationsklasse (als Spezialisierung der Klasse `Specification`, siehe Abb. 3-11 auf Seite 52) zurück.[120]

119. Nicht zuletzt die Erhaltung des Spezifikationsnamens als Basis des generierten Klassennamens macht diese Konstruktion erforderlich, da von der spezifikationsunspezifischen Laufzeitbibliothek nur ein normierter Funktionsname aufgerufen werden kann. Sie erlaubt jedoch mit geringfügigen Modifikationen auch eine Instanziierung verschiedener Spezifikationen im selben Prozess (siehe auch Abschnitt 7.5 zur Implementierung von Open-Estelle).

120. Auf die Funktion der `staticData`-Komponente kommen wir in Abschnitt 3.5.4 zurück.

Beispiel 3.24-b: Erzeugung einer Spezifikationsinstanz (hier für `xtp40`) im generierten Code

```
1: Specification* createSpecificationInstance() {
2:         return SPEC_xtp40::staticData.newInstance();
3: }
```

(Ende von Beispiel 3.24-b)

Die Ausführung der Spezifikation wird dann von der Funktion `run()` gesteuert, bis im Falle der Termination des implementierten Systems[121] durch Aufruf der Funktion `shutdown()` noch einige Aufräumarbeiten (die Ausgabe von Statistiken, etc.) ausgeführt werden und schließlich das Programm terminiert.

Die schrittweise Ausführung der Implementierung wird durch wiederholten Aufruf der Funktion `step()` realisiert (Zeile 7-8 in Beispiel 3.24-c). Jeder dieser Aufrufe realisiert jeweils einen Zyklus bestehend aus einer systemweiten Transitionsauswahl (Zeile 18) auf die im Erfolgsfall die Ausführungen der ausgewählten Transitionen (Zeile 22) zusammen mit dem zugehörigen Transport der dabei erzeugten Interaktionen (Zeile 23) folgt.

Beispiel 3.24-c: Schrittweise Ausführung der Implementierung durch `run()` und `step()`

```
 1: static void run(Specification* pSpec) {
 2:         tracing_int.notifySystemInit(pSpec);
 3:         timestatExec.start();
 4:         pSpec->execInitTrans();
 5:         Specification::forwardInteractions();
 6:
 7:         while (!tracing_int.aborted() &&
                   !SignalHdlr::testSigInt()
            )
 8:                 step(pSpec);
 9:
10:         tracing_int.notifySystemTermination(pSpec);
11: }
12:
13: static inline void step(Specification* pSpec) {
14:         tracing_int.notifyNewCycle(pSpec);
15:         timestatEval.start();
16:         Timer::update();
17:
18:         bool bFireable = pSpec->select();
19:
20:         if (bFireable) {
21:                 timestatExec.start();
22:                 pSpec->fire();
23:                 Specification::forwardInteractions();
```

121. Die Estelle-Semantik sieht keine Systemtermination im eigentlichen Sinne vor.
Es kann jedoch in Berechnungen Systemzustände geben, in denen kein neuer nächster Zustand im Sinne der *next-state*-Relation existiert, da keine Transition mehr ausgewählt oder gefeuert werden kann und auch keine Timer mehr laufen. Darüberhinaus ist auch eine programmierte oder durch externe Signale (z. B. durch Drücken der Tastenkombination „`Ctrl-C`“, siehe `SignalHdlr::testSigInt()`) initiierte explizite Termination möglich, die vom Laufzeitsystem abgefangen und synchron behandelt wird. (s. u.).

```
24:         }
25:         else {
26:                 timestatSleep.start();
27:                 Timer::sleep();
28:         }
29: }
```

(Ende von Beispiel 3.24-c)

Neben diesen Kernoperationen werden während der Ausführung auch diverse weitere Managementaufgaben durchgeführt. Diese dienen teilweise direkt der Spezifikationsausführung (wie z. B. die `Timer`-Steuerung), teilweise dienen sie jedoch auch nur der Unterstützung von Auswertefunktionen wie dem Tracing (`tracing`...) und der Ausführungszeit-Statistik (`timestat`...). In den folgenden Abschnitten geben wir einen kurzen Überblick über diese Managementaufgaben.

3.5.2 Zeitbegriffe auf Spezifikations- und Implementierungsebene

Das Zeitmodell von Estelle (Abschnitt 5.3.5 von [ISO97]) basiert auf der Vorstellung eines „*Zeitprozesses*“ („*time process*“), der das Voranschreiten der Zeit nebenläufig zur Ausführung der Spezifikation modelliert. Dazu ist jeder `DELAY`-Klausel in der Semantik ein *Delay-Timer* zugeordnet, der von seiner Aktivierung an ausgehend von einem Startwert durch den o. g. Zeitprozess dekrementiert wird, bis er schließlich 0 erreicht (siehe Abschnitt 9.6.4 von [ISO97]). Dabei werden keine konkreten Annahmen über die Schnelligkeit des Voranschreitens der Zeit bzw. die Ausführungsdauer von Operationen gemacht, was sich nicht zuletzt darin widerspiegelt, dass die in der `Timescale` angegebene Zeiteinheit der Delay-Parameter lediglich Kommentarcharakter hat. Es wird lediglich gefordert, dass die Zeit überhaupt (im Verlauf der Berechnung) *voranschreitet* und dieses Voranschreiten für alle Systemkomponenten (genauer: alle Delay-Timer der `DELAY`-Transitionen) *einheitlich* erfolgt. Entsprechend hat eine Implementierung einer Estelle-Spezifikation große Freiheiten bei der Modellierung von Zeit.

Die Grundlage der Zeitmodellierung im XEC-Laufzeitsystem bildet die in Abschnitt 3.4.1.3 bereits auszugsweise vorgestellten Klasse `Timer`. Diese realisiert die oben beschriebenen Estelle-Timer als internen Aspekt der Plattformanbindung.[122] Zur Anbindung an die `DELAY`-Transitionen enthält jede Instanz der zu ihrer Implementierung eingesetzten Klasse `DelayedTrans` mit der Komponente `timer` eine Instanz[123] der Klasse `Timer` (siehe UML-Diagramm in Abb. 3-15), über die sie in abstrakter Form ihr Timeout-Verhalten modellieren kann.

Die interne Realisierung der Klasse `Timer` basiert in der gegenwärtigen Fassung der XEC-Laufzeitbibliothek nicht auf einem zu dekrementierenden Restwertzähler, sondern auf einem Ablauf-Zeitstempel[124] (`Timer::tExpire`), der den absoluten Zeitpunkt des Ablaufens des

122. Quelldatei „`xecrt_context.cc/h`“

123. über den Zwischenschritt der Klasse `DelayTimer` spezialisiert

124. Um eine ausreichende Laufzeit ohne Überläufe zu ermöglichen, wird der dazu genutzte Typ `Timer:Stamp` als 64-Bit-Integer (`long long`) definiert. Ein 32-Bit-Zähler würde bei Mikrosekunden-Auflösung sonst bereits nach ca. 35 Minuten (bzw. 70 Minuten als vorzeichenloser Typ) überlaufen. Der 64-Bit-Typ läuft dagegen selbst bei Nanosekunden-Auflösung erst nach über 250 Jahren über.

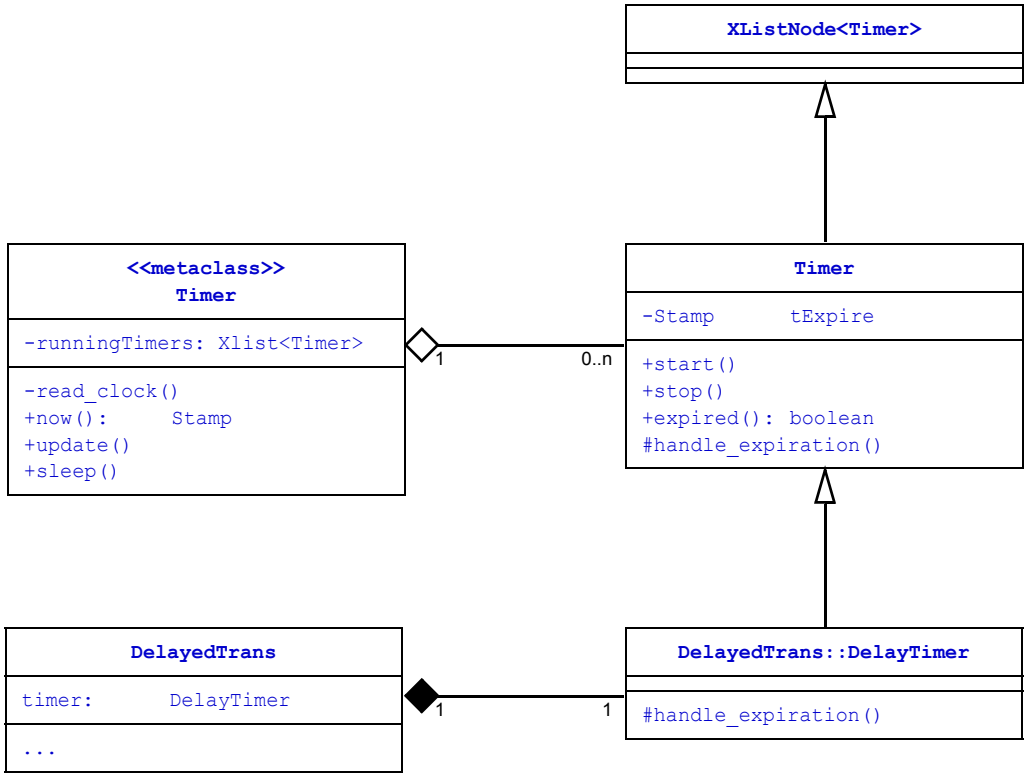


Abbildung 3-15: UML-Diagramm der Implementierung der Delay-Timer

Timers in Relation zu einer zentralen Uhr bewertet, die von einem Startwert 0 aus monoton ansteigt. Der Timer gilt als abgelaufen, wenn die zentrale Uhr diesen Zeitstempel erreicht oder übertrifft. Entsprechend wird der Timer gestartet, indem der Ablauf-Zeitstempel auf den aktuellen Wert der Uhr zuzüglich des gewünschten Timeouts gesetzt wird.

Zur Überwachung der laufenden Timer enthält die Metaklasse von `Timer` eine Liste `runningTimers`, in der alle laufenden Timer-Instanzen enthalten[125] sind. Entsprechend werden im Vorfeld der globalen Transitionsauswahl (siehe `Timer::update()` in Zeile 16 von Beispiel 3.24-c) über diese Liste die abgelaufenen Delay-Timer vorab ermittelt, die betroffenen Timer werden aus der Liste entfernt und die entsprechenden `DELAY`-Klauseln werden als erfüllt markiert. Zur Optimierung dieses Vorgangs wird die Liste `runningTimers` bzgl. der Ablaufzeitstempel aufsteigend sortiert geführt, so dass die Suche nur bis zum ersten nicht abgelaufenen Timer erfolgen muss.[126]

Diese zentrale Verwaltung der neu abgelaufenen Timer (also der möglicherweise dadurch schaltbar gewordenen `DELAY`-Transitionen) wird später insbesondere eine wichtige Grundlage für die ereignisgesteuerten Auswahloptimierungen in Kapitel 4 bilden.[127]

125. Dazu ist die Klasse `Timer` als potentieller Knoten einer solchen Liste von der Template-Basisklassen `XListNote<Timer>` abgeleitet (siehe auch Abschnitt 3.3.1).

126. Die Wirksamkeit dieser Optimierung beruht auf der Annahme, dass die Delay-Timer häufiger getestet als aktiviert und deaktiviert werden.

127. Als Schnittstelle zur Ereignissteuerung dient die von der abgeleiteten Klasse `DelayedTrans` später entsprechend überschriebene virtuelle Methode `Timer::handle_expiration()`.

Zur Realisierung der eigentlichen Uhr dient die abstrakte Basislasse `InternalClock`, deren virtuelle Methode `now()` [128] über die Schnittstellenfunktion `Timer::read_clock()` aufgerufen wird. Dazu können verschiedene Spezialisierungen dieser Basisklasse definiert werden, die durch Überschreiben dieser Methode entsprechend unterschiedliche Zeitmodelle realisieren.[129]

Diese Uhrenklassen realisieren dabei auch Methoden zur Behandlung von Systemzuständen, in denen keine schaltbare Transition gefunden werden kann, aber noch Delay-Timer laufen. In diesen Fällen kann das System bis zum Ablaufen des nächsten Timers als *inaktiv* betrachtet werden. Entsprechend kann die Implementierung zur Vermeidung von unnötigen (weil erfolglosen)[130] Auswahlzyklen in dieser Zeitspanne gemäß dem aktiven Zeitmodell „schlafen“ (s. u.). Dazu werden solche Situationen den konkreten Uhren über die virtuelle Methode `InternalClock::sleepUntil(InternalTimeStamp)` mitgeteilt.[131]

XEC realisiert zwei konkrete Zeitmodelle (als Spezialisierungen von `InternalClock`), die wir in den folgenden Abschnitten vorstellen.

3.5.2.1 Echtzeitmodell

Die Spezialisierung `InternalRTClock` („*real time clock*“) bezieht die Zeitmessung direkt auf einen plattformspezifischen Zugang zu einer Echtzeituhr, die als Zeitmodell die reale Ausführungsdauer der jeweiligen Systemoperationen (u. a. Auswahl- und Ausführungszyklen) zu Grunde legt. Der Zugriff auf die Echtzeituhr erfolgt über die in Abschnitt 3.6.4 vorgestellten plattformspezifischen Zeitmessfunktionen.

Dies bedeutet, dass für die unterstützten Timescale-Einheiten (siehe Abschnitt 3.3.7.7) ein direkter Bezug auf die reale Ausführungszeit besteht. Dies ist besonders sinnvoll, wenn unter Kommunikation mit externen (z. B. handimplementierten) Systemen realzeitbezogenen Timings realisiert werden sollen.

Diese Uhr wird durch den Kommandozeilenparameter “`-treal [maxsleep]`“ aktiviert.[132] Der optionale Parameter gibt an, wie im Falle eines Aufrufs der oben beschriebenen Methode `sleepUntil(...)` verfahren werden soll.

---

128. Der Aufruf dieser Methode kann über `Timer::now()` auch aus `PRIMITIVE`-Funktionen heraus erfolgen. Dadurch kann der Spezifikation direkter Zugriff auf das Zeitmodell gewährt werden. Dies ist je nach verwendetem Zeitmodell zur Erhaltung eines einheitlichen Zeitmodells dem direkten Aufruf einer plattformspezifischen Echtzeituhr vorzuziehen (siehe auch Abschnitt 4.2.2.7)

129. Die aktive Instanz der aktiven Uhr wird dabei über die Metaklassenmethode `InternalClock::clock()` ermittelt.

130. Wir gehen hier davon aus, dass nach einer erfolglosen Auswahl bis zum Eintreten eines potentiell aktivierenden Ereignisses (z. B. das Ablaufen eines Delay-Timers) auch wiederholte Ereignisse erfolglos sind oder zumindest (bei indeterministischer Auswahl) gemäß der Semantik erfolglos sein können. Siehe dazu auch Abschnitt 4.2.2.7.

131. In diesen Mechanismus ist auch die *Deadlockerkennung* integriert, die einen Blockadezustand ohne noch laufende Timer detektieren kann und ggf. das System (in Abhängigkeit von der Kommandozeilenoption “`-[no]deadlock`“) terminiert oder weiter pollt.

132. Sie wird auch verwendet, wenn keine andere Uhr ausgewählt wird.

(i) *Kein Parameter:* Das Laufzeitsystem blockiert den Prozess (unter UNIX durch Aufruf der Betriebssystemfunktion `sigaction(...)`[133] unter Einsatz eines entsprechenden Timeouts) bis zum geplanten Ablauf des Timers.

(ii) *Zahlenwert > 0:* Wie *(i)*, jedoch wird die Blockade spätestens nach der angegebenen Zeit (interpretiert in Mikrosekunden) beendet. Dies bedeutet eine abgeschwächte Form des Pollings.

(iii) *Zahl 0:* keine Blockade, die Methode `sleepUntil(...)` kehrt sofort zurück. Dies bedeutet echtes Polling.

(iv) „`SIMSLEEP`“: Anstatt die Wartezeit (pollend oder nicht pollend) abzuwarten, wird die echtzeitbasierte Uhr einfach auf die Zielzeit *vorgestellt*. Dadurch kehrt die Methode `sleepUntil(...)` sofort zurück, es findet jedoch kein Polling statt, da ja der nächste Timer dann sofort abgelaufen ist.

Gerade der letzte Punkt bildet eine interessante Mischung aus *Realzeitbezug* und *Zeitsimulation*: Die Bewertung der Ausführungsdauer der Systemoperationen basiert auf ihrem tatsächlichen Zeitbedarf, man vermeidet jedoch bei der Erprobung von (geschlossenen) Implementierungen Blockadephasen.

Im nächsten Abschnitt gehen wir in dieser Hinsicht einen Schritt weiter und geben den Realzeitbezug völlig auf.

3.5.2.2 Zeitsimulation

Die Spezialisierung `InternalSimClock` („*simulated clock*“) simuliert lediglich eine Uhr, indem sie feste Zeitkomponenten für bestimmte Operationen aufaddiert. Dazu werden auf Basis des in Abschnitt 3.6.3 vorgestellten Tracing-Systems bei bestimmten Ereignissen während der Ausführung der Implementierung zugeordnete Zeitwerte der simulierten Uhrzeit zugeschlagen.

Durch den Kommandozeilenparameter “`-tsim [fire [test]]`“ wird diese Uhr aktiviert. Die beiden optionalen Parameter geben an, mit welcher Ausführungszeit (in Mikrosekunden) das Feuern (Default: 10,0 μs) oder das Testen (Default: 0,5 μs) einer Transition gewichtet werden soll.

Die Behandlung von „Blockadephasen“ erfolgt in der Zeitsimulation natürlich analog zu der `Simsleep`-Option der Echtzeituhr durch *Vorstellen* der simulierten Uhr auf den übergebenen Zeitpunkt des nächsten Ablaufens eines Timers.

Der besondere Vorteil dieser simulierten Uhr ist, dass sie die Ausführung einer Spezifikation vollständig deterministisch[134] macht, da sich keine Schwankungen der Ausführungszeiten auf das System auswirken können. Dies ist von Vorteil, wenn für Analysen ein konkretes Szenario

133. Der Vorteil dieser Funktion gegenüber `sleep(...)` ist ihre bessere Granularität (Mikrosekunden statt Sekunden). Zudem können hier bei einer Anbindung an externe Kommunikationssysteme asynchrone Ereignisse (z. B. der Empfang einer TCP-Nachricht) referenziert werden, so dass die Blockade vorzeitig beendet werden kann.

134. Die erzeugte Implementierung ist abseits des Zeitaspekts bereits deterministisch. Variationen des Zeitmodells könnten jedoch (z. B. durch das Ablaufen oder nicht-Ablaufen eines Timeouts) Auswirkungen auf die ausgewählten Transitionen und damit die Zustandsraumentwicklung haben.

wiederholt ausgeführt werden soll. Besondere Bedeutung hat dies auch beim Debuggen der Implementierung auf Maschinenebene oder mit dem in XEC integrierten Estelle-Debugger (Abschnitt 3.6.5).

Einige Sonderfälle der Zeitparameter sind dabei besonders interessant:

(i) *test=0*: In diesem Fall hat die Anzahl der getesteten Transitionen keinen Einfluss auf das Zeitmodell. Dies kann bei jeweils identischen Auswahlergebnissen zur Steigerung der Vergleichbarkeit verschiedener Auswahlmodelle genutzt werden (siehe Kapitel 4).

(ii) *test=0 und fire=0:* Hier vergeht während der Ausführung des Systems zunächst überhaupt keine Zeit mehr.[135] Stattdessen wird nur noch in Blockadezuständen, in denen keine sofort schaltbare Transition mehr vorliegt durch den oben beschriebenen Mechanismus die Uhr auf den nächsten Zeitpunkt vorgestellt, an dem ein Delay-Timer abläuft.[136] Treten keine solchen Blockaden auf (also ist immer mindestens eine Transition schaltbar), so läuft kein Delay-Timer mehr ab.

3.5.3 Software-Plattformanbindung

Um die Grenzen der Ausdrucksfähigkeit von Estelle überwinden zu können, wurde die Möglichkeit zur Definition von `PRIMITIVE`- (bzw. `EXTERNAL`-) Funktionen und Prozeduren vorgesehen. Solche primitiven Funktionen und Prozeduren (und damit auch die sie enthaltende Spezifikation) haben keine Semantik, solange keine „*rigorose, implementierungsunabhängige (also mathematische) Definition des relevanten Blocks durch den Spezifizierer angegeben wird*“ (Abschnitt 8.2.4.3 von [ISO97]).

Bei der Gewinnung einer Implementierung müssen solche primitiven Funktionen und Prozeduren bei der Codegenerierung dem System hinzugefügt werden. XEC generiert dazu Prototyp-Methoden, die ggf. nachbearbeitet oder als Vorlage für separate Übersetzungseinheiten übernommen werden können (siehe auch Anhang A.8). Eine Möglichkeit zum Hinzufügen von Code für solche Funktionen ist die Auslagerung in eine getrennte Übersetzungseinheit, deren Implementierung dann beim Binden der automatisch generierten Implementierung hinzugebunden werden kann.

Alternativ kann auch C++-Code direkt als *Inline-Code* von der Spezifikation in den generierten Code übernommen werden. Dazu werden (in Anlehnung an die Realisierung in *EC*) Pseudo-kommentare der Form „`{$C$ ... }`“ in den Estelle-Spezifikationstext integriert.[137] Leider

135. Dieses extreme Zeitmodell ist i. A. nicht konform zur Estelle-Semantik (s. o.).

136. Dies hat interessanterweise gewisse Ähnlichkeit mit dem Zeitmodell diskreter Ereignisse in VHDL („*Very High Speed Integrated Circuit Hardware Description Language*“, [Hei+00]).

137. Leider ist das Kommentar-Endezeichen “`}`“ auch ein in C++ zulässiges (und kaum zu umgehendes) Syntaxelement. Die Verwendung von “`(* ... *)`“ als Kommentarklammern in Estelle löst das Problem nicht, da die Kommentarbegrenzer beliebig austauschbar sind und somit auch hier jedes “`}`“-Zeichen im Inline-Code den (Pseudo-) Kommentar (vorzeitig) beendet.
Einen gangbaren (wenn auch etwas unübersichtlichen) Lösungsansatz bieten die *Trigraph-Sequenzen* “`??<`“ und “`??>`“ als Ersatz für “`{`“ und “`}`“ im C++-Code [ISO98]. Günstiger erscheint hier jedoch die Auslagerung der Inline-Codesequenzen in separate Dateien und ihr textueller Import mit einem Include-Statement (`#include` ...). Dieses Include-Statement wird dabei (als Estelle-Kommentar) nicht vom PET-Include-Mechanismus (siehe Anhang A.1) sondern erst vom C++-Präprozessor expandiert. Der importierte Code unterliegt somit nicht der beschriebenen Problematik.

muss solcher Inline-Code sehr eng auf die Kodierungsmechanismen des verwendeten Codegenerators zugeschnitten werden und ist somit sehr unportabel. An dieser Stelle erweist sich jedoch bei Beschränkung auf einen konkreten Implementierungsgenerator ein deterministisches Kodierungsschema wie das von XEC als Vorteil (siehe Abschnitt 3.3), da so die Namen der generierten Definitionen stabil bleiben.

3.5.4 Modul-Metaklassen

In den Abschnitten 3.2.2 und 3.3.2 haben wir gesehen, wie XEC Estelle-Moduldefinitionen und die daraus zur Laufzeit erzeugten Modulinstanzen durch ein hierarchisches System von C++-Klassen (den Modulklassen) und deren Instanzen implementiert.

Um neben Implementierungsdaten, die *pro Modulinstanz* anfallen, auch *Instanz-übergreifende* Daten verwalten zu können, existiert parallel zum dynamischen System von Modulinstanzen auch noch ein statisches System von Objekten, die Verwaltungsinformationen zu jeweils einer Modulklasse bearbeiten und speichern und somit konzeptionell als Instanz der jeweiligen Metaklasse einer jeden Modulklasse fungieren.

Technisch werden diese jeweils als explizite statische Komponente „`staticData`“ der Template-Klasse „`StaticModuleData<class T>`“ in jeder Modulklasse angelegt, wobei durch die Parametrierung der Template-Klasse mit der umgebenden Modulklasse eine Spezialisierung auf diese erfolgt (siehe Beispiel 3.25).[138]

Beispiel 3.25: Modul-Metadaten in „`StaticModuleData`“

```
1: class SPEC_test : public Specification {
2:     public:
3:         // ...
4:         static StaticModuleData< SPEC_test > staticData;
5:         void terminate() {staticData.releaseInstance(this);}
6:         // ...
7: };
```

(Ende von Beispiel 3.25)

Die in der XEC-Laufzeitbibliothek definierte Template-Klasse `staticModuleData` dient zur Bereitstellung von *instanzübergreifenden Daten* zu Modulklassen. Dies umfasst neben Daten für das Monitoring und die dynamische Optimierung (siehe auch Profiling-Daten in Abschnitt 3.5.4.2) auch einen Modulinstanz-Cache, den wir im nächsten Abschnitt kurz vorstellen.

138. In C++ wird das Konzept der Metaklasse zu einer Klasse durch die statischen (d. h. mit „`static`“ attributierten) Methoden und Komponenten unterstützt, da diese klassen- und nicht objektbezogen instanziiert und zugegriffen werden. Insofern ist die statische Komponente „`staticData`“ einer Modulklasse eigentlich eine Komponente der (impliziten) Metaklasse zur Modulklasse.

3.5.4.1 Modulinstanz-Cache

Wie wir bereits in den Abschnitten 3.2.2.2 und 3.3.7 gesehen haben, implementiert XEC Transitionen innerhalb von Modulen nicht als passive Methoden, sondern als *aktive Objekte*. Diese Vorgehensweise bildet die Grundlage der adaptiv dynamischen Optimierungen und der dazu erforderlichen Erfassung statistischer Daten zur Laufzeit.

Der Nachteil dieser Implementierungstechnik besteht darin, dass bei der Instanziierung eines Moduls (also der C++-Modulklasse) jeweils auch alle erforderlichen Transitionsklassen-Instanzen neu erzeugt werden müssen. Dies, wie auch die Freigabe der Transitionsklassen-Instanzen beim Löschen einer Modulinstanz, kann bei einer großen Anzahl von Transitionen einen erheblichen zeitlichen Overhead verursachen. Dies gilt besonders vor dem Hintergrund, dass `ANY`-Klauseln von XEC expandiert und durch eine entsprechende Anzahl von Transitionsinstanzen implementiert werden.

Zur Kompensation dieses Nachteils wird bei der Freigabe einer Modulinstanz (implizit auf Grund der Freigabe der Vater-Modulinstanz oder explizit durch ein `RELEASE`- oder `TERMINATE`-Statement) nicht notwendigerweise auch die implementierende C++-Modulklassen-Instanz mit den zugehörigen Transitionsklassen-Instanzen freigegeben, sondern unter bestimmten Umständen für eine mögliche spätere Wiederverwendung in einem Modulinstanz-Cache aufbewahrt. Diese Deaktivierung und die Reaktivierung bei der späteren Wiederverwendung verlaufen dabei semantisch völlig transparent, und dienen ausschließlich dazu, die internen Strukturen der Modulinstanz-Implementierung nicht zu zerstören und ggf. später erneut vollständig anlegen zu müssen.

Semantisch relevante Aspekte der Modulinstanz-Repräsentation müssen dagegen vollständig nachgebildet werden. So werden zum Beispiel bei der Deaktivierung einer Modul-Klassen-Instanz alle Warteschlangen entleert und alle Kindmodulinstanzen rekursiv ebenfalls freigegeben, was wiederum ebenfalls möglicherweise durch eine Deaktivierung nach dem beschriebenen Muster erfolgt.

Die Aufbewahrung und Verwaltung der deaktivierten Modulklassen-Instanzen erfolgt dabei in der oben beschriebenen statischen Komponente `staticData` der jeweiligen Modulklasse. Bei der Erzeugung einer neuen Modulinstanz (implizit für das Spezifikationsmodul beim Systemstart oder explizit für alle anderen Module durch Aufruf eines `INIT`-Statements) werden entsprechend möglichst im Modul-Cache vorhandene deaktivierte Instanzen des selben Modultyps reaktiviert. Nur wenn dieser Cache leer ist, werden tatsächlich neue Instanzen erzeugt.

Die Implementierung dieses Mechanismus erfolgt durch die zwei Methoden `newInstance` und `releaseInstance` der oben genannten Modulklassen-Komponente `staticModuleData`, die die explizite Instanziierung beziehungsweise Termination von Modulklassen durch die Operatoren `new` und `delete` vollständig ersetzen. Dabei wird zur Vermeidung einer übermäßigen Speicherbelegung durch nicht mehr benötigte Modulinstanzen die maximale Anzahl von deaktivierten Modulinstanzen eines Typs im Cache auf einen parametrierbaren Wert beschränkt.

Der beschriebene Modulinstanz-Cache ist offensichtlich besonders effektiv in Systemen mit hoher *Strukturdynamik*, also einer wiederholten Instanziierung und Termination von Modulinstanzen beziehungsweise Modulinstanz-Bäumen. Dies ist zum Beispiel in der Estelle-Spezifikation von XTP 4.0 auf Grund der Modellierung der Kontextmodule, die die Endpunkte einer Verbindung darstellen, als Estelle-Modulhierarchie der Fall. Hier müssen bei einem Verbindungsaufbau in beiden XTP-Protokollmaschinen insgesamt zehn Modulinstanzen erzeugt und entsprechend bei einem Verbindungsabbau wieder aufgelöst werden (siehe Abb. 2-6 auf Seite 20). Bei

wiederkehrenden Verbindungsauf- und -abbau-Zyklen kann hier nach dem ersten Durchlauf eine Neuinstanziierung beziehungsweise Freigabe der eigentlichen Modul- und Transitionsinstanzen vollständig vermieden werden.

3.5.4.2 Ausführungsparameter und Profildaten

Neben laufzeitorientierten Aspekten wie z. B. dem vorgestellten Modulinstanz-Cache können die Modul-Metaklassen auch zur *persistenten Speicherung* von Optimierungs- und Ausführungsparametern für Modul- und Transitionsklassen dienen.

Dazu werden beim Start des Systems *Parameterdateien* gelesen und den dort namentlich referenzierten Modul-Metaklassen zugeordnet. Diese Parameterdateien können dann Eigenschaften der zugehörigen Module beeinflussen, die jenseits des von der Spezifikation formalisierten Verhaltens liegen.

Ein Beispiel für solche Parameter ist die sinnvolle Anzahl von Modulinstanzen eines Typs im Modulinstanz-Cache oder die Parametrierung von Monitoringmechanismen, die ansonsten als Annotationen im Spezifikationstext übergeben werden müssten.

Gerade die Gewinnung von Optimierungsparametern ist hier besonders interessant, da Eigenschaften einer Systemkomponente (z. B. die Wahrscheinlichkeit für *spontanes Schalten* von Transitionen in einem Modul, siehe Abschnitt 4.2.2.5) nicht erst während der Systemausführung dynamisch ermittelt werden müssten, sondern bereits beim Systemstart aus früheren Ausführungen (bzw. aus gezielten Profiling-Untersuchungen) heuristisch vorhergesagt werden könnten.

Die Gewinnung der Parameterdaten kann dabei grundsätzlich auf folgende Arten erfolgen:

- Speicherung der in den Modul-Metaklassen vorliegenden Daten durch selbige z. B. bei der Systemtermination.
  Dies bietet den Vorteil, dass das Format der Profildateien einschließlich der Namenshierarchie des beschriebenen Systems von den Modul-Metaklassen gesteuert werden kann.
- Gewinnung von Parameterdaten aus Monitoring-Daten (siehe Abschnitt 3.6.3).
- Manuelle Erstellung von Parameterdaten auf Basis der Intension des Spezifizierers.

Insgesamt beinhaltet der Ansatz eine erhebliche Komplexität bzgl. seiner Verwertung bei der Optimierung der automatisch generierten Implementierungen (siehe Kapitel 4). Die entsprechenden Methoden und Werkzeuge befinden sich noch in Entwicklung. Entsprechend können noch keine Aussagen über die effektive Wirksamkeit des Ansatzes gemacht werden.

3.6. Statistik und Monitoring

Eine wesentliche Zielsetzung bei der Entwicklung von XEC war auch die Möglichkeit, *quantitative Leistungsdaten* auf verschiedenen Niveaus ermitteln zu können. Dazu liefern die von XEC generierten Implementierungen einerseits grundlegende *statistische Daten* über ihre Ausführung, andererseits kann auch auf ein detailliertes und effizientes *Softwaremonitoring-System* mit offline-Auswertung zurückgegriffen werden.

Während die Statistik einen Überblick über Daten wie z. B. die Anzahl der verschickten bzw. empfangenen Nachrichten oder die gesamte bzw. durchschnittliche Zeit zur Transitionsauswahl und -Ausführung verschafft, kann mit Hilfe des Monitoringsystems eine detaillierte Untersuchung eines Laufes der Implementierung erfolgen. Eine objektorientierte Auswertebibliothek ermöglicht dabei, auch über die Möglichkeiten der bereits vorhandenen grafischen Auswertewerkzeuge hinaus spezifische Messdaten abzuleiten.

3.6.1 Ereignis-Tracing

Der *Tracing*-Mechanismus der XEC-Laufzeitbibliothek stellt die zentrale Schnittstelle zwischen den bisher dargestellten *Ablaufsteuerungsmechanismen* und den nun betrachteten *Auswertemechanismen* dar: Während erstere auf die Ausführung der eigentlichen Spezifikation im Sinne der Estelle-Semantik abzielen, sind die Auswertemechanismen ausschließlich zur *Analyse des Systems und seiner Ausführung* erforderlich und bleiben somit vollständig optional.

Das Ziel des Einsatzes einer solchen zentralen und einheitlichen Schnittstelle für alle angebundenen Auswertemechanismen ist es, den generierten Code und die plattformunspezifischen Teile der Laufzeitbibliothek (also alle Module außer `xecrt_context.cc`) von den Details der Auswertemechanismen zu isolieren. Gleichzeitig soll die Schnittstelle jedoch nur möglichst minimale Performanceeinbußen bewirken.

Über die Klasse `Tracing` werden aus der Laufzeitbibliothek heraus durch Aufruf von *Notify*-Methoden verschiedene Ereignisse gemeldet und zentral ausgewertet. Innerhalb des Moduls `xecrt_context.cc` wird diese Klasse durch eine plattformspezifischere Klasse `TracingInt` spezialisiert und ergänzt. Insgesamt existiert von diesen Klassen nur eine einzige Instanz `TracingInt tracing_int`, die außerhalb des Moduls `xecrt_context.cc` über `Tracing& tracing` abstrahiert[139] als Instanz ihrer Basisklasse zugreifbar ist.

Die Notify-Methoden umfassen dabei verschiedene Aspekte der zentralen Steuerung (z. B. `TracingInt::notifySystemInit(Specification*)`, siehe auch Zeile 2 von Beispiel 3.24-c auf Seite 99) oder der lokalen Auswahl und Ausführung von Modulen und Transitionen (z. B. `Tracing::notifyStartTest(SimpleTrans*)`).

Insgesamt können anhand der Aufrufe dieser Methoden alle wesentlichen Ereignisse der Systementwicklung nachvollzogen werden. Entsprechend können alle in den folgenden Abschnitten dargestellten Auswertemechanismen darüber angesteuert werden. Dabei können auf Basis der Parametrierung bei der Übersetzung des Systems (siehe Anhang A.8) die einzelnen Auswertemechanismen auch *selektiv deaktiviert* werden, so dass der damit jeweils verbundene

139. Dabei ist `tracing` ist eine feste Referenz auf das spezialisierte `tracing_int`. Dadurch kann die Definition der Spezialisierung ein internes Geheimnis des Moduls `xecrt_context` bleiben.

Overhead entfallen kann. Im Extremfall kann der Tracing-Mechanismus sogar soweit deaktiviert werden, dass die oben beschriebenen Notify-Methoden vollständig durch leere[140] Inline-Methoden ersetzt werden und dadurch kein Overhead mehr von ihrem Aufruf ausgeht.[141]

3.6.2 Ereigniszähler und Ausführungszeitstatistiken

Um auch ohne Einsatz des im nächsten Abschnitt vorgestellten Offline-Monitorings mit geringem Aufwand Ausführungsstatistiken gewinnen zu können, steht die Klasse `Counter` bereit. Die daraus gewonnenen Daten bilden die Grundlage für die meisten der in Kapitel 4 und Kapitel 6 dargestellten Auswertungen.

Jede Instanz der Klasse `Counter` enthält u. a. einen Zähler, der explizit mit den entsprechend überladenen Präfix-Operatoren "`++`" und "`--`" erhöht oder erniedrigt werden kann. Durch geeigneten Aufruf dieser Operatoren kann somit der interne Zähler synchron mit der Anzahl bestimmter Objekte im System gehalten werden. So existieren z. B. *Instanzzähler* für Module, Transitionen oder Nachrichten (s. u.). [142]

Alternativ können die Zähler auch als reine *Ereigniszähler* betrieben werden, indem bei jedem Auftreten eines Ereignisses (z. B. bei „`Module::evalCounter`" jeder modullokalen Auswahl) der "`++`"-Operator aufgerufen wird (also "`++Module::evalCounter`").[143]

Die einzelnen Counter-Instanzen werden über einen Mechanismus ihrer Basisklasse `ReportItem` automatisch in einer zentralen Liste geführt. So können durch einen zentralen Aufruf alle Zählerstände ausgegeben werden. Dies erfolgt insbesondere (gesteuert durch den Parameter "`-v`" des XEC-Laufzeitsystems, siehe Anhang A.4) bei der Termination der Implementierung.

Da jede Counter-Instanz zudem als Konstruktor-Parameter einen beschreibenden Text erhält, werden die Zählerstände entsprechend beschrieben. Bei Instanzzählern (s. o.) wird zudem neben der Gesamtzahl der "`++`"-Aufrufe auch der *akkumulierte Zählerstand*[144] („`NOW`") und der *Maximalwert des akkumulierten Zählerstands* („`PEAK`") ausgegeben (siehe Zeile „*interactions*" in Beispiel 3.26-a).[145]

So kann z. B. anhand der Ausgaben des Interaktionszählers sofort festgestellt werden, wie viele Nachrichten insgesamt verschickt wurden, wie viele gleichzeitig maximal existiert haben (also gleichzeitig „unterwegs" waren) und wie viele bei der Termination des Systems noch existierten. Der letztgenannte Wert zeigt bei einer Termination durch Deadlock insbesondere an, ob es in dem System *unspezifizierten Empfang* gegeben hat.

140. Die Methoden haben eine leeren Statement-Block und ihr Aufruf wird daher bei der Inline-Expansion durch den C++-Compiler völlig eliminiert.

141. Z. B. das Makefile-Templates `optimize` deaktiviert das Tracing-System, um eine optimierte Implementierung zu gewinnen (siehe Spalte „Logging" in Tabelle A.36 auf Seite 407).

142. z. B. "`++SimpleTrans::instCounter`" bzw. "`--SimpleTrans::instCounter`"

143. Bei Ereignis-Zählern wird der "`--`"-Operator nie aufgerufen. Die beschriebenen Zusatzwerte werden daher nur bei Zählern ausgegeben, bei denen "`--`" mindestens einmal aufgerufen wurde. Dadurch wird auch für Zähler, die eigentlich Instanzzähler sind, u. U. nur ein Wert ausgegeben.

144. also die Anzahl der "`++`"-Aufrufe abzüglich der Anzahl der "`--`"-Aufrufe

145. Im Beispiel 3.26-a wurden 3600 Interaktionen verschickt, es existierten jedoch immer nur maximal 2 gleichzeitig. Bei Ausgabe der Zähler (Systemtermination) existierten keine Interaktionen mehr.

Beispiel 3.26-a: Ausgabe der Instanz- und Ereigniszähler des XEC-Laufzeitsystems

```
> ===== COUNTER REPORT ============================================
> system cycles                  TOTAL:   3801
> external activities            TOTAL:      0
> module instances               TOTAL:     28   PEAK:   28   NOW: 20
> module activations             TOTAL:     28   PEAK:   28   NOW: 20
> module evaluations             TOTAL:   3822
> module events                  TOTAL:   3600
> module secondary events        TOTAL:      0
> module fireable repeatedly     TOTAL:      0
> module executions              TOTAL:   3800
> module tree-sleep shortcuts    TOTAL:      3
> module local-sleep shortcuts   TOTAL:      0
> module wakeup for exp. var.    TOTAL:      0
> module independent roots       TOTAL:     19
> trans. instances               TOTAL:     53   PEAK:   53   NOW: 43
> trans. executions              TOTAL:   3800
> trans. evaluations             TOTAL:   7644
> trans. evaluations (delay)     TOTAL:      0
> trans. executions  (delay)     TOTAL:      0
> timer instances                TOTAL:      0
> timer start                    TOTAL:      0
> timer expiration               TOTAL:      0
> interactions                   TOTAL:   3600   PEAK:   2   NOW: 0
> ANY-TYPE instances             TOTAL:      0
> copy ANY-TYPE (maybe multiref) TOTAL:      0
> casts to   ANY-TYPE            TOTAL:      0
> casts from ANY-TYPE            TOTAL:      0
> clock sleeps                   TOTAL:      0
> ===== END REPORT     ============================================
```

(Ende von Beispiel 3.26-a)

Die `Counter`-Instanzen werden typischerweise als Metaklassenkomponenten der jeweiligen Klassen des XEC-Laufzeitsystems angelegt. So enthält z. B. allein die Klasse `Module` 11 Counter-Instanzen.[146] Einige Zähler werden auch über den oben beschriebenen Ereignis-Tracing-Mechanismus angesteuert.

Der `Counter`-Mechanismus wird zudem auch im Codegenerator XEC selbst genutzt, um Statistiken zur Spezifikation zu gewinnen. Die Ausgabe wird auch hier mit der XEC-Option "`-v`" gesteuert (siehe Beispiel A.2.1):

Beispiel 3.26-b: Zählerausgabe von XEC

```
> ================================================
> Headers                        TOTAL:   6
> Headers w/o exp. var.          TOTAL:   6
> Headers w/o exp. ip.           TOTAL:   0
> Bodys                          TOTAL:   7
> Bodys w/o int. ip.             TOTAL:   7
```

146. In der Klassendefinition von Module: „`static Counter evalCounter, fireCounter, ...`"

```
> Bodys w/o int. var.            TOTAL:   5
> Bodys w/o prio-trans           TOTAL:   7
> Bodys w/o delay-trans          TOTAL:   7
> Bodys w/o children exp. vars   TOTAL:   7
> Transitions                    TOTAL:   13
> FROM-Clauses                   TOTAL:   0
> TO-Clauses                     TOTAL:   0
> PROVIDED-Clauses               TOTAL:   1
> WHEN-Clauses                   TOTAL:   9
> WHEN-Clauses with fixed IP     TOTAL:   9
> DELAY-Clauses                  TOTAL:   0
> DELAY-Clauses w. single value  TOTAL:   0
> DELAY-Clauses w. null value    TOTAL:   0
> PRIORITY-Clauses               TOTAL:   0
> ANY-Clauses                    TOTAL:   2
> ================================================
```

(Ende von Beispiel 3.26-b)

Aufbauend auf dem o. g. `ReportItem`-Mechanismus werden auch die von XEC intern geführten Zeit-Statistiken ausgegeben. Dazu existiert im Modul `xecrt_context` die Spezialisierung `TimingStatistics`, die einen akkumulierenden echtzeitbezogenen[147] Zeitmesser realisiert. Von dieser Klasse existieren drei Instanzen, die für die Aufsummierung der Zeiten verantwortlich sind, die insgesamt in Transitionsauswahlphasen (`timestatEval`), Transitionsausführungsphasen (`timestatExec`) und Systemblockadephasen (`timestatSleep`) verbracht werden. Zu einem Zeitpunkt ist immer genau einer der drei Zeitmesser aktiv.

Am Ende der Ausführung des Systems kann dann eine Statistik ausgegeben werden, wie viel Zeit jeweils in den drei Zuständen insgesamt verbracht wurde. Dabei wird zusätzlich auch die mittlere pro getestete bzw. ausgeführte Transition verbrauchte Zeit angegeben:

Beispiel 3.26-c: Ausgabe der Timing-Statistiken des XEC-Laufzeitsystems

```
> ===== TIMER REPORT   ============================================
> total trans. execution time  TOTAL: 0.001679 sec (3800 * 0.442 usec)
> total trans. evaluation time TOTAL: 0.000973 sec (7644 * 0.127 usec)
> total sleep time             TOTAL: 2.630744849e-07 sec
> ===== END REPORT     ============================================
```

(Ende von Beispiel 3.26-c)

Die Steuerung erfolgt durch Aufruf der Methode `TimingStatistics::start()`, durch die der bisher laufende Zeitmesser angehalten und der angegebene gestartet wird (siehe Zeile 15, 21 und 26 von Beispiel 3.24-c auf Seite 99). Durch diesen Mechanismus kann auf einfache und kompakte Weise eine Ausführungszeitstatistik für die Spezifikation gewonnen werden, und auch hier stammen viele der in den späteren Kapiteln dargestellten Ergebnisse aus dieser einfachen Form der Leistungsmessung durch das XEC-Laufzeitsystem.

147. Die Auswahl einer Zeitbasis für das Estelle-Zeitmodell (insbesondere die Zeitsimulation, siehe Abschnitt 3.5.2) hat keinen Einfluss auf diese Zeitmessung: Es wird immer nur die zur Ausführung benötigte Realzeit gemessen (siehe Abschnitt 3.6.4).

3.6.3 Timed-Traces und Offline-Monitoring

Für präzise Analysen können durch das optionale Monitoring-Paket von XEC während der Ausführung einer Implementierung Ereignishistorien („timed traces“) erzeugt werden. Um eine möglichst geringe Beeinträchtigung der Performance durch dieses Software-Monitoring-System gewährleisten zu können, werden

(i) nur die relevanten Messdaten erfasst (parametrierbar),

(ii) die Messdaten in einem Ringpuffer im Hauptspeicher in Rohform gesammelt,

(iii) die Messdaten bei der Termination der Implementierung in eine Datei geschrieben.

Zur Offline-Weiterverarbeitung der Daten steht eine Klassenbibliothek zur Verfügung, auf deren Basis gezielte Auswertungen realisiert werden können. Hier hat sich die Nutzung kleiner spezialisierter *Auswertetools* bewährt, da mit diesen automatisiert spezifische Analysen zu einzelnen Fragestellungen durchgeführt werden können.

Eine weitere Analysemöglichkeit ergibt sich durch die Konvertierung der Messdaten in das von *PATO*[148] („*Performance Analysing Tool*“, [Pen96]) verwendete Dateiformat, um sie mit diesem Werkzeug grafisch auswerten zu können (siehe Abb. 3-16).

Die Anbindung des Software-Monitoring-Systems an das eigentliche XEC-Laufzeitsystem erfolgt über den Ereignis-Tracing-Mechanismus (siehe Abschnitt 3.6.1) und konnte so relativ stark von dem Restsystem isoliert werden. Lediglich einige kleinere Erweiterungen wie z. B. zur Unterstützung der Verfolgung konkreter Pakete erforderten Erweiterungen außerhalb des `xecrt_context`-Moduls.

Zur Aktivierung des Software-Monitorings ist zunächst die Aktivierung der entsprechenden Übersetzungsoptionen erforderlich (z. B. durch die XEC-Option “`-m monitor`“, siehe Anhang A.8). Beim Aufruf der generierten Implementierung kann dann über die Option “`-monitor [level [name [size]]]`“ das Monitoring aktiviert und parametriert werden, wobei neben dem Namen der Ausgabedatei (*`name`*, default „`monitor.log`“) und der Größe des Ringpuffers (*`size`*, default 100'000) auch mit *`level`* die Menge der Ereignisklassen angegeben werden kann, die eingeschlossen werden sollen (default: „`smtp`“):

- „`s`“: Spezifikationszyklen,
- „`m`“: Module,
- „`t`“: Transitionen und
- „`p`“: Pakete

Die Ausgabedatei enthält neben den Timed-Traces auch ein Dictionary zur Interpretation der *Ereignistypen* (z. B. das Feuern einer Transition), der *statischen Strukturhierarchie* der Spezifikation (Module, Transitionen, Pakettypen) und der *dynamischen Objekthierarchie* (Modul- und Transitionsinstanzen). Dadurch kann die dynamische und statische Struktur der Spezifikation allein anhand dieser Datei rekonstruiert werden. Insbesondere bestehen die einzelnen Timed-Traces neben dem Zeitstempel lediglich noch aus einem Ganzzahl-Tripel als Referenz in diese drei Beschreibungslisten (siehe Anhang A.9.1).

148. PATO wurde ursprünglich für den Implementierungsgenerator DINGO (siehe Abschnitt 2.1.4) entwickelt und ermöglicht eine statistische Analyse von solchen Timed-Traces. Da die ursprünglich enthaltene und auf DINGO zugeschnittene Monitoring-Bibliothek nicht sinnvoll auf XEC übertragen werden konnte, wurde eine Konvertierung der XEC-eigenen Monitoring-Daten in das PATO-Format entwickelt.

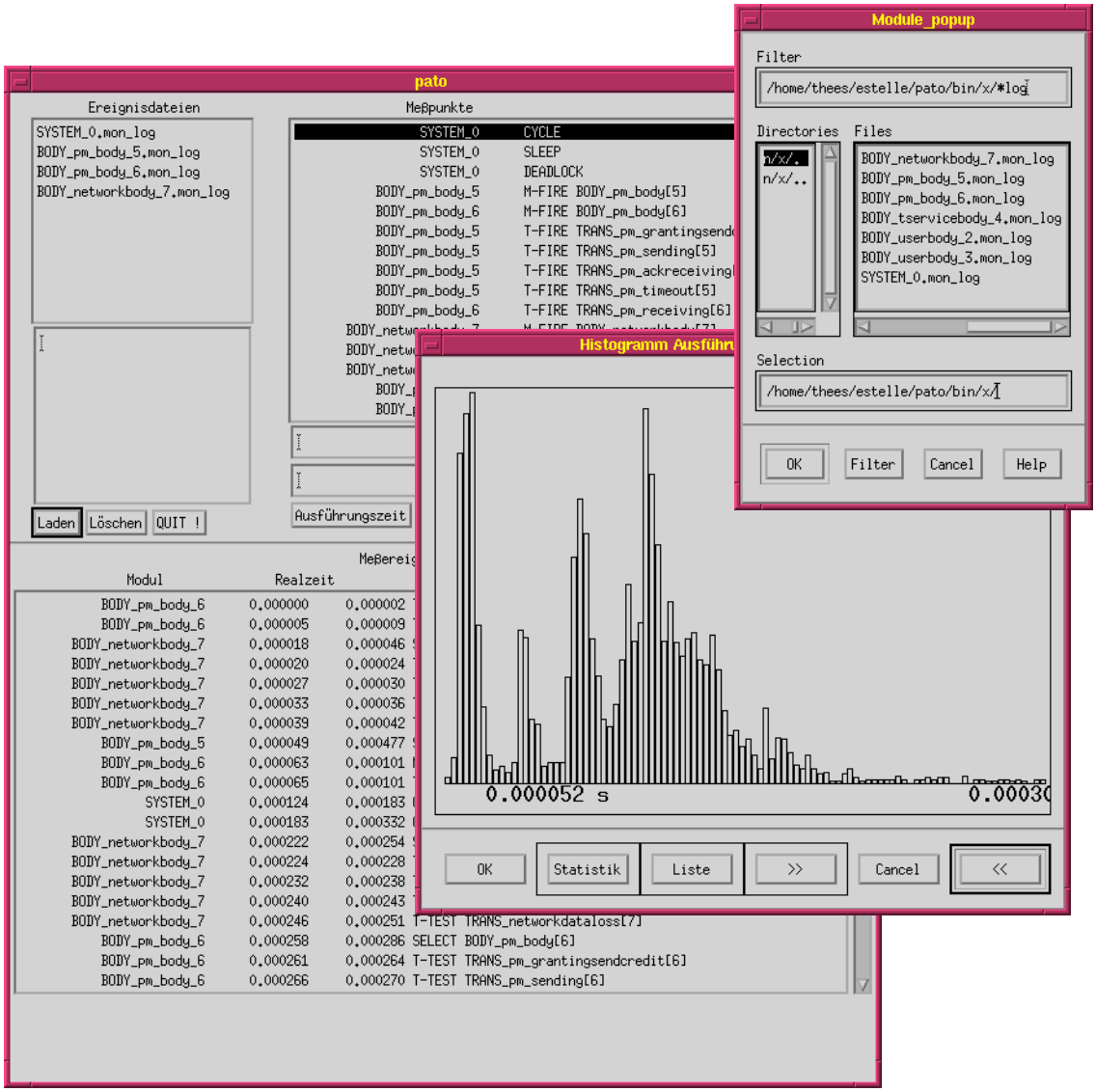


Abbildung 3-16: Grafische Oberfläche des Offline-Monitoring-Werkzeugs PATO

Die Zeitstempel selbst sind dabei Rohdaten aus der plattformspezifischen Zeitmessmethode (siehe nächsten Abschnitt) und werden anhand der ebenfalls im Header der Datei angegebenen Parameter bei der Offline-Analyse interpretiert. Diese können z. B. mit dem Werkzeug `mondump` lesbar gemacht werden (siehe Anhang A.9.2).

3.6.4 Realzeitmessung

Die effiziente und präzise Ermittlung der aktuellen Realzeit ist eine wichtige Grundlage für die realzeitbezogene Estelle-Uhr (siehe Abschnitt 3.5.2.1), die Ausführungszeitstatistiken des XEC-Laufzeitsystems (siehe Abschnitt 3.6.2) und ganz besonders auch für die Erzeugung von Timed-Traces zum Zwecke des Offline-Monitorings (siehe Abschnitt 3.6.3).

Bei der Installation des XEC-Toolkits wird im Rahmen der Selbstkonfiguration ermittelt, welche geeigneten Zeitmessfunktionen verfügbar sind. Dazu werden in der Quelldatei „`xecrt_hrtime.h`“ zwei entsprechende Zugriffsfunktionen definiert:

- `unsigend long long hrtime()`
  liefert einen Zeitstempel in einer plattformspezifischen Einheit.
- `unsigned long long hrtimescale()`
  liefert den dazu passenden Umrechnungsfaktor (Anzahl der Ticks pro Sekunde).

Diese Uhrendefinitionen werden dann durch eine der folgendenden Funktionen implementiert:

(a) `gettimeofday(...)`
auf den meisten UNIX-artigen Plattformen; liefert Mikrosekunden in einem Record

(b) `gethrtime(...)`
auf Solaris-Plattformen; liefert Echtzeit in Nanosekunden

(c) IA32-Maschineninstruktion `RDTSC`
auf IA32[149]-CPUs; liefert gezählte CPU-Zyklen des Hosts

Speziell die letzte Methode ist extrem effizient und genau, erfordert bei der Umrechnung jedoch eine Anpassung an die CPU-Taktfrequenz. Insbesondere die Monitoring-Zeitstempel werden zur Optimierung dennoch in diesem Format abgelegt und erst bei der Offline-Analyse umgerechnet (s. o.).

3.6.5 Debugger

Das XEC Laufzeitsystem bietet auf Basis des oben beschriebenen Ereignis-Tracing-Mechanismus (siehe Abschnitt 3.6.1) auch ein *Debugger-Interface*, mit dessen Hilfe während der Ausführung der Spezifikation der Systemzustand interaktiv ausgewertet und die Transitionsauswahl (im Rahmen des Indeterminismus) beeinflusst werden kann.

An dieses Interface kann neben einer einfachen integrierten Textmodus-Oberfläche auch der grafisch interaktive „*Estelle Graphical Debugger*“ EGD ([Wen98], siehe Abb. 3-17) angekoppelt werden. Dieser ist als Java-Programm realisiert und interagiert über eine Socket-Schnittstelle mit der Implementierung.

149. z. B. Intel-Pentium-IV oder AMD-Athlon

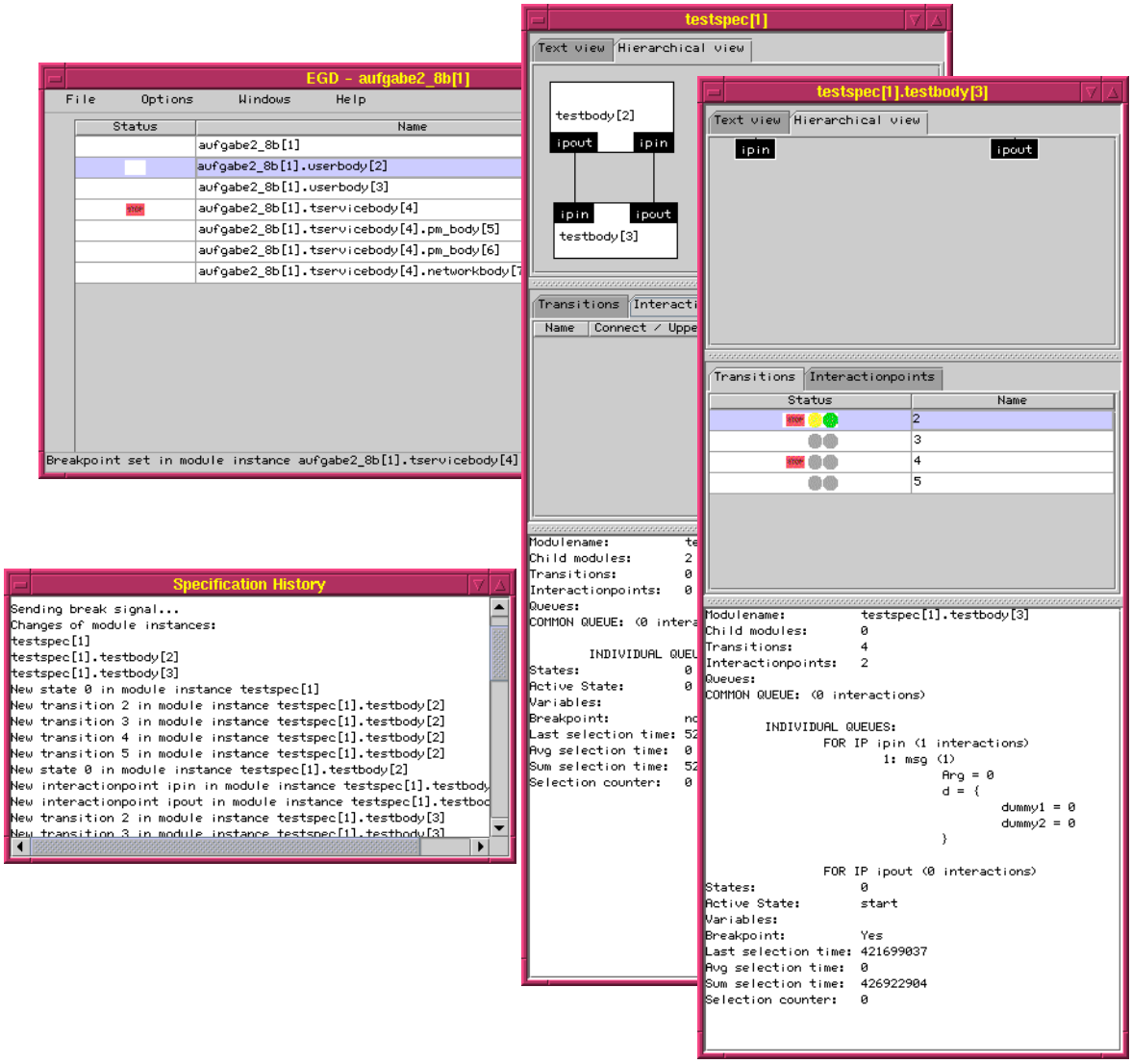


Abbildung 3-17: Grafische Oberfläche des interaktiven Debuggers EGD

3.7. Estelle-Erweiterungen

XEC implementiert einige Estelle-Erweiterungen, die zur Steigerung der Ausdrucksfähigkeit von Estelle oder als Basis zur effizienten Implementierung dienen. Bis auf die *Asynchronous-Process-Erweiterung* sind alle Erweiterungen als Teil der vorgestellten Arbeit entwickelt worden. Einige weitere in diesem Text andiskutierte Estelle-Erweiterungen wurden dagegen nicht implementiert, da jeweils eine günstigere Lösung identifiziert wurde.

Wir geben hier einen kurzen Überblick über die im XEC-Toolkit implementierten Estelle-Erweiterungen. In Tabelle 3.4 sind die zu ihrer Aktivierung erforderlichen PET-Optionen dargestellt.

- *Asynchronous-Process*

  Die *Asynchronous-Process-Erweiterung* [BrGo94] ergänzt die Modulattributierung von Estelle um die Fähigkeit, innerhalb eines Subsystems asynchron operierende Kindmodule spezifizieren zu können (siehe auch Abschnitt 4.2.2.4). Die technische Umsetzung dieser Erweiterung in PET und XEC ist relativ unaufwändig. Dies gilt ebenso für die Umsetzung in der XEC-Laufzeitbibliothek, da aufgrund der sequentiellen Ausführung letztlich nur die globale Modulauswahl beeinflusst wird. Wir setzen diese Erweiterung in Abschnitt 4.2 zur Auswahloptimierung ein.

- *Independent-Module*

  Die in Abschnitt 4.2.2.4 eingeführte *Independent-Module*-Erweiterung dient ebenfalls zur Erweiterung der Modulattributierung von Estelle, macht jedoch geringere Einschnitte in die Estelle-Semantik. Auch hier ist die technische Umsetzung in PET und XEC relativ unaufwändig. Die Umsetzung dieser Erweiterung im Laufzeitsystem ist (unabhängig vom Ausführungsmodell) ebenfalls einfach. Wir setzen auch diese Erweiterung in Abschnitt 4.2 zur Auswahloptimierung ein.

- *Explizite Referenzübergabe*

  Die in Abschnitt 6.4.5 eingeführte Estelle-Erweiterung zur *expliziten Referenzübergabe* ermöglicht eine konzeptionell einfache und zudem sehr effizient implementierbare Möglichkeit zur Spezifikation der Übergabe von Nutzdaten zwischen Protokollkomponenten. Wir setzen sie entsprechend in Abschnitt 6.4 zur Optimierung des Datentransports ein.

- *Containertyp-Erweiterung*

  In Abschnitt 5.2.2 führen wir die *Containertyp-Erweiterung* ein. Diese Erweiterung ergänzt stark getypte Beschreibungstechniken wie Estelle um die Möglichkeit, abstrakte Transportdienste formal (also semantisch und syntaktisch eindeutig interpretierbar) zu spezifizieren. Die Implementierung besteht im Wesentlichen nur aus der Erweiterung des Typmodells von PET und dem Hinzufügen einer Klassendefinition in die XEC-Laufzeitbibliothek.

  Wir nutzen diese Erweiterung in Kapitel 5 zur Steigerung der Ausdrucksfähigkeit von Estelle bei der Spezifikation heterogener oder gar offener Kommunikationssysteme. Zudem wird sich in Kapitel 6 zeigen, dass sie als Grundlage für eine sehr effizient implementierbare Übergabe von Nutzdaten zwischen Protokollkomponenten dienen kann. Wir setzen sie entsprechend in Abschnitt 6.4 zur Optimierung des Datentransports ein.

- *Open-Estelle*

  In Kapitel 7 entwickeln wir die Estelle-Erweiterung *Open-Estelle* zur Erweiterung der Ausdrucksfähigkeit von Estelle bzgl. der formalen Spezifikation offener Systeme und ihrer Komposition. Die Implementierung der Erweiterung erforderte nicht unerhebliche Modifikationen des Compiler-Frontends PET. Die Integration in XEC und das XEC-Laufzeitsystem war dagegen kompakt realisierbar. Neben der Fähigkeit zur getrennten Implementierung und nachträglichen Komposition offener Teilsysteme durch XEC wurde auch ein Werkzeug zur Verschmelzung offener Systeme auf Spezifikationsebene entwickelt.

Estelle-Erweiterung	**PET-Kommandozeilenoption**	**Bemerkungen**
Asynchronous-Process	`-xasync`	[BrGo94], siehe Abschnitt 4.2.2.4
Independent-Module		siehe Abschnitt 4.2.2.4
Explizite Referenzübergabe	`-xptria`	siehe Abschnitt 6.4.5
Containertyp (any-type)	`-xat`	siehe Abschnitt 5.2

Open-Estelle	`-xopen`
	`-xopen_ur`
	`-I ...`
		siehe Kapitel 7

(alle)	`-x`	alle o. g. Erweiterungen aktivieren

Tabelle 3.4: PET-Optionen zur Aktivierung von Estelle-Erweiterungen (V2.02)

3.8. Zusammenfassung

In diesem Kapitel haben wir den Estelle-Implementierungsgenerator XEC („*eXperimental Estelle Compiler*“) vorgestellt, den wir als Plattform für die Implementierung, detaillierte Untersuchung und quantitative Bewertung von Implementierungstechniken entwickelt haben.

XEC generiert aus einer Estelle-Spezifikation ein analog strukturiertes hierarchisches C++-Klassensystem, das aufbauend auf der objektorientierten XEC-Laufzeitbibliothek das spezifizierte System implementiert. Dabei kommt ein Kodierungsschema zum Einsatz, das soweit wie möglich Strukturen, Typen, Zustandsräume und Operationen 1:1 in äquivalente C++-Konstrukte abbildet. So behält XEC u. a. den strukturierten Namensraum der Estelle-Spezifikation auf deterministische Weise bei und kann so auf den Einsatz künstlich generierter Namen fast vollständig verzichten. Dies wird später eine wichtige Grundlage zur komponentenweisen Implementierung unter Open-Estelle darstellen (siehe Kapitel 7).

Die Ausführung der Spezifikation erfolgt unter umfassender Kontrolle der Laufzeitbibliothek, so dass viele Implementierungsvarianten und Optimierungsansätze allein durch Manipulation der Laufzeitbibliothek realisierbar sind und eine Modifikation des Codegenerators selbst meist nicht erforderlich ist (siehe Kapitel 4 und 6). Dies vereinfacht die Realisierung von Implementierungsvarianten und bildet zusammen mit dem klaren Kodierungsschema die Grundlage für eine „offensichtliche Korrektheit“ der gewonnenen Implementierungen.

Nicht zuletzt auch aufgrund der sehr guten Integration von Ausführungsstatistik- und Softwaremonitoring-Funktionalitäten erfüllt XEC alle zu Beginn des Kapitels identifizierten Anforderungen an eine Implementierungs- und Experimentierplattform für formale Beschreibungstechniken. Er stellt somit selbst ein wichtiges Teilergebnis dieser Arbeit dar.

Über den kompletten Sprachumfang von Estelle hinaus unterstützt XEC auch verschiedene Estelle-Erweiterungen, die wir in den folgenden Kapiteln zur Verbesserung der effizienten Implementierbarkeit oder zur Steigerung der Ausdrucksfähigkeit von Estelle bei Bedarf jeweils einführen werden.

4. Effiziente Implementierung

Im vorangegangenen Kapitel 3 wurde das Implementierungsmodell von XEC im Sinne eines *Implementierungsframeworks* zusammen mit einigen ersten *Referenz-Implementierungen* von verschiedenen Aspekten der Auswahl und Ausführung von Transitionen vorgestellt.

In diesem Kapitel untersuchen wir nun aufbauend auf dieser Infrastruktur anhand der in Abschnitt 2.2 diskutierten Ansätze verschiedene Optimierungstechniken bzgl. der *Kontrollflussabwicklung* der generierten Implementierungen. Dabei stehen Optimierungen *konzeptioneller Natur* im Vordergrund, die größtenteils auch unabhängig von einer konkreten Implementierungstechnik anwendbar sind.

Wir beginnen diese Untersuchung in Abschnitt 4.1 mit einem kurzen Überblick über die in XEC verfolgten Konzepte zur Optimierung der *Basisperformance* und des *Schaltens von Transitionen* der gewonnenen Implementierung.

Die damit verbundenen Fragestellungen dienen in erster Linie jedoch nur als Grundlage für die anschließend diskutierten Optimierungen von *Managementaspekten*, die für automatenbasierte formale Beschreibungstechniken spezifisch sind. Diese werden anhand der *globalen* (Abschnitt 4.2) und der *modullokalen Transitionsauswahl* (Abschnitt 4.3) entwickelt und quantitativ evaluiert.

Während die Bewertung der Effizienz verschiedener Optimierungsmaßnahmen in diesen Abschnitten sich primär auf die Anzahl der zu behandelnden Transitionen und Module bezieht, wird in Abschnitt 4.4 dann eine ausführungszeitbezogene Bewertung der Optimierungsmaßnahmen vorgenommen.

In Abschnitt 4.5 folgt dann schließlich eine kurze Zusammenfassung der gewonnenen Ergebnisse, wobei insbesondere die Frage der Übertragbarkeit auf andere Beschreibungstechniken diskutiert wird.

4.1. Basisperformance und Transitionsausführung

Die Estelle-Semantik gibt als Ausführungsmodell für das spezifizierte System wechselnde Phasen der *Auswahl* von Transitionen und des *Schaltens* der ausgewählten Transitionen vor. Diese beiden Phasen sind letztlich in jeder semantiknahen[1] Implementierung einer solchen Spezifikation wieder zu finden, unabhängig davon, ob diese manuell oder automatisch erzeugt wurde. Die Auswahlphase kann bei in dieser Hinsicht ausreichend *einfachen* Spezifikationen zum Teil auch stark reduziert ablaufen, wenn für die vorgegebene Spezifikation ein entsprechendes Modell semantisch zutreffend anwendbar ist. Dies kann so weit gehen, dass die Trennung zwischen Transitionsauswahl und Transitionsausführung zum Teil sogar weitgehend aufgehoben wird, und die Aufgabe der Transitionsauswahl bereits zur Compilezeit weitestgehend eliminiert wird. Ein Beispiel dafür ist das in Abschnitt 4.2.1.2 diskutierte *Activity-Thread-Modell*, das jedoch nur unter erheblichen stilistischen Einschränkungen anwendbar ist.

Im Gegensatz zu den möglichen Vereinfachungen der Transitionsauswahlphase sind die zum *Feuern der Transitionen* vorgesehenen Operationen meist nur in weitaus geringerem Umfang durch konzeptionelle Maßnahmen optimierbar. Die Ursache dafür liegt in der stark imperativ ausgerichteten Struktur der (weitestgehend im Pascal-Subset von Estelle beschriebenen) Transitionsrümpfe, die es dem Spezifizierer im Rahmen der Estelle-Abstraktionen ermöglichen, sehr implementierungsnah und damit auf relativ niedrigem konzeptionellem Niveau zu spezifizieren.

Setzt man nun voraus, dass die in den Transitionsrümpfen vorgegebenen Operationen letztlich zur Protokollabwicklung zwingend erforderlich sind (siehe Abschnitt 2.2.2), so kann durch eine effizienzorientierte, aber weitestgehend direkte 1:1-Abbildung der spezifizierten Operationsprimitive auf die Implementierungssprache (in unserem Fall C++) eine Implementierung gewonnen werden, die im Wesentlichen dem entspricht, was ohnehin auch Ergebnis einer manuellen Implementierung der Spezifikation gewesen wäre.

XEC setzt in der vorliegenden Version bei der Implementierung von Anweisungsblöcken und Ausdrücken auf dieses Grundprinzip: Für jede (strukturierte oder unstrukturierte) Anweisung und für jeden (Teil-) Ausdruck wird eine semantisch treffende, aber effiziente Implementierung in der Zielsprache eingesetzt (siehe Kapitel 3). Diese Vorgehensweise bietet auf Grund der strukturellen Ähnlichkeiten zwischen dem Pascal-Anteil von Estelle und der Implementierungs-Zielsprache C++ erhebliche Vorteile:

(i) Sie *vereinfacht den Codegenerator*, da die Implementierung zu großen Teilen durch eine Art Makro-Substitution beschrieben werden kann.

(ii) Sie unterstützt die Optimierungstechniken des nachgeschalteten C++-Compilers, da die entstehenden Codestrukturen auf Grund der Nähe zur Ausgangspezifikation große Ähnlichkeit mit handgeschriebenem C++-Code haben und somit die (ebenfalls auf handgeschriebenen Quellcode ausgerichteten) *Optimierungs-Heuristiken* greifen.[2]

(iii) Sie ermöglicht auf Grund der relativen Einfachheit und Vorhersagbarkeit des Kodierungsprozesses dem versierten Spezifizierer, einen bezüglich der Implementierung *effizienten Spezifikationsstil* anzuwenden. Dabei sind jedoch im Normalfall keine speziellen Anpassungen an den Codegenerator erforderlich, sondern es genügt die Anwendung allgemeiner Grundkonzepte der effizienten Programmierung.[3]

1. Zur strukturellen und operationellen *Semantiknähe* von korrekten Implementierungen siehe auch Kapitel 1.

Zusammen mit einer sorgfältig auf Laufzeit-Performance optimierten Implementierung der dabei verwendeten Basisoperationen (aus der Codegenerierung oder in den von der Laufzeitbibliothek zur Verfügung gestellten Mechanismen) ergibt sich so eine Implementierung, die über weite Strecken einer möglichen manuellen Implementierung der selben Protokollmechanismen entspricht und somit bezüglich ihrer Laufzeit-Performance mit ihr konkurrieren kann.

Seine Grenzen erreicht dieses Implementierungskonzept jedoch bei der Optimierung von spezifizierten Operationen, die *nicht* eigentlicher Teil der Problemlösung sind, sondern durch die Spezifikation des Protokolls in einer formalen Beschreibungstechnik mit ihren jeweiligen Abstraktionen bedingt sind. Ein typisches Beispiel ist der Transport von Nutzdaten durch einen Protokollstack. Während die automatische Implementierungsgenerierung aus einer formalen Estelle-Spezifikation mit ihren komponentenweise getrennten Zustandsräumen und der Übertragung von Nutzdaten als „*by-value*“-Nachrichtenparameter zunächst nur eine weitgehend analoge Implementierung mit einer entsprechenden Anzahl von Kopieroperationen auf diese Nutzdaten erzeugt, kann ein manueller Implementierer seine globale Sicht auf die Verwendung der Nutzdaten und seine Kenntnis bzw. Interpretation des Ausgangsproblems, welches der formalen Spezifikation zu Grunde lag, einsetzen, um Kopieroperationen der Nutzdaten auf ihrem Weg durch den Protokollstack zu vermeiden (siehe Kapitel 5).

Diese Problematik basiert zu großen Teilen auf einer Steigerung der eigentlichen Problemkomplexität bei der Formalisierung in einer konkreten formalen Beschreibungstechnik mit ihren jeweiligen Abstraktionen. Sie beruht jedoch auch auf der allgemeinen Fragestellung, wie *implizite globale Eigenschaften* einer Systembeschreibung gewonnen und zur *lokalen* Optimierung eingesetzt werden können. Diese Aufgabe besitzt erhebliche Komplexität und ist nach wie vor Forschungsgegenstand im Bereich der automatischen Optimierung durch Compiler für imperative Programmiersprachen (siehe auch [AlSeUl85], [GBJ00]).

Die zentrale Zielsetzung unserer Untersuchungen bleibt dagegen die Frage nach der effizienten Implementierbarkeit der *spezifischen Aspekte formaler Beschreibungstechniken*. Dabei werden Fragestellungen aus dem konventionellen Compilerbau imperativer Programmiersprachen nur am Rande behandelt.

---

2. So verzichtet XEC zum Beispiel auf die von EC bei der Codegenerierung allgegenwärtig eingefügten Hilfsvariablen, in die Parameter und Zwischenergebnisse von Ausdrücken explizit abgelegt werden. Diese Hilfsvariablen können bei einigen Kombinationen von Zielplattform und Compiler u. U. eine Performancesteigerung bewirken, eine Verallgemeinerung dieses positiven Einflusses scheint jedoch (zumindest bei modernen optimierenden Compilern) eher unwahrscheinlich, da die so eingeführte erhöhte Komplexität des C-Quellcodes von dem C-Compiler erst wieder durch aufwändige Datenflussanalysen auf das Ausgangsproblem zurückgeführt werden muss. Sollte die Ablage von Zwischenergebnissen in künstlich eingeführten Hilfsvariablen auf einer konkreten Zielplattform einen Performancevorteil bieten, so wird ein moderner optimierender Compiler diese Hilfsvariablen selbständig einführen.

3. So sollten zum Beispiel bezüglich ihrer Komplexität effiziente Algorithmen angewandt oder die Berechnung konstanter Ausdrücke aus Schleifen ausgelagert werden, so weit dies nicht sogar aus den oben genannten Gründen noch nachträglich durch den nachgeschalteten C++-Compiler geleistet werden kann (siehe auch „*common subexpression elimination*“ oder „*strength reduction*“, z. B. in [AlSeUl85]).

4.2. Globale (Modul-) Auswahl

Das XEC-Laufzeitsystem betreibt die generierte Implementierung gemäß der Estelle-Semantik in abwechselnden Phasen der Auswahl und Ausführung von Transitionen. Die Transitionsauswahl, deren Optimierung in diesem Kapitel im Vordergrund steht, kann in zwei Ebenen untergliedert werden:

(i) die *modullokale Auswahl*, also die Auswahl einer schaltbaren Transition in einer gegebenen Modulinstanz und

(ii) die darauf aufbauende *globale Auswahl*, also die Auswahl eines Moduls, das eine solche schaltbare Transition anbietet.

In diesem Abschnitt entwickeln und evaluieren wir verschiedene Optimierungstechniken der *globalen (Modul-) Auswahl (ii)*, während die *modullokale Auswahl (ii)* Gegenstand von Abschnitt 4.3 sein wird.

Zielsetzung der Optimierung der globalen Auswahl ist es, mit einer möglichst geringen Anzahl[4] von Modultests diejenigen Module zu identifizieren, deren jeweilige schaltbaren Transitionen in einer folgenden Ausführungsphase geschaltet werden können.

Als Qualitätsindikator für die globale Auswahl definieren wir entsprechend die *globale Modulauswahleffizienz*[5] Eff_{mod} als Quotienten zwischen der Anzahl der geschalteten[6] (nm_{fired}) und der getesteten *Modulinstanzen* (nm_{tested}):

$$Eff_{mod} = nm_{fired} / nm_{tested}$$

Dieser Effizienzbegriff ist zunächst *von der Ausführungszeit unabhängig* und erlaubt es uns, viele Szenarien und Auswahlmechanismen rein analytisch und unabhängig von der Effizienz der sonstigen Implementierung (insbesondere der modullokalen Transitionsauswahl) zu bewerten. Wir werden später in Abschnitt 4.4 dann auch die realzeitbezogene Effizienz der Implementierungen mit einbeziehen.

4.2.1 Konventionelle Implementierungsmodelle

Bei der Implementierung von formal beschriebenen kommunizierenden EFSMs (*Extended Finite State Machines*) auf prozessorientierten Plattformen kann man zwischen zwei grundlegenden Implementierungsmodellen unterscheiden, dem *Server-Modell* und dem *Activity-Thread-Modell* [Svo89].

---

4. Genau genommen ist die Minimierung der Summe der für die modullokalen Auswahlen getesteter Module benötigte Zeit das relevante Kriterium. Bei der hier vorgenommenen isolierten Betrachtung der globalen Auswahl bietet die Anzahl der Module jedoch eine erste Abschätzung (siehe auch Abschnitt 4.3.1).

5. Zur sprachlichen Vereinfachung nutzen wir den Begriff „globale Auswahl“ (soweit er sich nicht explizit auf Transitionsauswahlen bezieht) als Synonym für die „globale Modulauswahl“.

6. Wir benutzen die Begriffe *„testen“* und *„schalten“ von Modulen* in Analogie zur entsprechenden Begriffsbildung bei Transitionen. Insbesondere nutzen wir auch hier die Begriffe *„feuern“* und *„ausführen“* gelegentlich als Synonyme für *„schalten“*.

Beide Modelle haben hinsichtlich der notwendigen stilistischen Einschränkungen, ihrer automatischen Implementierbarkeit und der Effizienz der resultierenden Implementierungen Vor- und Nachteile, die wir im Folgenden kurz darstellen.

4.2.1.1 Server-Modell

Gerade bei stark transitionsorientierten Beschreibungstechniken wie Estelle bietet sich zur Implementierung das *Server-Modell* an. Dabei existieren ein oder mehrere Server-Prozesse, die im Wechsel Transitionsauswahl- und -Ausführungszyklen nach dem in Abb. 4-1 dargestellten Schema durchführen.

```
while true do
    begin
        select_transitions();
        if found then
            fire_transitions();
    end;
```

Abbildung 4-1: Grundprinzip des Server-Modells

Charakteristisch für die Anwendung des Server-Modells bei der Implementierung von Estelle-Spezifikationen ist, dass durch die enge Analogie zur Estelle-Semantik keinerlei grundsätzliche Einschränkungen beim Spezifikationsstil notwendig sind, um automatisch semantikkonforme Implementierungen direkt aus Spezifikationen generieren zu können, d.h. das Server-Modell ist ein *universelles Implementierungsmodell* für Estelle.

Die Nachteile des Server-Modells werden jedoch gerade durch diese Flexibilität verursacht. Eine naive (d.h. nicht speziell optimierte) Server-Implementierung wird nach jedem Transitionsausführungszyklus erneut einen vollständigen Auswahlzyklus in den betreffenden Subsystemen durchführen müssen.

So wird in dem in Abb. 4-2 dargestellten Beispiel nach dem Schalten einer Transition und dem damit verbundenen Nachrichtentransport[7] (1., 3. und 5.) jeweils eine Transitionsauswahl[8] in Form einer rekursiven Suche nach einer schaltbaren Transition (2. und 4.) durchgeführt.

In dem konkreten Beispiel wurden insgesamt 10 Module getestet und 2 Transitionen gefeuert. Es ergibt sich somit eine Auswahleffizienz bzgl. des Tests von Modulinstanzen von 20%.

Zur Abschätzung der *mittleren* Komplexität des Auswahlverfahrens gehen wir von einem ACTIVITY-attributierten Modulinstanzbaum mit *n* Knoten aus. Nehmen wir weiter an, dass jeweils immer genau eine Modulinstanz eine oder mehrere schaltbare Transitionen anbietet und die Wahrscheinlichkeit dafür bei allen Modulinstanzen gleich ist. Da die ACTIVITY-Attributierung es erlaubt, optimieren wir die Auswahl dadurch, dass beim Vorfinden einer Modulin-

7. Die erste Nachricht (1.) möge ebenfalls als Folge des Feuerns einer Transition oder (im Sinne eines offenen Systems) als externer Stimulus des beobachteten Systems entstanden sein (s. u.).

8. Wir gehen im Beispiel von einer Modulattributierung aus, die bei der Auswahl nicht zwingend die Betrachtung aller Modulinstanzen einer Ebene erfordert (z. B. durchgängig (SYSTEM-)ACTIVITY).

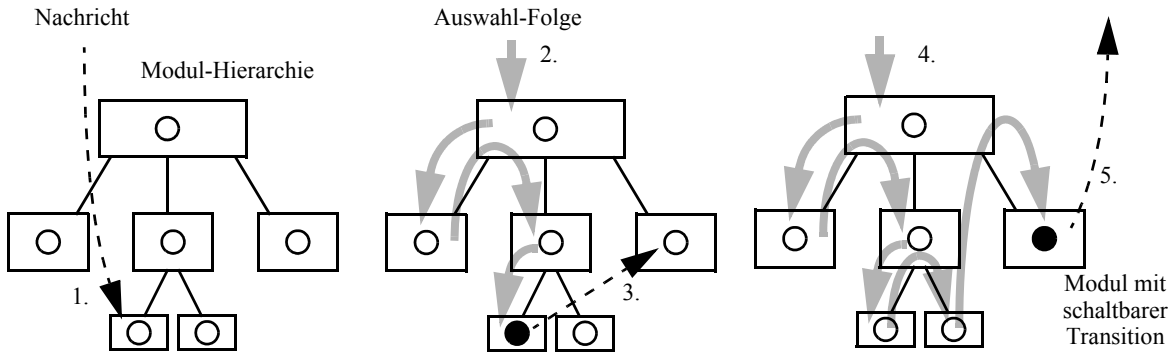


Abbildung 4-2: Auswahl- und Ausführungszyklen im Server-Modell

stanz mit einer schaltbaren Transition die Auswahl sofort[9] (erfolgreich) abbricht. Dann müssen im Mittel $n/2$ Modulinstanzen getestet werden. In jedem Fall ist die algorithmische Komplexität des Verfahrens linear, also aus $O(n)$.

Bei einem sehr einfachen Protokollszenario, wie es in Abb. 4-2 dargestellt ist, ergibt sich mit *n=6*, dass im Mittel dreimal mehr Modulinstanzen getestet als gefeuert werden, im Mittel die *globale Auswahleffizienz* bzgl. des Tests von Modulinstanzen Eff_{mod} also ca. 33% beträgt.

In komplexeren Systemen kann dieser Wert aufgrund der größeren Zahl von Transitionen sehr viel ungünstiger ausfallen. So beinhaltet selbst die einfachste sinnvolle Testumgebung[10] für die in Abschnitt 2.3.4 vorgestellte Estelle-Spezifikation von XTP 4.0 bereits 18 Modulinstanzen (siehe auch Abb. 4-5). Unter den o. g. Vorbedingungen ergibt sich somit eine zu erwartende Auswahleffizienz bzgl. des Tests von Modulinstanzen von nur 11%. Bei komplexeren Szenarien[11] kann schnell auch die 1%-Marke unterschritten werden.

Bei diesen Werten ist zu beachten, dass wir bisher lediglich die Zahl der zu prüfenden *Module* betrachtet haben. Die Zahl der dabei zu prüfenden *Transitionen*[12] liegt typischerweise um mehr als eine Größenordnung[13] höher und es ergibt sich eine entsprechend geringere Gesamtauswahleffizienz für die Transitionen. Wir kommen später in Abschnitt 4.3 auf diesen Punkt zurück.

Offensichtlich kommt der effizienten globalen Auswahl eine ganz erhebliche Bedeutung bei der Optimierung der Implementierung zu. Ziel muss es dabei sein, möglichst frühzeitig eine Modulinstanz mit schaltbaren Transitionen zu finden. Ein wesentlicher Ansatz ist hier – im Gegensatz

9. Die Einhaltung der *Vater-Sohn-Priorität* kann durch eine entsprechende Reihenfolge beim Testen (z. B. im Rahmen des rekursiven Durchlaufens des Modulinstanzbaums) sichergestellt werden.

10. Ein unstrukturiertes Netzwerkmodul (1 Modulinstanz), zwei unstrukturierte Benutzermodule (je eine Modulinstanz), zwei XTP-Protokollmaschinen (je $5n+2$ Modulinstanzen bei *n* bestehende Verbindungen) und das Spezifikationsmodul selbst; bei einer einzigen bestehenden Verbindung ergeben sich somit 18 Modulinstanzen (siehe auch Abb. 2-6 auf Seite 20).

11. In einem Szenario mit vier XTP-Instanzen wird die 1%-Marke bereits bei jeweils fünf offenen Verbindungen unterschritten. Dabei ist zu beachten, dass Verbindungen nach ihrer Beendigung durch den Nutzer noch einige Zeit im „closing"-Zustand weiter existieren, also durch wiederholten Verbindungsauf- und -Abbau der Effekt verstärkt wird.

12. genauer: Transitions*instanzen* (siehe Abschnitt 3.3.7.8)

13. In dem oben beschriebenen einfachsten XTP-Szenario beinhalten die 18 Modulinstanzen bereits 580 Transitionsinstanzen, im Schnitt hat jede Modulinstanz also ca. 32 Transitionsinstanzen.

zum beschriebenen naiven Auswahlmodell – nicht die Modul- und Transitionsauswahl ausschließlich anhand des *Systemzustandes* durchzuführen (und somit in jeder Auswahlphase komplett von Neuem zu beginnen), sondern zur Optimierung möglichst auf *Zusatzinformationen* aus früheren Zyklen zurückzugreifen. Solche Zusatzinformationen zu gewinnen und einzusetzen ist der wesentliche Ansatzpunkt des im nächsten Abschnitt vorgestellten Modells.

4.2.1.2 Activity-Thread-Modell

Das *Activity-Thread-Modell* [Svo89] basiert auf der Annahme, dass Aktivität in der Spezifikation ausschließlich auf dem Transport und der Verarbeitung von Nachrichten (Interaktionen) beruht. In solchen Implementierungen bewegt sich während der Ausführung des Systems der Kontrollfluss zusammen mit dem zu transportierenden Datenpaket durch die einzelnen Teilsysteme.[14] Das Eintreffen einer Nachricht an einem externen Interaktionspunkt des (topologisch offenen) Systems startet einen neuen Activity-Thread. Dieser betreibt den Transport der Nachricht und der u. U. daraus abgeleiteten Nachrichten durch das System.

In [HeMiKö97] wird eine Abbildung des Activity-Thread-Modells auf die FDT Estelle entwickelt. Dabei mussten zur Überbrückung der konzeptionellen Differenzen zwischen dem semantischen Modell von Estelle und dem Activity-Thread-Modell folgende Einschränkungen an die Struktur der zu Grunde liegenden Spezifikation gemacht werden:

- Jede Transition ist eine Eingabetransition, d.h. sie nimmt zu Beginn ihrer Ausführung eine Interaktion an.
- Transitionen machen höchstens eine Ausgabe, die zudem nur als letzte Aktion erfolgen darf. Transitionen, die keine Ausgabe machen, oder deren Ausgabe die Systemgrenzen überschreitet, beenden den eingehenden Activity-Thread.[15]
- Jede Nachricht definiert eindeutig, welche Transition sie annehmen soll, d.h. es gibt keinen Indeterminismus bei der Transitionsauswahl.[16]

Die Hauptschleife einer Activity-Thread-Implementierung könnte damit wie in Abb. 4-3 dargestellt aussehen, wobei die Variable „*message*“ jeweils bereits direkt auf die nächste schaltbare Transition schließen lässt.

Eine globale Transitionsauswahl im eigentlichen Sinne findet somit nicht statt, sondern die Auswahl der nächsten zu schaltenden Transition erfolgt zusammen mit dem Transport der Nachricht (siehe Abb. 4-4).

---

14. Genauer wird die Verarbeitung durch eine Sequenz von Prozeduraufrufen realisiert, die in etwa jeweils einer Estelle-Transition entsprechen.
15. In [HeMiKö97] wird eine Erweiterung diskutiert, die mehrere Ausgaben in einer Transition erlauben soll. Da diese jedoch die eigentliche Idee des Activity-Thread-Modells aufgibt, wird sie hier nicht näher untersucht.
16. Die Behandlung von `PROVIDED`-Klauseln in Estelle-Spezifikationen erfordert auch im Activity-Thread-Modell eine (wenn auch auf die in Frage kommenden Transitionen beschränkte) explizite Transitionsauswahl.

```
while true do
    begin
        message := receive_external();
        while message <> EMPTY do
            message := fire_transition_for(message);
    end;
```

Abbildung 4-3: Grundprinzip des Activity-Thread-Modells

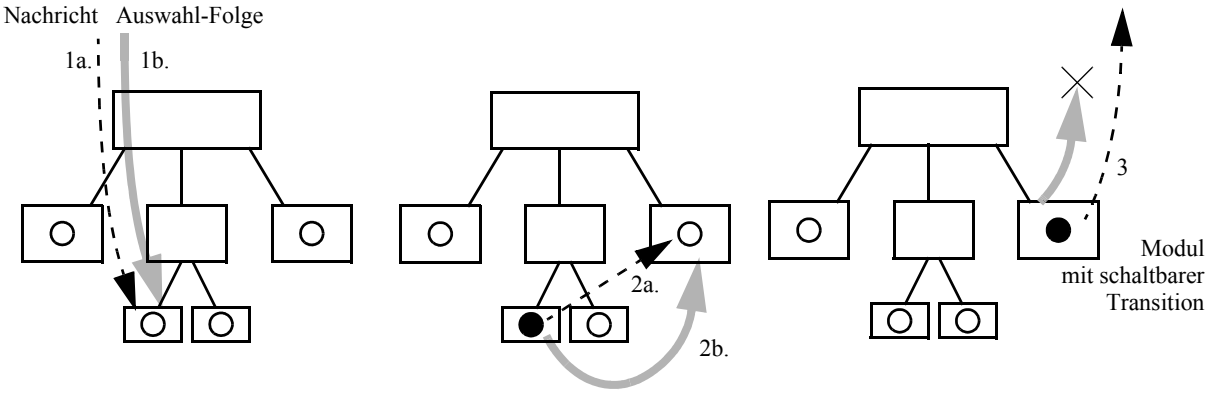


Abbildung 4-4: Auswahl- und Ausführungszyklen im Activity-Thread-Modell

Eine solche Activity-Thread-Implementierung wird im Vergleich zu einer einfachen Server-Implementierung meist eine höhere Performance bieten, da offensichtlich der Suchraum für die nächste zu schaltende Transition weitaus kleiner oder sogar minimal ist. Bei einem laufenden Activity-Thread ergibt sich bezüglich der Suche nach dem Modul mit der nächsten schaltbaren Transition idealerweise eine Auswahleffizienz[17] von 100%.

Die Ursache für diese hohe Effizienz ist, dass die Activity-Thread-Implementierung zwischen den Systemzyklen Informationen über die jeweils nächste zu feuernde Transition aus der vorangegangenen gewinnt und somit die eigentliche Transitionsauswahl entfällt.

Um das Activity-Thread-Modell insbesondere auf Estelle anwenden zu können, sind jedoch neben den oben genannten noch einige weitere Einschränkungen des Spezifikationsstils nötig:

- Da die Vater-Sohn-Priorität der Estelle-Semantik die vorab-Bestimmung der nächsten Transition behindert, wird diese ausgeschlossen, indem aktive[18] Module auf Blattmodule beschränkt sind. Es ist jedoch zu beachten, dass dadurch aber auch keine dynamische Modulstruktur mehr möglich ist!
- Der Einsatz der `ACTIVITY`-Modulattributierung ist zwingend.
- Es gibt keine spontanen Transitionen oder Transitionen mit `DELAY`-Klauseln, da deren Feuern nicht durch den Empfang einer Nachricht angestoßen wird.

17. Soweit dieser Begriff noch sinnvoll definiert ist, da ja keine eigentliche Auswahl stattfindet. In gewissem Umfang kann jedoch die Verwaltung des Activity-Threads als Auswahloperation betrachtet werden, wodurch sich eine Modulauswahleffizienz von 100 % ergibt.

18. also Module, die Transitionen enthalten (können)

Ziel dieser Beschränkungen ist es, die der Estelle-Semantik zu Grunde liegenden Synchronisationsregeln (speziell die Vater-Sohn-Priorität) und die damit verbundenen Probleme zu umgehen, indem die Voraussetzungen für deren Anwendung ausgeschlossen werden: Durch den Zwang, Transitionen nur in Blattmodulen anzulegen, ist keine explizite Vater-Sohn-Synchronisation nötig, und durch das Verbot der `PROCESS`-Attributierung ist keine Synchronisation zwischen Geschwistermodulen nötig.

Offensichtlich bewirken die zur Anwendung des Activity-Thread-Modells notwendigen Restriktionen eine ganz erhebliche *Beeinträchtigung der Ausdrucksfähigkeit* von Estelle. So ist insbesondere eine Darstellung dynamischer Strukturen, wie sie z. B. in der Estelle-Spezifikation zu XTP 4.0 (siehe Abschnitt 2.3.4) zu finden ist, nicht effektiv möglich. Die Spezifikation enthält in ihrer Urfassung ein komplettes Kommunikationsszenario mit Benutzer- und Netzwerkmodulen (siehe Abb. 4-5). Der wesentliche Aspekt ist die Modellierung der *Kontextmodule*, die in XTP die Endpunkte einer jeden Verbindung darstellen, als Modulinstanzen. Dies bedingt die Notwendigkeit einer dynamischen Erzeugung und Vernichtung dieser Kontextmodule und damit auch von Transitionen in deren Vatermodul, der Protokollinstanz. Somit ist die problemorientierte Spezifikation von XTP aufgrund ihrer dynamischen Struktur nicht mit dem Activity-Thread-Modell implementierbar.

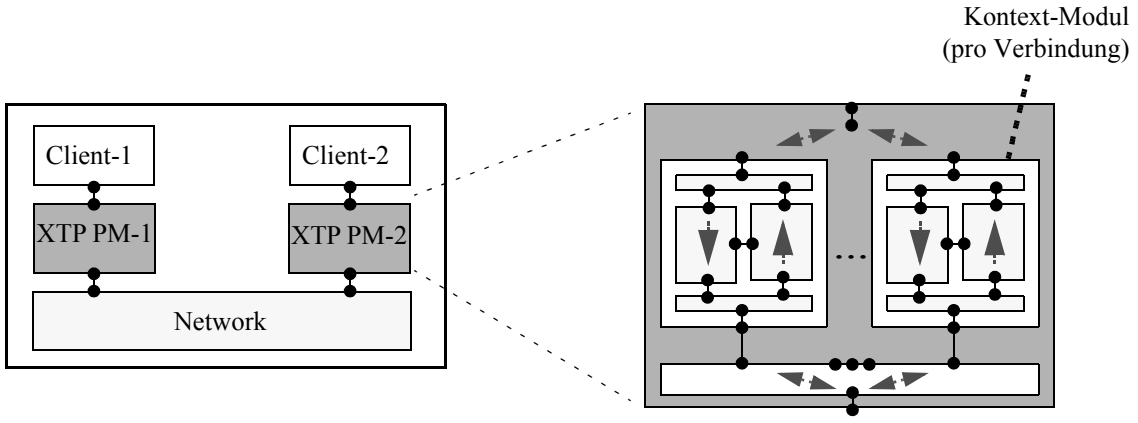


Abbildung 4-5: Modul- und Verbindungsstruktur einer Estelle-Spezifikation von XTP

Aber auch andere wichtige Protokollabläufe sind im Activity-Thread-Modell nicht ohne weiteres abbildbar. So resultiert aus dem Zwang zur Paarung jeweils genau eines Empfangs- und eines Sendeereignisses in jeder Transition, dass wichtige Operationen wie Unterbrechungen im Nachrichtentransport[19] oder Verzweigungen der Nachrichtenwege[20] nicht modelliert werden können. Auch ist die Determiniertheit der Nachrichtenverarbeitung nur in einfachen Protokollen vorzufinden, die nur eine geringe Dynamik und Parametrierbarkeit der Protokollabläufe beinhalten.

Wir werden im nächsten Abschnitt eine Familie von Implementierungsmodellen vorstellen, die basierend auf dem Server-Modell die Performance-Vorteile einer Activity-Thread-Implementierung liefern, ohne deren strenge Restriktionen zu erzwingen. Diese Modelle bilden die Basis für das Ausführungsmodell der von XEC generierten Implementierungen.

19. z. B. zur Zwischenpufferung und Entkopplung von Sende- und Empfangsströmen; in der XTP-Spezifikation werden die zu übertragenden Nutzdaten (als Bytesequenz) dazu in einem Ringpuffer zwischengespeichert und u. U. erst verzögert verschickt.

20. z. B. zur Quittierung eines vom Netzwerk empfangenen Paketes an den Absender und gleichzeitigen Weitergabe der Nutzlast an den Empfänger (erforderlich für alle Transportschichtprotokolle) oder zur Ratenkontrolle; beide Aspekte treten in der XTP-Spezifikation auf

4.2.2 Ereignisgesteuerte Implementierungsmodelle

Wie wir gesehen haben, basiert das *Server-Modell* genau wie die Estelle-Semantik auf der Auswertung des aktuellen Systemzustands zur Bestimmung einer (zulässigen oder notwendigen) Folgeaktion, wie dem Feuern einer Transition. Das Server-Modell in seiner Reinform berücksichtigt dabei nicht, *wie* dieser Systemzustand erreicht wurde. Entsprechend muss in einer sequentiellen, prozessorientierten Implementierung im Allgemeinen der gesamte Modulbaum nach diesen schaltbaren Transitionen durchsucht werden.

Das *Activity-Thread-Modell* geht den umgekehrten Weg und definiert die nächste zu schaltende Transition bei der Abwicklung eines Threads ausschließlich über die Vorgeschichte – mehr noch ist mit dem Start des Threads der gesamte Ablauf und damit die Folge der zu schaltenden Transitionen festgelegt. Diese erheblich Vereinfachung bei der Transitionsauswahl setzt jedoch erhebliche stilistische Restriktionen bei den zu Grunde liegenden Spezifikationen voraus, um konsistent mit der Estelle-Semantik zu bleiben. Wie kritisch diese Einschränkungen sind, kann man z. B. daran sehen, dass die Spezifikation gänzlich ihrer strukturellen Dynamik beraubt wird und so z. B. wie oben gezeigt, eine problemorientierte Spezifikation von XTP unmöglich macht.

Wir untersuchen nun, wie die Vorteile des *zustandsgesteuerten* (und damit Semantik-nahen und universellen) Server-Modells und des vorgeplanten (und damit effizienten, aber nur eingeschränkt nutzbaren) Activity-Thread-Modells kombiniert werden können, indem die Ereignisse zum Erreichen des aktuellen Systemzustands bei der Transitionsauswahl mit berücksichtigt werden, sie also (ganz oder teilweise) *ereignisgesteuert* abläuft. Die Kombination beider Steuerungsmodelle führt uns zu dem im folgenden Abschnitt dargestellten *hybriden Implementierungsmodell* von XEC.

4.2.2.1 Ein hybrides Implementierungsmodell

Das von XEC eingesetzte Implementierungsmodell ist im Grunde ein Server-Modell und setzt daher auch keine stilistischen Beschränkungen der zu implementierenden Spezifikation voraus. Es nutzt jedoch u. a. die Grundidee des Activity-Thread-Modells aus, die Ausführung der Spezifikation durch den Datentransport zu steuern und so die Transitionsauswahl zu optimieren. Die Anwendung dieser Methoden wird zwar durch die Einhaltung gewisser Stilregeln bei der Spezifikation erleichtert, diese sind jedoch keine zwingende Voraussetzung, d.h. die Effizienzsteigerung ist *über den Grad der Einhaltung dieser Regeln skalierbar*.

In seiner Eigenschaft als Server-Modell basiert es zunächst auf einem globalen Auswahlmechanismus, der rekursiv die Modulhierarchie durchläuft und Module mit schaltbaren Transitionen sucht. Anschließend werden diese Transitionen gemäß der Estelle-Semantik gefeuert, und ein neuer Auswahlzyklus beginnt. Dieser Auswahlmechanismus wird jedoch wesentlich abgekürzt, wenn Informationen aus vorherigen Zyklen genutzt werden können, um die globale und modullokale Auswahl zu vereinfachen. Dabei stehen im Wesentlichen zwei Fragen im Vordergrund:

(i) *Welche Modulinstanz beinhaltet eine schaltbare Transition?*
(Globale Auswahl)

(ii) *Welche Transitionen innerhalb dieser Modulinstanzen sind schaltbar?*
(Modullokale Auswahl)

Bei beiden Fragestellungen kann der Einsatz von Informationen aus vorangegangenen Auswahl- und -Ausführungszyklen eine erhebliche Verkleinerung des Suchraums bewirken. Basieren diese Informationen auf Ereignissen[21] während einer vorangegangenen Ausführungsphase, so bezeichnen wir das Auswahlverfahren (zumindest teilweise) als *ereignisgesteuert*.[22]

Das in XEC eingesetzte Verfahren zur Transitionsauswahl nutzt bei der *globalen Auswahl* u. a. die Grundidee des Activity-Thread-Modells aus, indem hier zuerst diejenigen Module auf schaltbare Transitionen untersucht werden, in denen in den vorangegangenen Zyklen ein (von außen induziertes) Ereignis wie z. B. der Empfang einer Nachricht stattgefunden hat. Wird in einem solchen Modul eine schaltbare Transition gefunden, so kann diese (unter Einhaltung der Estelle-Synchronisationsregeln) direkt geschaltet werden. Dies führt dazu, dass bei einer Kette von Paketbearbeitungs- und Weitertransportaktionen, wie sie z. B. beim Activity-Thread-Modell ja vorausgesetzt wird, unmittelbar diejenigen Module von der Transitionsauswahl erfasst werden, die auch die nächste schaltbare Transition enthalten. Zusammen mit einer effektiven modullokalen Transitionsauswahl (Abschnitt 4.3), kann somit eine mit dem Activity-Thread-Modell vergleichbare Auswahleffizienz erreicht werden.

Betrachten wir dazu in Fortsetzung der obigen Beispiele das in Abb. 4-6 dargestellte Szenario, bei dem (analog zu einer der in Abschnitt 4.2.1.2 genannten Anforderungen an die Anwendbarkeit des Activity-Thread-Modells) ausschließlich Blattmodule aktiv[23] sein können und entsprechend als Systemmodule (schraffiert dargestellt) spezifiziert sind.

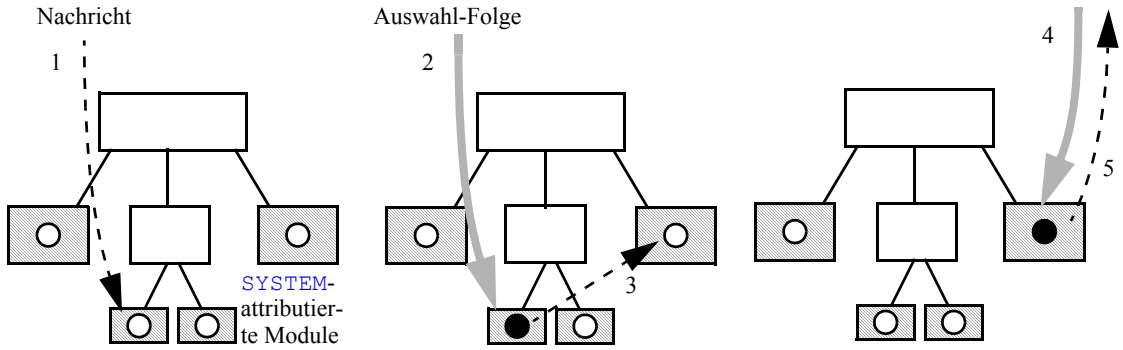


Abbildung 4-6: Auswahl- und Ausführungszyklen im hybriden Ausführungsmodell von XEC

Das oben beschriebene Activity-Thread-Modell hätte hier aufgrund vorab festgelegter Sequenzen die Bearbeitung der eingehenden Nachricht in den beteiligten Modulen abgewickelt. In unserem Server-basierten Modell stehen solche Vorabfestlegungen nicht zur Verfügung bzw. sind zur Erhaltung der Flexibilität auch nicht gefordert. Entsprechend muss eine effiziente Folge von zu testenden Modulen *dynamisch* ermittelt werden.

---

21. Solche Ereignisse sind z. B. der Transport einer Nachricht oder die Änderung eines Kontrollzustandes. Ereignisse sind streng zu unterscheiden von ihrem Ergebnis, letztlich also dem neuen Systemzustand.

22. Im Gegensatz zu den *zustandsgesteuerten Auswahlverfahren*, die im Extremfall ausschließlich den Zustand zum Zeitpunkt der gerade laufenden Auswahlphase berücksichtigen.

23. Wir verwenden den Begriff „*aktiv*“ in diesem Kapitel zur Vermeidung von Überschneidungen nicht im Sprachgebrauch des Estelle-Standards statisch für ein unattributiertes, transitionsloses Modul, sondern spezifischer für eine Modulinstanz, die auch eine schaltbare Transition anbieten kann.

Das Auswahlverfahren von XEC favorisiert dazu gezielt Modulinstanzen, in denen in früheren Ausführungszyklen *Ereignisse* aufgetreten sind, die möglicherweise das Schaltbarwerden einer Transition bewirken können. Dies ist z. B. dann der Fall, wenn eine Nachricht zu einem Modul übertragen und in eine Warteschlange eingereiht wird, die zuvor leer war. Eine solche Nachricht könnte im nächsten Auswahlzyklus von einer entsprechenden Eingabetransition[24] des Moduls verarbeitet werden.

Ob dies tatsächlich möglich ist, hängt davon ab, ob neben der `WHEN`-Klausel auch alle anderen Schaltbedingungen erfüllt sind und keine Transitionen mit höheren Prioritäten (innerhalb des Moduls oder in Vater-Modulen) vorliegen. Man kann jedoch beobachten, dass in typischen Kommunikationsprotokollspezifikationen fast alle Komponenten (Modulinstanzen) zumindest lokal ununterbrochen *reaktiv* sind und eine eingehende Nachricht entsprechend beim Empfang sofort weiterverarbeiten können. Da dieser Effekt auch aus einer konzeptionellen Sicht der Natur reaktiver Protokollkomponenten entspricht, legen wir ihn als eine *heuristische Annahme* dem Optimierungsmodell und unseren weiteren Überlegungen zu Grunde. Wir überprüfen diese Annahme später anhand verschiedener Test- und Benchmarkspezifikationen.

In unserem Beispielszenario aus Abb. 4-6 gibt es aufgrund der Modulattributierung keine Prioritäten zwischen den Modulen, und die Transitionsauswahl kann im jeweils nächsten Zyklus sofort mit dem Modul beginnen, das im vorangegangenen Zyklus eine Nachricht in der beschriebenen Weise empfangen hat. Die *Modul-Auswahlsequenz folgt also dem Nachrichtentransport* und es bildet sich so die selbe Sequenz, wie wir sie bereits in Abb. 4-4 bei der Activity-Thread-Implementierung des selben Kommunikationsszenarios gesehen haben.

In der Tat erreicht dieses ereignisgetriebene Auswahlverfahren bei Spezifikationen, die den in Abschnitt 4.2.1.2 für das Activity-Thread-Modell beschriebenen Restriktionen genügen, eine Modulauswahleffizienz von 100%. Im Gegensatz jedoch zum Activity-Thread-Modell ist das in XEC eingesetzte Auswahl- und Ausführungsmodell ein *universelles Implementierungsmodell*, d.h. es ist auch bei Systemen anwendbar, die diese strengen Restriktionen nicht erfüllen: Der Auswahlmechanismus stellt sicher, dass auch bei Situationen, in denen die Estelle-Synchronisierungs- und Auswahlregeln zusätzliche (vorrangige) Transitionsauswahlen erfordern, diese eingehalten werden und dennoch die gewonnenen Informationen aus früheren Auswahl- und Ausführungszyklen zur Optimierung genutzt werden können.

Modifizieren wir also das bisherige Beispielszenario z. B. so, dass aktive Module nicht nur auf Blattebene vorkommen,[25] so sind die Voraussetzungen für das Activity-Thread-Modell nicht mehr erfüllt (siehe mittleres Modul in Abb. 4-7). Empfängt eines der so entstandenen nicht `SYSTEM`-attributierten Kindmodule eine Nachricht in eine leere Warteschlange (1), so wird gemäß der oben beschriebenen Auswahlheuristik die Modulauswahl auf dieses Kindmodul abzielen. Gemäß der Vater-Sohn-Priorität werden dabei jedoch zunächst die aktiven Vatermodule auf Schaltbarkeit getestet (2a), bis schließlich das eigentliche Auswahlziel erreicht wird (2b).

Diese Vorgehensweise garantiert die Einhaltung aller von der Estelle-Semantik vorgegebenen Prioritäten und erreicht gleichzeitig die Auswahl mit dem notwendigen Minimum an Modultests. Dabei *skaliert* die Wirksamkeit des Auswahlverfahrens mit dem Grad der Einhaltung bestimmter Spezifikationsstilregeln, die ähnlich zu den Vorbedingungen des Activity-Thread-Modell sind, erzwingt ihre Einhaltung jedoch nicht:

---

24. Die Transition muss dazu u. a. eine `WHEN`-Klausel besitzen, die den Interaktionspunkt und den Typ der Nachricht referenziert. Es ist zu beachten, dass in Estelle verschiedene Interaktionspunkte eine gemeinsame Warteschlange nutzen können (siehe auch Abschnitt 3.3.7.6).

25. Entsprechend sind auch nicht mehr alle Blattmodule `SYSTEM`-attributiert.

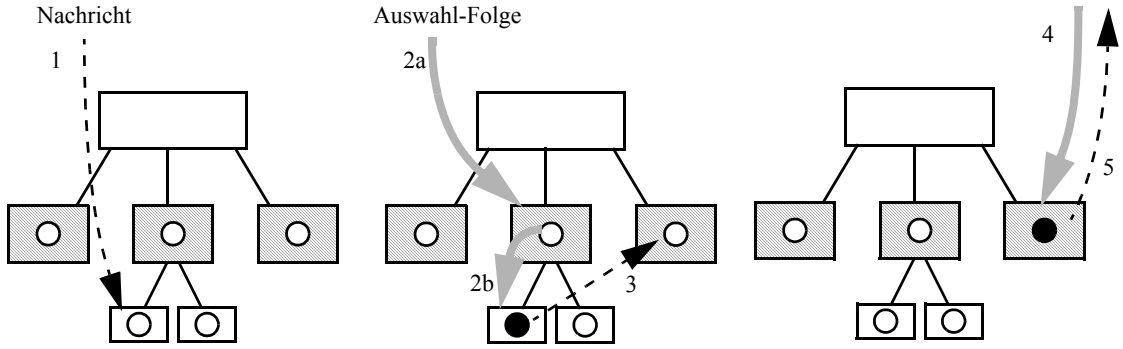


Abbildung 4-7: Vater-Sohn-Priorität im hybriden Ausführungsmodell von XEC

- Werden sie *voll* eingehalten (Abb. 4-6), so ergibt sich eine Modulauswahleffizienz von 100%.
- Werden sie nur *teilweise* eingehalten, so sinkt die Effizienz (z. B. auf 67% in Abb. 4-7[26]), was gegenüber dem reinen Server-Modell (20%-25% Effizienz,[27] siehe Abb. 4-2) jedoch immer noch eine *deutliche Verbesserung* bedeutet. (Ein Einsatz des Activity-Thread-Modells wäre in diesem Fall überhaupt nicht möglich.)
- Bei völliger Missachtung der Spezifikationsregeln konvergiert das hybride Verfahren schließlich in seinem Verhalten und seiner Auswahleffizienz gegen ein konventionelles Server-Modell.

In den folgenden Abschnitten verfeinern wir die Eigenschaften des Auswahlverfahrens und des förderlichen Spezifikationsstils, entwickeln eine effiziente Implementierung und ergänzen diese um zusätzliche Mechanismen zur Steigerung ihrer Wirksamkeit. Parallel dazu evaluieren wir jeweils die einzelnen Verfahren anhand von Benchmarkspezifikationen.

4.2.2.2 Ereignissteuerung der Modulauswahl

Wir beginnen mit der Frage, welche externen[28] Ereignisse im Sinne des oben beschriebenen hybriden Auswahlmodells in einem Modul eine Transition schaltbar machen können.

Die *(potentiell) aktivierenden externen Ereignisse* zu einem Modul sind:

(i) Der *Empfang einer Nachricht* in eine zuvor leere Warteschlange,

(ii) der *Ablauf des Timers* einer `DELAY`-Transition,

(iii) die *Manipulation einer exportierten Variable* durch das Vatermodul und

(iv) die *Manipulation einer* durch ein Kindmodul *exportierten Variablen*.

---

26. Es wurden 3 Module getestet und 2 Transitionen gefeuert.

27. Prüft man direkt über die Subsysteme, so können in der oben angenommenen Auswahlsequenz des Server-Modells 2 Transitionstests eingespart werden und es ergibt sich eine Verbesserung der Auswahleffizienz von 20% auf 25%.

28. also von einer anderen Modulinstanz oder (bei Berücksichtigung offener Systeme wie beim Activity-Thread-Modell) von einer externen Umgebung ausgehend

Alle anderen Ereignisse, welche die Schaltbarkeit einer Transition des Moduls beeinflussen, müssen von diesem Modul selbst ausgehen und können daher nur zusammen mit dem Feuern einer Transition in dem Modul auftreten. Eine Ausnahme bilden lediglich indeterministische Ausdrücke und Schaltbedingungen,[29] die prinzipiell auch ohne Stimulus ihr Ergebnis ändern könnten. Hier gilt jedoch, dass der Indeterminismus insbesondere auch das Nicht-Ändern der Ergebnisse beinhaltet und somit von der Indeterminiertheit dieser Ausdrücke abgesehen werden kann.

Für unseren hybriden Auswahlalgorithmus sind prinzipiell alle vier o. g. Ereignistypen relevant. Zum Zwecke der Activity-Thread-Nachbildung ist jedoch Punkt *(i)* von zentraler Bedeutung, da Transport und Empfang einer Nachricht gerade die treibende Kraft dieses Modells war. Die Einschränkung, dass nur der Empfang in eine leere Warteschlange relevant ist, beruht auf der strikten FIFO-Eigenschaft von Warteschlangen in Estelle, durch die alle nachfolgenden Nachrichten in der Warteschlange für das Empfängermodul zunächst unsichtbar bleiben und damit auf die Schaltbarkeit keinen Einfluss haben.[30]

Der Ablauf eines Timers *(ii)* ist ebenfalls ein (bzgl. des hybriden Auswahlmodells) relevantes Ereignis, da es in vielen Protokollspezifikationen einen protokollbezogenen Timeout modelliert und daher häufig zur Emission einer neuen Nachricht[31] und damit u. U. zu einer neuen Ereigniskette führt.

Zwei wesentlich schwieriger zu handhabende Ereignisse sind die Manipulationen von Variablen durch andere Module. Hier kommen exportierte Variablen des Moduls selbst *(iii)* und die seiner unmittelbaren Kindmodule *(iv)* in Frage. Ein typisches Beispiel sind Flag-Variablen, über die ein Modul seinem Vatermodul einen bestimmten Zustand anzeigt.[32] Werden diese exportierten Variablen in einer Schaltbedingung[33] einer Transition referenziert, so kann die Schaltbarkeit des entsprechenden Moduls von außen manipuliert werden.

Da ändernde Zugriffe auf exportierte Variablen zudem in den untersuchten Spezifikationen nur wenig Relevanz haben,[34] berücksichtigen wir die Punkte (iii) und (iv) zunächst in der Implementierung des hybriden Auswahlmodells nicht. Dies hat – wie auch alle anderen aufgeführten Ausgestaltungen der Auswahlheuristik – auf die *Korrektheit* der Implementierung keinen negativen Einfluss, da die Heuristiken nur dazu genutzt werden, Freiräume innerhalb der Grenzen des semantischen Modells von Estelle zu steuern.

---

29. z. B. „`FORONE b: boolean SUCHTHAT true DO random := b;`“ kann ohne Änderung des Systemzustandes bei jedem Aufruf spontan einen anderen Wert an `random` zuweisen.

30. z. B. in SDL [ITU94] ist u. a. durch das SAVE-Konstrukt die FIFO-Eigenschaft nicht zwingend erfüllt.

31. z. B. der Wiederholung eines verloren gegangenen Paketes in einem Transportschichtprotokoll

32. So exportieren die Kontext-Module der XTP-Protokollmaschine die `boolean`-Variable „`abort_context`“, über die das Kontextmodul seinem Vatermodul die Bereitschaft anzeigt, terminiert zu werden. Entsprechend existiert in diesem Vatermodul eine Transition, die durch das Setzen dieses Flags aktiviert wird.

33. Die Zugriffe können dabei auch indirekt über Funktionsaufrufe erfolgen. Es muss sich zudem bei den referenzierenden Klauseln nicht zwingend um die `PROVIDED`-Klausel handeln. Auch ein Interaktionspunk-Array-Index einer `WHEN`-Klausel oder die Parameter einer `DELAY`-Klausel können solche exportierten Variablen referenzieren. (Letzteres ist unkritisch, da die `DELAY`-Parameter nur beim Start des Timers evaluiert werden müssen.)

34. Sie kommen nur in der Estelle-Spezifikation von XTP 4.0 vor und dienen dort lediglich dem Verbindungs- und Systemmanagement, sind also an regulären Datenübertragungen kaum beteiligt.

Betrachten wir dazu ausgehend von der Referenzimplementierung des globalen Auswahlprozesses die Implementierung der ereignisgesteuerten Auswahlheuristik in XEC.

Wie wir in Abschnitt 3.4.3 gesehen haben, enthält jede Modulinstanz (also jede Instanz der Klasse `Module`) mit der Komponente `Module::Children` eine Liste von Kindmodulinstanzen[35] (siehe Abschnitt 3.3.2.1). Ausgehend von der Spezifikationsmodulinstanz ergibt sich somit der Modulinstanzbaum des Systems. Die globale Auswahl erfolgt in der vorgestellten Referenzimplementierung rekursiv entlang dieser Baumstruktur (siehe Abschnitt 3.4.3). Dabei wird eine Tiefensuche nach einem Modul mit schaltbaren Transitionen ausgeführt. Bei Erfolg wird die Rekursion für den darunter liegenden Teilbaum abgebrochen, da aufgrund der Vater-Sohn-Priorität eine solche Suche keine Auswirkung auf das Auswahlergebnis haben könnte. Zudem wird bei Modulen, die nicht `PROCESS`-attributiert sind, auch die Suche auf dieser Ebene abgebrochen, da nur ein Modul ausgewählt werden muss.[36] Dieses Verfahren implementiert zunächst ein *einfaches Server-Modell* (siehe Abschnitt 4.2.1.1 und Abb. 4-2 auf Seite 124).

Zur Implementierung eines ereignisgesteuerten Auswahlverfahrens benötigen wir nun eine Methode, um die globale Transitionsauswahl mit einer möglichst geringen Anzahl von zu testenden Modulen zu dem Modul zu führen, in dem das Ereignis stattgefunden hat (das also z. B. eine Nachricht empfangen hat). Eine kompakte Lösung auf Basis der o. g. Referenzimplementierung besteht darin, das betroffene Modul und seine Vatermodule in den jeweiligen sie enthaltenden Listen zum vordersten (und damit zum jeweils zuerst getesteten) Listenelement zu machen. Dadurch wird die Tiefensuche eine Modultestsequenz liefern, die vor dem angestrebten Modul lediglich nur die absteigende Folge seiner (direkten oder indirekten) Vatermodule enthält. Unter Berücksichtigung der Vater-Sohn-Priorität ist diese Sequenz für das angestrebte Ziel optimal.

Die in Beispiel 4.27 auszugsweise[37] wiedergegebene Methode `Module::event()` implementiert diesen Algorithmus, indem ausgehend vom aktuellen Modul in der Modulhierarchie bis zum Spezifikationsmodul aufgestiegen wird (Schleife ab Zeile 2) und jeweils durch Aufruf der Methode `makeFirst()` die Verschiebung an den Anfang der Kindmodulliste getätigt wird (Zeile 4).[38]

Beispiel 4.27: Methode `Module::event()` zur ereignisgesteuerten Auswahl (Auszug)

```
1: void Module::event(bImportant = true) {
2:         for (Module* p = this ; p ; p = p->pParent) {
3:                 if (bImportant && Module::getOptMessageDriven())
4:                         p->makeFirst();
5:         }
6: }
```

(Ende von Beispiel 4.27)

35. Da die Klasse `Module` durch Vererbung auch ein `XListNode<Module>` ist, können ihre Instanzen direkt in diese Liste eingekettet werden.

36. In ACTIVITY-attributierten Modulen kann nur ein schaltbares Modul ausgewählt werden. Da zwischen den Geschwistermodulen keine Prioritäten bestehen, kann die Suche beim ersten Treffer abgebrochen werden. In PROCESS-attributierten Modulen müssen alle schaltbaren Module ausgewählt werden. Daher muss die Suche fortgesetzt werden. In unattributierten Modulen gelten die Nebenläufigkeitsregeln zwischen Subsystemen, es kann also u. a. jede Zwischenform gewählt werden.

37. Der Parameter `bImportant` wird erst in den unten folgenden Verfeinerungen sinnvoll eingesetzt.

38. `makeFirst()` stammt aus der Basisklasse `XListNode<Module>`

Aufgerufen wird die Methode `event()` bei verschiedenen Ereignissen, durch die eine Transition eines Moduls schaltbar werden könnte (s. o.). Im Vordergrund steht hier der Empfang einer Nachricht in eine zuvor leere Warteschlange, die durch Aufruf der in Beispiel 4.28 wiedergegebenen Methode `InterActionAbstract::updateDest()` realisiert wird.[39]

Beispiel 4.28: Aufruf von `Module::event()` wg. Nachrichtenempfangs

```
1: void InterActionAbstract::updateDest() {
2:         pDest = pOrgDest->lowerEndpoint();
3:         if (pDest->queue().isEmpty())
4:                 pDest->event();
5:         appendTo(&pDest->queue());
6: }
```

(Ende von Beispiel 4.28)

Das Ablaufen eines Timers einer `DELAY`-Transition führt ebenfalls über die Methode `handle_expiration()`[40], zum Aufruf der `event`-Methode.

Beispiel 4.29: Aufruf von `Module::event()` wg. Delay-Timer-Ablauf

```
1: void DelayedTrans::DelayTimer::handle_expiration() {
2:         pTrans->module()->event();
3: }
```

(Ende von Beispiel 4.29)

Diese ersten relativ geringfügigen Erweiterungen erfüllen bereits unsere obigen Zielsetzungen bzgl. einer effizienten ereignisgesteuerten Auswahlheuristik.

Wir untersuchen die Wirkung der Optimierungsmaßnahmen anhand des in Abschnitt 2.3.2 eingeführten „*Ping-Pong*“-*Benchmark*s (siehe Abb. 4-8), bei dem zwei Dienstnutzer („`Usr`“ Module) jeweils über einen dreistufigen Protokollstack (Module „`PM_1`“ bis „`PM_3`“) und ein vermittelndes Netzwerk (Module „`Net`“ und „`Router`“) mit einer parametrierbaren Anzahl von „Hops“ Nutzdaten (als Parameter von Sendeaufträgen) austauschen können. Alle diese Module sind direkte Kindmodule des (`SYSTEMACTIVITY`-attributierten) Spezifikationsmoduls.

Die Dienstnutzer senden nun – wie der Name des Benchmarks vermuten lässt – in einem Ping-Pong-Muster Nachrichten von Ende zu Ende der Kommunikationsstrecke, die Übertragung erfolgt dabei verlustfrei und unverzögert. Die dazu erstellte Estelle-Spezifikation wird später in Abschnitt 6.2.4 nochmals in fast identischer Form Verwendung als Datenübertragungsbenchmark finden (siehe auch Anhang B.1).

Wir verwenden hier folgende Parameter für den Benchmark:

- Anzahl Hops (*H*): 5
- Anzahl der Ping-Pong-Durchläufe: 100
- Anzahl der PDUs im „`Resend_Buffer`“[41]: 3
- Größe von „`T_UserData`“: 10

39. Neben dem Empfang einer im letzten Schaltzyklus neu erzeugten Nachricht kann dies auch durch ein `attach`- oder `detach`-Statement bewirkt werden.

40. Sie wurde zur Abstraktion der internen Aspekte des Timer-Managements (u. a. Klasse `DelayedTrans::DelayTimer`) und der Auswahloptimierung (Klasse `Module`) eingeführt.

41. Nur relevant bei der Nutzung als Datenübertragungsbenchmark (siehe Abschnitt 6.2.4).

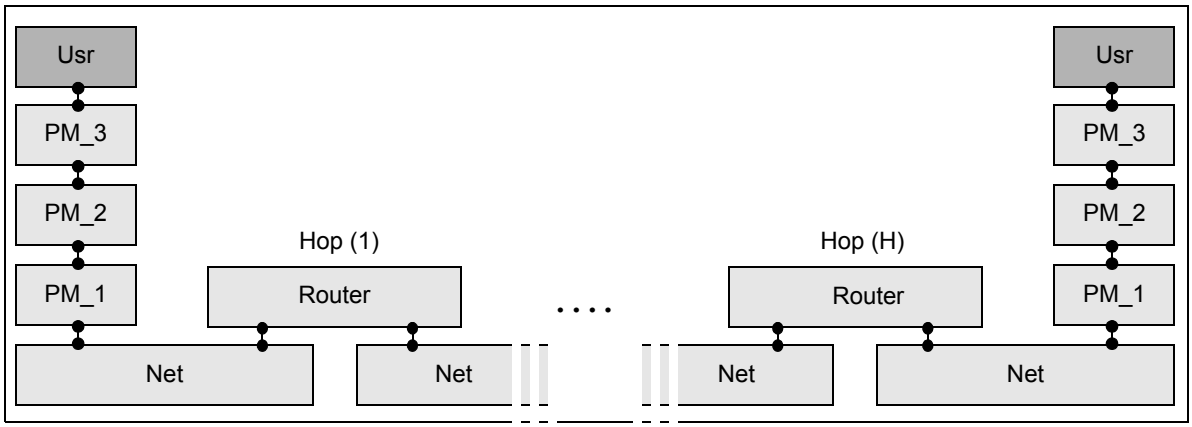


Abbildung 4-8: Modulinstanz- und Verbindungsstruktur des Ping-Pong-Benchmarks

In dem System existieren in dieser Konfiguration 20 Modulinstanzen (einschließlich des Systemmoduls), und jede Ende-zu-Ende-Übertragung (kompletter Umlauf) erfordert insgesamt 38 Auswahl- und Ausführungszyklen (jeweils eine schaltbare Transition pro Zyklus).

Das Basisauswahlverfahren erfordert zur Auswahl der insgesamt 3.800 schaltbaren Transitionen den Test von 41.885 Modulen. Die Auswahleffizienz beträgt also 9,1 % und entspricht somit recht gut dem Erwartungswert: In dieser sehr flachen Hierarchie wird in jedem Umlauf das Spezifikationsmodul und anschließend im Mittel die Hälfte der 19 Kindmodule durchlaufen. Es ergeben sich im Mittel $1 + 19/2 = 10,5$ zu testende Module pro Zyklus, also eine Effizienz von 9,5 %.[42]

Im Vergleich dazu kann die nachrichtengesteuerte Auswahl wesentlich zielgerichteter dasjenige Modul ansteuern, welches im vorangegangenen Zyklus jeweils eine Nachricht empfangen hat und damit aktiviert wurde. Zur Auswahl der insgesamt 3.800 schaltbaren Transitionen werden nun nur noch 7.623 Module getestet, die Auswahleffizienz beträgt also knapp[43] 50 % und ist somit um den Faktor 5,5 besser als im einfachen Server-Modell (siehe Tabelle 4.5).

Tabelle 4.5: Modulauswahleffizienz Ping-Pong-Benchmark (ACTIVITY)

Auswahlverfahren	getestet	gefeuert	Effizienz
Basis[a]	41.714	3.800	9,1 %
ereignisgesteuert[b]	7.623	3.800	49,8 %

a. Laufzeit-Optionen: -notreesleep -noimodopt **-eventdrive 0**
b. Laufzeit-Optionen: -notreesleep -noimodopt **-eventdrive 1**

42. Die Abweichung vom Erwartungswert ergibt sich daraus, dass ausgewählte Module aus der Liste `Module::Children` ausgekettet und nach dem Feuern wieder eingekettet werden (siehe Abschnitt 3.4.3). Dadurch ändert sich die Test-Reihenfolge der Kindmodule im Verlauf der Ausführung, und die Auswahl ist nicht mehr völlig unkorreliert zur Ausführung.

43. Die Effizienz hat mit steigender Zahl von Zyklen tatsächlich genau 50% als Grenzwert. Die hier aufgetretenen 23 zusätzlichen Tests beruhen auf dem Initialisierungs- und Terminationsoverhead, bei dem die erste schaltbare Transition beim Systemstart bzw. die Nichtschaltbarkeit aller Transitionen zur Deadlockerkennung und Termination eine (bzgl. der Anzahl von Zyklen) feste Zahl zusätzlicher Tests erfordert.

Der Effekt, dass pro Auswahl jeweils zwei Module getestet werden mussten, ist mit der Tatsache zu erklären, dass das Spezifikationsmodul attributiert ist und daher zur Wahrung der Vater-Sohn-Priorität das Vater-Modul jeweils vor dem Kindmodul getestet wird. Könnte dies vermieden werden, so könnte die Effizienz auf fast 100 % gesteigert werden. Wir untersuchen in den nächsten Abschnitten, wie dieses Ziel erreicht werden kann.

4.2.2.3 Senkung der Synchronisation zwischen Modulinstanzen in Standard-Estelle

Offensichtlich ist die Effizienz des ereignisgesteuerten Auswahlverfahrens davon abhängig, ob das zur Auswahl favorisierte Modul auch auf kürzestem Wege angesteuert werden kann. Dem stehen in Estelle jedoch zwei Synchronisationsaspekte im Wege:

(i) zwischen Vater- und Sohn-Modulen (Vater-Sohn-Priorität)

(ii) zwischen Geschwistermodulen (innerhalb `PROCESS`-attributierter Vatermodule)

Im obigen Beispiel beeinträchtigte die Attributierung des Spezifikationsmoduls und die resultierende Vater-Sohn-Priorität *(i)* die Effizienz des Auswahlverfahrens auf 50 %, da jeweils zusammen mit dem angestrebten Modul *zusätzlich* immer auch *alle Vatermodule* getestet werden mussten. Dies umfasst aufgrund der flachen Hierarchie hier nur das Spezifikationsmodul. Allgemein sind es bei einem Modulbaum mit der Höhe h maximal $h \angle 1$ Vatermodule (im obigen Beispiel mit $h = 2$ also $2 \angle 1 = 1$).

Die Wirkung der Synchronisation zwischen Geschwistermodulen *(ii)* kann hingegen deutlich größer sein, wie man durch entsprechende Umattributierung von `ACTIVITY` auf `PROCESS` zeigen kann. Hierbei müssen in jedem Zyklus, der zu einer Auswahl in einem der Blattmodule führt, auch alle Geschwistermodule getestet werden.

Allgemein müssen bei einem vollständigen Modulbaum vom Grad n und der Höhe h mindestens $n \cdot (h \angle 1) + 1$ Module[44] zur Auswahl eines Blatt-Moduls getestet werden. Im obigen Beispiel ergibt sich mit $n = 19$ und $h = 2$ gerade die Anzahl von 20 pro Zyklus zu testenden Modulen und tatsächlich bestätigt sich bei einem entsprechend attributierten Ping-Pong-Benchmark die erwartete Auswahleffizienz von 5 % (siehe Tabelle 4.6).

Tabelle 4.6: Modulauswahleffizienz Ping-Pong-Benchmark (PROCESS)

Auswahlverfahren	getestet	gefeuert	Effizienz
Basis[a]	76.020	3.800	5,0 %
ereignisgesteuert[b]	76.020	3.800	5,0 %

a. Laufzeit-Optionen: -notreesleep -noimodopt **-eventdrive 0**
b. Laufzeit-Optionen: -notreesleep -noimodopt **-eventdrive 1**

44. Dies sind n Geschwister pro Ebene (mit Ausnahme des Spezifikationsmoduls, daher $h \angle 1$) und schließlich das Spezifikationsmodul selbst (wir gehen dabei von einem vollständig PROCESS-attributierten Modulbaum aus).

Die ereignisgesteuerte Auswahl hat hier keinerlei Vorteile gebracht, da immer alle Module getestet werden mussten.[45] Zwar kann in größeren (genauer: höheren) Modulbäumen durch die Beschränkung auf die favorisierten Teilbäume aus der ereignisgesteuerten Auswahl durchaus ein erheblicher Gewinn erzielt werden, jedoch schränkt die mit der `PROCESS`-Attributierung spezifizierte Synchronisation die Möglichkeiten zur Optimierung ein.

Dies führt uns zu einer ersten Spezifikationsstilregel[46] zur Unterstützung einer effizienten Implementierung: *Die Spezifikation von Modulsynchronisationen sollte auf ein notwendiges Minimum beschränkt werden.*

Dies ist, wie wir oben gesehen haben, neben der Reduktion der Vater-Sohn-Synchronisation, auf die wir später zurückkommen, nur noch durch Vermeidung der `PROCESS`-Attributierung möglich. Umgekehrt stellt sich sogar die Frage, wann diese Attributierung wirklich erforderlich ist.

Die `PROCESS`-Attributierung eines Moduls bewirkt ein strikt synchrones Voranschreiten seiner Kindmodule. Die Vorteile einer solchen auf dem semantischen Modell basierenden Synchronisation sind Fairness und Determinierung des Modulauswahlprozesses ohne Notwendigkeit einer expliziten (z. B. nachrichtenbasierten) Synchronisation der Module. So kann z. B. allein durch die Attributierung sichergestellt werden, dass ein Nachrichtenerzeuger (Sender) einen Nachrichtenempfänger nicht „überrennen“ kann.[47] Dadurch kann u. U. eine explizite kommunikationsbasierte Synchronisation eingespart werden, wie sie z. B. als explizit spezifizierte Sendekreditvergabe[48] zwischen einem User-Modul und den XTP-Protokollmaschinen in der XTP-4.0-Spezifikation vorzufinden ist.

Andererseits bedeutet der Rückgriff auf diese Extremform von Fairness auch, dass (insbesondere manuell erstellte) Implementierungen diese implizite Synchronisation exakt nachbilden müssen, um das intendierte Verhalten einzuhalten. Dies kann speziell bei verteilten Systemen auf Implementierungsebene nur kommunikationsbasiert erfolgen, wodurch sich der konzeptionelle Unterschied zwischen (explizit) spezifizierten und den zu ihrer Implementierung notwendigen Operationen vergrößert.[49] Letztlich ist diese enge Synchronisation in Bezug auf eine mögliche Implementierung nur für eng gekoppelte (also insbesondere nicht verteilt implementierte) Systeme sinnvoll.

Weiterhin garantiert der Einsatz einer `PROCESS`-Attributierung noch nicht, dass die Kommunikationswarteschlangen zwischen Sender und Empfänger nicht unbegrenzt gefüllt werden können. So kann ein Sender durch mehrere Sendeoperationen innerhalb einer Transition oder

---

45. Es müssen jeweils alle 20 Modulinstanzen pro Zyklus getestet werden, aufgrund des Szenarios bietet jedoch immer nur ein (Blatt-) Modul eine schaltbare Transition an.

46. siehe Ansatzpunkt *(c)* in Abschnitt 2.2.2

47. Nachrichtenkommunikation in Estelle ist asynchron und die Warteschlangenlänge ist unbegrenzt. Ein Sender kann also dauerhaft eine beliebig höhere Rate beim Erzeugen von Nachrichten haben als die Empfangsrate des Empfängers.

48. Sender und Empfänger steuern dabei durch ein explizit spezifiziertes Zusatzprotokoll letztlich die Maximalbelegung der Warteschlangen.

49. Ein typisches Beispiel ist der „**D**istributed **I**mplementatio**n** **G**enera**t**o**r**“ (DINGO), der auch die verteilte Implementierung von derart eng synchronisierten Modulen erlaubt und entsprechend in jedem Zyklus eine erhebliche Zahl von Synchronisationsnachrichten über das Basiskommunikationssystem (normalerweise TCP) übertragen muss. Neben der resultierenden geringen Performance aufgrund der Wartezeiten ist auch eine Abbildung auf bzw. Ankopplung an reale Protokollschnittstellen nicht ohne weiteres möglich.

durch das Schalten von mehreren Transitionen pro Zyklus[50] seine Senderate beliebig erhöhen. Umgekehrt kann ein Empfänger möglicherweise mehrere Zyklen zur Verarbeitung einer Nachricht benötigen. Offensichtlich erfordert der Einsatz der PROCESS-Attributierung zur Ratenkontrolle eine genaue Auslegung der beteiligten Komponenten.

Leider bietet Estelle unterhalb dieses radikalen Mittels keine Fairness bei der Auswahl zwischen Geschwistermodulen, so dass hier ggf. auf eine explizite kommunikationsbasierte Synchronisation (wie z. B. die schon erwähnte explizite Sendekreditvergabe) zurückgegriffen werden muss. Teilweise ergeben sich solche expliziten Synchronisationen aber auch schon aus dem spezifizierten Protokoll selbst, wie z. B. im Falle der Kommunikation zwischen Transportschicht-Protokollmaschinen.[51] Allgemein ist aus konzeptioneller Sicht sicherlich eine protokollbasierte (und daher auf explizit spezifizierten Mechanismen beruhende) Fairness einer alleine auf dem semantischen Modell basierenden vorzuziehen.

Wie zu erwarten gibt es aber auch bei der Regel zur Performancesteigerung durch Vermeidung der PROCESS-Attributierung Ausnahmen: Ändert man das obige Szenario derart ab, dass in einem Zyklus mehrere der Geschwister-Module gleichzeitig schaltbar sind, so kann die Auswahleffizienz erheblich ansteigen. Dazu ist es erforderlich, jeweils pro Zyklus in mehreren Modulen eine Nachricht zu verarbeiten und somit gleichzeitig eine ganze Menge in einem Pipeline-Schema durch die Nachrichtentransportkette zu leiten. Im Idealfall können dadurch in jedem Zyklus jeweils alle Blattmodule schalten. Der Vorteil ergibt sich daraus, dass gemäß der Estelle-Semantik das Vatermodul dabei in jedem Zyklus nur einmal geprüft werden muss. Es ergibt sich somit im Idealfall eine Auswahleffizienz von 95 %.[52]

Der Vorteil in diesem Szenario ergibt sich jedoch nicht wirklich durch die *Steigerung* der Synchronisation (also durch die PROCESS-Attributierung), sondern durch die *Senkung* der Auswirkungen einer anderen Synchronisationsbeziehung, nämlich der *Vater-Sohn-Priorität*. Vermeidet man diese, so kann man (auch außerhalb solch recht konstruierter Szenarien) sogar noch bessere Effizienzen erreichen.

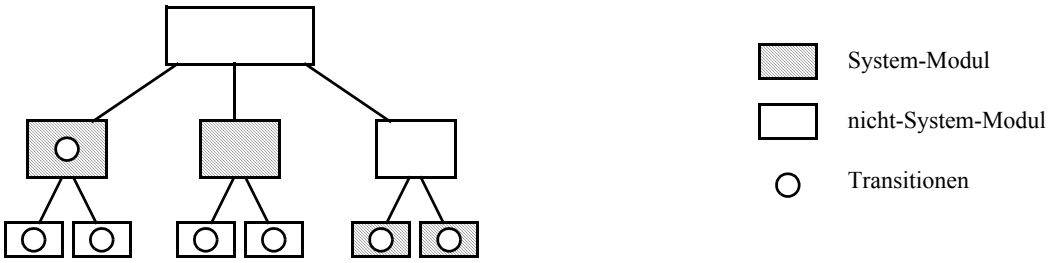


Abbildung 4-9: Unabhängige Module (1)

50. Dies ist durch eine weitere Unterstrukturierung in Kindmodule unter Weiterführung der PROCESS-Attributierung möglich.

51. Hier wird durch das Protokoll (z. B. Sliding-Window) die Anzahl *unterschiedlicher* Nachrichten in den Warteschlangen zwischen den Protokollmaschinen (also im Basisdienst) beschränkt. Zu einer effektiven Beschränkung der *Gesamtzahl* von Nachrichten sind jedoch Zusatzannahmen wie die maximale Paketlebensdauer und das Paketwiederholungsschema (u. a. Timeouts) erforderlich.

52. In jedem Zyklus werden 20 Module getestet und 19 geschaltet (in beiden bisher betrachteten Auswahlmodellen).

Zur Reduktion der Auswirkungen der Vater-Sohn-Priorität führt XEC eine Liste von Modulen (bzw. Modul-Teil-Bäumen), in denen eine Auswahl unabhängig von allen Vater- oder Geschwistermodulen stattfinden kann.

Betrachten wir dazu die in Abb. 4-9 dargestellte Modulinstanzhierarchie mit insgesamt 4 Subsystemen (schraffiert gekennzeichnet). Im Gegensatz zum bisherigen Auswahlverfahren muss hier nicht die gesamte Hierarchie bei der Auswahl durchlaufen werden, sondern sie kann direkt ausgehend von den Subsystemmodulen gestartet werden.

Diese Module, die wir allgemeiner als „*unabhängige Module*“ („*independent modules*“)[53] bezeichnen, werden parallel zur Baumstruktur der Module (`Module::Children`) als zentrale Liste `Specification::IndependentModules` geführt (siehe Abb. 4-10). Da dabei die exklusive Zugehörigkeit zu einer Liste zugunsten der (stark aggregierenden) Modulbaumstruktur (siehe Abschnitt 3.3.2.1) erhalten bleibt, wird diese neue Liste als schwache Aggregation mit dem Typ `XPtrList<Module>` realisiert.[54]

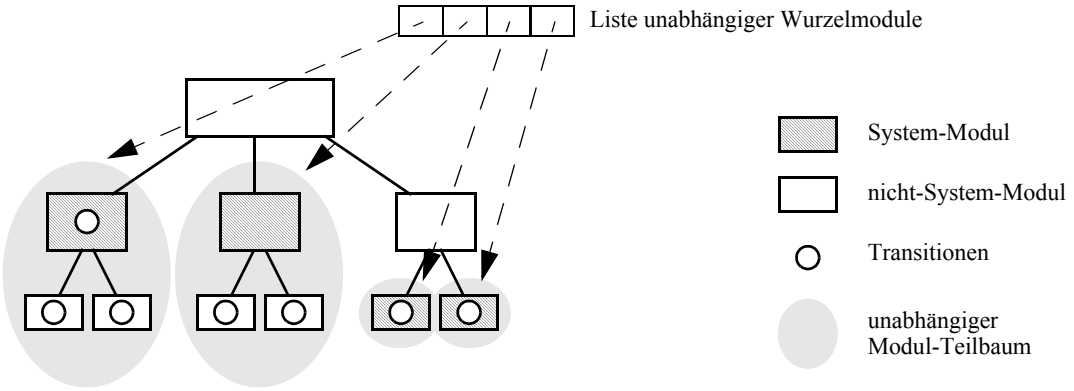


Abbildung 4-10: Unabhängige Module (2): System-Module

Bei einer Transitionsauswahl genügt es nun, von diesen Modulinstanz-Teilbäumen aus absteigend nach Modulen mit schaltbaren Transitionen zu suchen. Unter bestimmten Umständen können jedoch auch unterhalb der Subsystemebene unabhängige Module gefunden und in die `IndependentModules`-Liste eingefügt werden.

Enthält ein unabhängiges Modul keine eigenen Transitionen[55] und ist es selbst (`SYSTEM`-)`ACTIVITY`-attributiert,[56] so bestehen für die Kindmodulinstanzen ebenfalls keine relevanten Abhängigkeiten zwischen den Modulen oder zu ihrem Vatermodul[57], und sie sind entsprechend selbst unabhängige Module.

53. Die entsprechende Optimierung ist im Quellcode über bedingte Kompilierung abhängig vom Makro `OPT_IMOD_LIST` kenntlich gemacht.

54. Als Besonderheit werden `Ptr`-Knoten in dieser Liste nicht frei alloziert, da die `Module`-Instanzen zu ihrer Selbstverwaltung (also dem Ein- und Ausketten aus der Liste) Zugriff auf die sie referenzierenden Knoten benötigen. Stattdessen sind die Knoten als `XPtrListNode<Module>` `Module::IndependentModulesNode` in die `Module`-Instanz aggregiert (siehe dazu auch Abschnitt 3.3.1 und Anhang A.5). Dadurch werden insbesondere die Knoten bei der Destruktion der referenzierten Module automatisch mit beseitigt und ausgekettet.

55. bis auf eventuelle Initialisierungstransitionen, welche keine Transitionen im eigentlichen Sinne darstellen (siehe dazu auch Abschnitt 3.3.7.9)

56. In einem solchen Fall wäre es auch erlaubt, auf Spezifikationsebene die Kindmodule zu Subsystemen zu machen. Dies hat aber subtile semantische Implikationen bzgl. der Atomarität im Zusammenwirken der Kindmodule zur Folge (s. u.).

In einer Implementierung, die (wie es bei XEC der Fall ist) strikt sequentielle Folgen von Transitionsauswahl- und -Ausführungszyklen über alle Subsysteme hinweg realisiert, können diese unabhängigen Module (welche ja echte Teilbäume eines Subsystems sind) genau wie Subsysteme behandelt werden.[58] Ihr Vatermodul kann dabei ähnlich wie ein unattributiertes Modul in der `IndependentModules`-Liste unberücksichtigt bleiben.

Natürlich kann der Vorgang von diesen Kindmodulen aus, welche nunmehr selbst unabhängige Module sind, ggf. wiederholt werden, so dass im Extremfall nur noch die Blattmodule eines mehrstufigen Subsystem-Modulbaums in die Liste aufgenommen werden.[59]

Betrachten wir dazu das zweite Subsystem von links in Abb. 4-10, welches die o. g. Anforderungen erfüllt. Statt dieses Moduls können nun seine Kindmodule in die Liste aufgenommen werden (siehe Abb. 4-11).

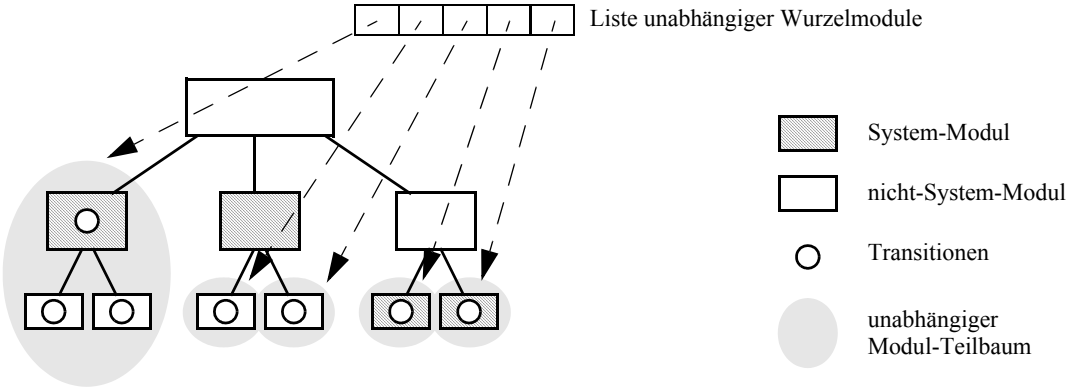


Abbildung 4-11: Unabhängige Module (3): `ACTIVITY`-Module

Die bisherigen Methoden zur Senkung der Synchronisation zwischen Modulinstanzen bewirken im Ping-Pong-Benchmark aufgrund der nicht-Attributierung des Spezifikationsmoduls, dass alle Blattmodule in der `IndependentModules`-Liste eingehängt werden und somit keinerlei modellbasierte Synchronisation mehr zwischen den Modulinstanzen besteht. Dadurch kann die nachrichtengesteuerte Auswahl immer direkt auf das jeweilige Empfängermodul zugreifen und so eine Auswahleffizienz auf Modulebene von praktisch 100% erreichen (siehe Tabelle 4.7).

---

57. Die Vater-Sohn-Priorität greift nicht und mögliche exportierte Variablen der Kindmodule sind nach der Initialisierungsphase nur noch von diesen zugreifbar und somit unkritisch.

58. XEC behandelt unattributierte Module bezüglich des Zusammenwirkens von Subsystemen wie ACTIVITY-attributierte Module, d.h. es wird jeweils für ein Subsystem (genauer: für einen unabhängigen Teilbaum) ein Auswahl- und anschließend ein Ausführungszyklus durchgeführt. Alternative Verschränkungen von Auswahl- und Ausführungszyklen zwischen Subsystemen, wie sie die Estelle-Semantik ebenfalls erlaubt, können hier die angesprochene Unabhängigkeit von Teilsystemen unterhalb der Subsystemebene beeinträchtigen.

59. Im pathologischen Fall, dass auch solche Blattmodule keine Transitionen besitzen, führt der o. g. Algorithmus dazu, dass sie ebenfalls nicht in die Liste aufgenommen werden, also völlig aus der Auswahl ausgeschlossen werden. Hier könnte natürlich sogar auf die Forderung nach einer `ACTIVITY`-Attributierung verzichtet werden.

Tabelle 4.7: Modulauswahleffizienz Ping-Pong-Benchmark (ACTIVITY) mit Independent-Module-Optimierung

Auswahlverfahren	getestet	gefeuert	Effizienz
Basis[a]	41.714	3.800	9,1 %
ereignisgesteuert[b]	7.623	3.800	49,8 %
independent Modules[c]	38.084	3.800	10,0 %
independent Modules und ereignisgesteuert[d]	3.822	3.800	99,4 %

a. Laufzeit-Optionen: -notreesleep ***-no*imodopt -eventdrive 0**
b. Laufzeit-Optionen: -notreesleep ***-no*imodopt -eventdrive 1**
c. Laufzeit-Optionen: -notreesleep **-imodopt -eventdrive 0**
d. Laufzeit-Optionen: -notreesleep **-imodopt -eventdrive 1**

Bei komplexeren Spezifikationen, die z. B. aufgrund der Notwendigkeit einer dynamischen Konfiguration nicht auf aktive Vatermodule verzichten können, ist eine solche Maximaleffizienz leider nicht mit den bisher diskutierten Mitteln zu erreichen. Ein wesentlicher Grund ist hier die Vater-Sohn-Priorität. Im folgenden Abschnitt untersuchen wir Spracherweiterungen, die in solchen Fällen wirksam sind.

4.2.2.4 Senkung der Synchronisation zwischen Modulinstanzen durch Erweiterungen

In [BrGo94] wird „*Asynchronous-Process*“ als eine semantische und syntaktische Estelle-Erweiterung vorgestellt, die dem Spezifizierer ein Mittel in die Hand gibt, innerhalb von Subsystemen asynchron operierende Kindmodule zu spezifizieren und somit die Möglichkeiten zur Spezifikation von Konkurrenz in Estelle zu steigern. Dazu wird das Vatermodul mit der neu eingeführten Attributierung „`ASYNCHRONOUS PROCESS`“[60] versehen.

Da diese Erweiterung neben der ursprünglich intendierten Steigerung der Ausdrucksfähigkeit von Estelle durch die Senkung der Synchronisation auch die Wirksamkeit der oben vorgestellten ereignisgesteuerten Implementierung steigert, wurden in das XEC-Toolkit entsprechende Erweiterungen integriert.[61]

Die Kindmodulinstanzen eines mit „`ASYNCHRONOUS PROCESS`“ attributierten Vatermoduls werden dabei nicht synchron mit dem umgebenden Subsystem ausgeführt, sondern bilden gewissermaßen jeweils eigene Subsysteme auf Basis der Interleaving-Semantik von Estelle. Sie qualifizieren sich damit als *unabhängige Module* im obigen Sinne. Entsprechend werden diese Kindmodulinstanzen ebenfalls in die `IndependentModules`-Liste aufgenommen.

Nehmen wir als Beispiel aufbauend auf dem obigen Szenario (siehe Abb. 4-11) an, dass das erste Systemmodul von links als „`ASYNCHRONOUS SYSTEMPROCESS`“ attributiert ist. Da die beiden Kindmodulinstanzen[62] dadurch unabhängige Module im obigen Sinne werden, werden

60. bzw. „`ASYNCHRONOUS SYSTEMPROCESS`“

61. Zur Aktivierung der erweiterten Syntax muss `PET` mit der Option „`-async`“ aufgerufen werden, (siehe Anhang A.1).

62. Zur Erinnerung: Die `ASYNCHRONOUS`-Attributierung wirkt sich nur auf die Synchronisation der Kindmodule und nicht auf das attributierte Modul selbst aus (s. o.).

sie in die IndependentModules-Liste aufgenommen (siehe Abb. 4-12). Das Vatermodul, welches in dem Beispiel zufälligerweise selbst ein unabhängiges Modul[63] ist, bleibt in seinem Status dabei zunächst[64] unverändert, d.h. es verbleibt in der Liste.

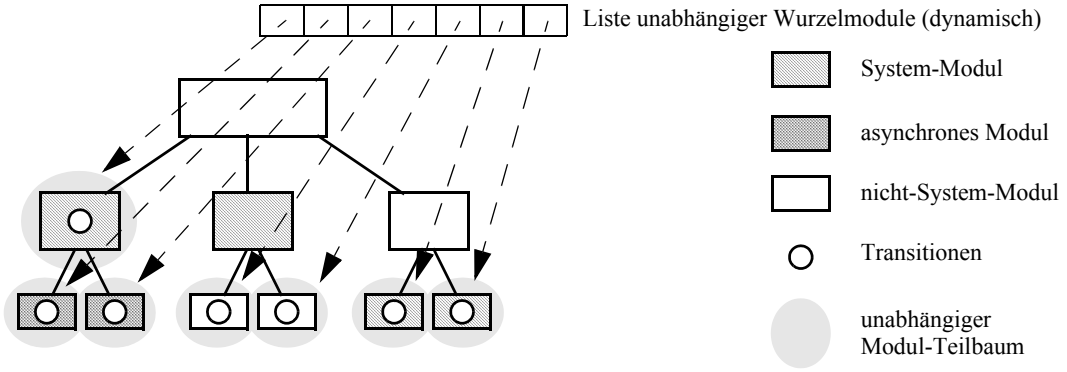


Abbildung 4-12: Unabhängige Module (4): ASYNCRONOUS-Module

Da derartige asynchrone Modulinstanzen dynamisch erzeugt und terminiert werden können, entsteht hier im Gegensatz zu den bisher diskutierten Mechanismen erstmals Dynamik bzgl. der Zusammensetzung der IndependentModules-Liste. Diese Dynamik bildet gerade den Vorteil der Erweiterung bzgl. einer effizienten Implementierung, da sie eine konzeptionelle Trennung der Modulhierarchie und der Unabhängigkeit der Module bietet. Im Idealfall können so alle attributierten Module auch unabhängig sein, wodurch eine Maximierung der Modulauswahleffizienz ermöglicht wird.

So können im oben beschriebenen Ping-Pong-Benchmark auch im Falle einer dynamischen Konfigurierbarkeit und dem dazu erforderlichen Vorhandensein von Transitionen im Spezifikationsmodul alle Module unabhängig sein. Dadurch wird die in Tabelle 4.7 wiedergegebene Auswahleffizienz von fast 100% erreicht.

Leider hat die Anwendung der Asynchronous-Process-Erweiterung auch einige Nachteile. So sind zur Vermeidung von konkurrierenden Zugriffen auf gemeinsame Variablen keine exportierten Variablen in den asynchronen Kindmodulen erlaubt. Diese Einschränkung kann insbesondere die Anwendung der Erweiterung auf existierende Spezifikationen beeinträchtigen. So sind zur breiten Anwendung in der XTP-4.0-Spezifikation erhebliche Modifikationen erforderlich, da innerhalb der Protokollmaschinen exportierte Variablen vielfältig eingesetzt werden.

Ein anderes Problem resultiert auf der Wirkungsrichtung der „ASYNCHRONOUS PROCESS“ Attributierung auf die Asynchronität der *Kindmodule*. Dadurch können nur alle oder keines der Kindmodule asynchron operieren, was die Flexibilität der Erweiterung einschränkt. Insbesondere wirkt sich dies im Zusammenhang mit dem Verbot exportierter Variablen der Kindmodule aus: Soll nur ein Kindmodul asynchron operieren, so darf keines der Geschwister exportierte Variablen besitzen.

Schließlich ergeben sich einige semantische Probleme durch die vollständig asynchrone Operation solcher Kindmodule, die speziell bei ihrer Termination zutage treten.

---

63. Wenn das Vatermodul stattdessen kein Systemmodul wäre, könnte es durchaus abhängig sein.

64. Besitzt das Vatermodul keine Transitionen, so sind die obigen Regeln zur Behandlung unabhängiger, ACTIVITY-attributierter Module ohne Transitionen analog anwendbar.

Bei genauerer Betrachtung wird deutlich, dass zur Steigerung der Unabhängigkeit der Module zum Zwecke einer effizienteren Implementierbarkeit die Asynchronous-Process-Erweiterung mit ihrer völligen Abkopplung der Kindmodulzyklen vom Vatermodul überdimensioniert ist.

Die wesentliche Eigenschaft der Asynchronous-Process-Erweiterung, die wir benötigt haben, ist die Aufhebung der Vater-Sohn-Priorität während der Auswahl. Entsprechend schlagen wir nun eine alternative Estelle-Erweiterung vor, die genau diese Entkopplung liefert.

Dazu führen wir nun die *Independent-Module-Erweiterung* ein. Diese basiert syntaktisch auf den neuen Modulattributierungen „`INDEPENDENT ACTIVITY`“ und „`INDEPENDENT PROCESS`“[65], die jeweils alternativ zu einer „`ACTIVITY`“- bzw. „`PROCESS`“-Attributierung eingesetzt werden können und entsprechend bzgl. der Modulattributierungsregeln äquivalent behandelt werden (siehe z. B. Tabelle 7.29 auf Seite 331).[66]

Wie der Name schon erwarten lässt, ist das Ziel der Independent-Module-Erweiterung, das attributierte Modul zu einem unabhängigen Modul im obigen Sinne zu machen. Dies wird semantisch dadurch erreicht, dass lediglich bei der Transitionsauswahl die Vater-Sohn-Priorität zwischen dem entsprechend attributierten Modul und seinem Vatermodul aufgehoben wird.

Betrachten wir dazu die ursprüngliche semantische Definition der Transitionsauswahl $AS(gid_P)$ in Abschnitt 5.3.3 von [ISO97]:

Definition 4.1: Definition of $AS(gid_P)$ (original):

(a) If the tree of gid_P is simply the single instance P, i.e., gidP = (P; s_P) and (s_P, t_P) is the local situation of P in gidP , then $AS(gid_P) = \{t_P\}$ (the set {null} is identified with the empty set).

(b) Let $P_1, P_2, \ldots, P_k$, k = 1, be children instances of P in the tree of gid_P , and let (s_P, t_P) be the local situation of P in gid_P .

(1)If $t_P \neq null$, then $AS(gid_P) = \{t_P\}$

(2)If $t_P = null$ and P is an activity or systemactivity, then $AS(gid_P)$ equals one of the nonempty sets $AS(gid_{Pi})$, i = 1, . . . , k, if such exists, and is empty otherwise. The choice is nondeterministic.

(3)If $t_P = null$ and P is a process or systemprocess, then $AS(gid_{P_i}) = \bigcup_{i=1}^{k} AS(gid_{P_i})$.

Die Vater-Sohn-Priorität wird hier durch die Bedingung „$t_P \neq null$“ bzw. „$t_P = null$“ in den Punkten (1), (2) und (3) realisiert. Entsprechend kann die gewünschte Erweiterung in der Semantik durch Erweiterung von Unterpunkt (1) erfolgen, indem bei einer entsprechenden `INDEPENDENT`-Attributierung in dem Fall, dass sowohl die Vater- als auch die Kindmodulauswahl (analog zu (2) bzw. (3)) erfolgreich sind, indeterministisch entweder die Transition des

65. Es existiert jedoch keine „SYSTEM“-Variante dieser Attributierung, da diese aufgrund der nicht existierenden Vater-Sohn-Priorität überflüssig wäre.

66. Die Independent-Module-Erweiterung ist verträglich mit der Asynchronous-Process-Erweiterung und kann aufgrund der entgegengesetzten Wirkungsrichtung (zum Vater- statt zu den Kindmodulen, s. u.) sogar mit ihr im selben Modul kombiniert auftreten. Die einzige mögliche Kombination ist dabei „`ASYNCHRONOUS INDEPENDENT PROCESS`“, wobei diese Attributreihenfolge dann verbindlich ist. Ein „`INDEPENDENT`“-Attribut des Kindmoduls eines „`ASYNCRONOUS`“-attributierten Vatermoduls wird dagegen semantisch vollständig überdeckt.

Vatermoduls oder die Transitionen der Kindmodule auswählt. Wir verzichten an dieser Stelle auf die konzeptionell unaufwändige aber technisch etwas unübersichtliche Formalisierung der Semantik auf Basis von Def. 4.1.

Somit unterscheidet sich die Independent-Module-Erweiterung in zwei wesentlichen Punkten von der Asynchronous-Process-Erweiterung:

(i) Die Reduktion der Synchronisierung bezieht sich nur auf die *Auswahlphase*, d.h. die Modulinstanz ist weiterhin in die Auswahl- und Ausführungsphasen des umgebenden Subsystems[67] integriert.

(ii) Die Attributierung eines Moduls ändert lediglich die Synchronisationsbeziehung zwischen dem Modul und seinem Vatermodul.

Dabei vermeidet die Independent-Module-Erweiterung die Nachteile der Asynchronous-Process-Erweiterung:

- Die Modellierung, welche Module unabhängige Module sind, ist präzise steuerbar *(ii)*.
- Der Eingriff in die Estelle-Semantik ist konzeptionell und technisch geringfügig und bewirkt durch die weiterhin gegebene Synchronität bei Auswahl und Ausführung keinerlei Synchronisations- und Konkurrenzprobleme *(i)*.
- Aufgrund dieser Synchronität ist insbesondere keine Beschränkung für den Einsatz *exportierter Variablen* erforderlich *(i)*.

Insgesamt können somit zunächst *alle attributierten nicht-Systemmodule*[68] ohne syntaktische oder semantische Beschränkungen als Independent-Module attributiert werden, wodurch (zusammen mit den ohnehin unabhängigen Systemmodulen) *sämtliche attributierten Module unabhängig* im obigen Sinne werden.

Die Entscheidung, ob ein Modul als Independent-Module attributiert werden kann, ist dabei ausschließlich von den vom Spezifizierer erwünschten Vorrangregelungen zwischen Vater- und Sohnmodulen abhängig. Bei genauerer Betrachtung zeigt sich, dass die in Estelle vorgesehene strikte Prioritätsregelung in diesem Bereich oft überhaupt nicht erforderlich ist, bzw. gegebenenfalls leicht durch eine explizit spezifizierte Synchronisation ersetzt werden kann (vergl. auch die Diskussion zur PROCESS-Attributierung in Abschnitt 4.2.2.3). Hier sind gegebenenfalls präzise Analysen der zum korrekten Funktionieren einer Spezifikation erforderlichen Vorrangregelungen nötig.

Aber auch wenn nur ein Teil der Module als unabhängig attributiert werden kann, so skaliert die resultierende Steigerung der Abhängigkeiten zwischen den Modulen praktisch linear und trägt somit zur Steigerung der Effektivität der ereignisgesteuerten Modulauswahl bei. Dabei ergeben sich unter XEC für unabhängige Module die selben Performance-Vorteile wie für asynchrone Module (s. o.). Da die `INDEPENDENT`-Attributierung jedoch auch bei Modulen mit exportierten Variablen anwendbar ist, bietet sie potentiell ein höheres Optimierungspotential.

Jenseits der hier betrachteten *sequentiellen* Implementierungen kann die Asynchronous-Process-Erweiterung u. U. jedoch durchaus bzgl. der effizienten Implementierbarkeit Vorteile gegenüber der Independent-Module-Erweiterung bieten, da erstere im Sinne einer echten Parallelausführung auf verteilten oder Mehrprozessor-Systemen eine vollständig asynchrone Operation

67. Aufgrund der Estelle-Attributierungsregeln sind diese Module immer *Teil* eines Subsystems, bilden jedoch *selbst kein* Subsystem.

68. also eben gerade die als `PROCESS` (nicht aber `SYSTEMPROCESS`) oder `ACTIVITY` (nicht aber `SYSTEMACTIVITY`) attributierten Module

der Teilsysteme erlaubt, Auswahl- und Ausführungszyklen also nicht mehr synchronisiert sind. Die Independent-Module-Erweiterung unterstützt derartige Parallelverarbeitungen im Allgemeinen nur *innerhalb* jeweils eines Auswahl- oder Ausführungszyklus.[69]

Wir werden später in Abschnitt 4.4 die Wirkung dieser Umattributierung für die sequentiellen Implementierungen von XEC nochmals an komplexeren Spezifikationen quantitativ bewerten.

4.2.2.5 Erweiterte ereignisgesteuerte Auswahlheuristiken

Unsere bisherigen Überlegungen zur Effizienzsteigerung der globalen Modulauswahl zielten auf eine Annäherung an die Arbeitsweise des Activity-Thread-Modells ab. Dazu wurden Heuristiken, Spezifikationsstilaspekte und Spracherweiterungen diskutiert, die in Abhängigkeit von der Einhaltung der dem Activity-Thread-Modell zu Grunde liegenden Voraussetzungen dessen Effizienz approximieren können.

Mit zunehmender Abweichung von diesen Voraussetzungen sinkt auch die Wirksamkeit der Heuristiken und das vorgestellte Verfahren geht schließlich gleitend in eine rekursive und zunehmend ungerichtete Suche durch den Modulinstanzbaum über, wie wir sie bereits bei der Urform des Server-Modells kennen gelernt haben (siehe Abschnitt 4.2.1.1).

Ein typisches Beispiel für eine solche Situation ist das Ende einer Kette von Nachrichtenverarbeitungsschritten. Im ursprünglichen Activity-Thread-Modell (Abschnitt 4.2.1.2) stellt diese Situation auch das Ende des Activity-Threads dar und die Aktivierung des (Teil-) Systems endet, bis von außen durch die Übergabe einer neuen Nachricht ein neuer Activity-Thread gestartet wird. Der Preis für diese konzeptionelle Einfachheit ist jedoch die beschränkte Anwendbarkeit des Activity-Thread-Modells.

Das oben vorgestellte hybride Ausführungsmodell erlaubt als universelles Implementierungsmodell für Estelle jedoch auch komplexere Ereignisverläufe. Man findet in Protokollspezifikationen u. a. folgende Szenarien (siehe Abb. 4-14), die von der einfachen Activity-Thread-Verarbeitung abweichen:

(a) Spontane Erzeugung mehrerer Nachrichten.
Dieses Verhalten findet man u. a. häufig in der Anwendungsschicht, wenn Kommunikationsdienstnutzer (z. B. auf ein Anfrage-Paket hin) eine ganze Folge von Antwortpaketen erzeugen. Da die Erzeugung dieser Nachrichten häufig zeitlich ausgedehnt (also nicht als ein atomarer Vorgang) modelliert werden soll, erfolgt die Emission durch das Feuern mehrerer Transitionen. Von außen erscheinen diese Pakete dann als scheinbar *spontane* (also nicht unmittelbar durch äußere Ereignisse induzierte) Nachrichtenemissionen. In diese Klasse fallen auch timeoutinduzierte Paketemissionen (z. B. Paketwiederholungen in Transportschichtprotokollen).

(b) Multiplikation eines Activity-Threads.
Sendet ein Knoten einer Nachrichtenverarbeitungskette auf den Empfang einer Nachricht hin *mehrere Folgenachrichten* an den Nachfolgeknoten weiter, so erscheint eine der Nachrichten als Fortsetzung des Activity-Threads und die weiteren Nachrichten erzeu-

69. Es entstehen durch die Independent-Module-Erweiterung jedoch keine Zugriffskonflikte bzgl. exportierter Variablen, wie sie bei der Asynchronous-Process-Erweiterung zum Ausschluss dieses Kommunikationsmittels zwischen asynchron operierenden Modulen geführt hat (siehe Abschnitt 6.4.3).

gen jeweils scheinbar neue Threads an das selbe Ziel. Eine solche Vervielfachung („*Multiplikation*“) eines Activity-Threads tritt in Protokollspezifikationen z. B. in einer Paketfragmentierer-Komponente auf.

(c) *Verzweigung eines Activity-Threads.*
Dies ist eine Variation des vorherigen Szenarios, bei dem ebenfalls mehrere Pakete entstehen, die jedoch mit unterschiedlichen Zielen (gewissermaßen in *unterschiedliche Richtungen*) verschickt werden. Dies tritt fast immer bei Transportschichtprotokollen auf, wenn diese beim Empfang einer Nachricht vom Basisdienst sowohl die Nutzlast nach „oben“ weiterreichen, als auch den Empfang zur Partnerprotokollinstanz (also nach „unten“) quittieren.

(d) *Unterbrechung und spätere Fortsetzung eines Activity-Threads.*
Dies tritt auf, wenn nach dem Empfang einer Nachricht durch eine Transition nicht sofort von der selben Transition eine Folgenachricht erzeugt wird, sondern dies erst später erfolgt. Aus Sicht des Activity-Thread-Modells hat der Activity-Thread damit geendet und entsteht ggf. später neu. Ein Spezialfall dieses Szenarios liegt vor, wenn die Folgetransition erst nach Ablauf eines Timers starten kann.

(e) *Spontane interne Operationen.*
Gerade in komplexen Protokollen werden interne Managementaufgaben oft zur Senkung der Komplexitäten in separate Transitionen ausgelagert, welche von den Paketverarbeitungs-Transitionen getrennt sind. So werden z. B. in der Estelle-Spezifikation von XTP 4.0 nicht mehr benötigte Kontextmodule von spontanen Transitionen ihres Vatermoduls „irgendwann“ beseitigt.

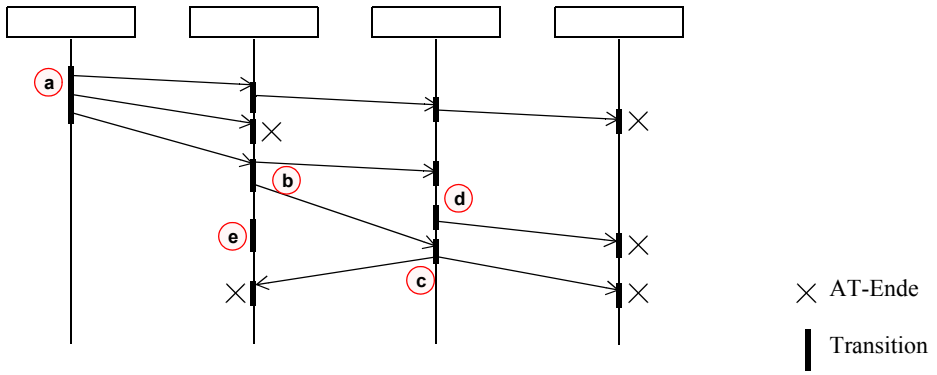


Abbildung 4-13: Vom Activity-Thread-Modell abweichende Szenarien

Die Gemeinsamkeit dieser Szenarien ist, dass neben dem vordergründigen Activity-Thread des Ende-zu-Ende-Nachrichtentransports weitere Transitionsaktivierungen oder gar neue (zu dem noch laufenden *parallel* existierende) Activity-Threads entstehen. Tritt dies vermehrt auf, so wird das Konzept der Erkennung und Verfolgung *favorisierter* Module schnell verwässert, da dann viele (oder im Extremfall alle) Modulinstanzen dafür in Frage kommen.

Betrachten wir dazu die in [Turn93] aufgeführte Rumpf-Spezifikation eines Sliding-Window-Protokolls. Ergänzt man diese Protokollmaschinen um die fehlenden Kernkomponenten und vervollständigt sie mit geeigneten Benutzer- und Netzwerkmodulen zu einem vollständigen Testszenario (Abb. 4-14 bzw. Anhang B.4), so kann man die meisten der o. g. Effekte in der Spezifikation vorfinden (siehe Abb. 4-15). Dazu übertragen die Benutzermodule in mehreren Runden unsynchronisierte Sequenzen (Bursts) von Nachrichten (a) an die Transportschicht-Protokollmaschinen (c). Das Netzwerkmodul überträgt die von den Protokollmaschinen er-

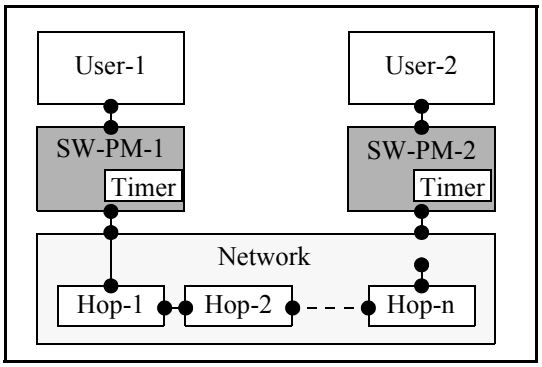


Abbildung 4-14: Modulinstanz- und Verbindungsstruktur des Sliding-Window-Benchmarks

zeugten PDUs in mehreren Hops, welche zusätzlich optional über interne Operationen zeitverzögert werden (d).[70] Gegebenenfalls werden zusätzlich durch Timeouts[71] in den Protokollmaschinen Paketwiederholungen ausgelöst.

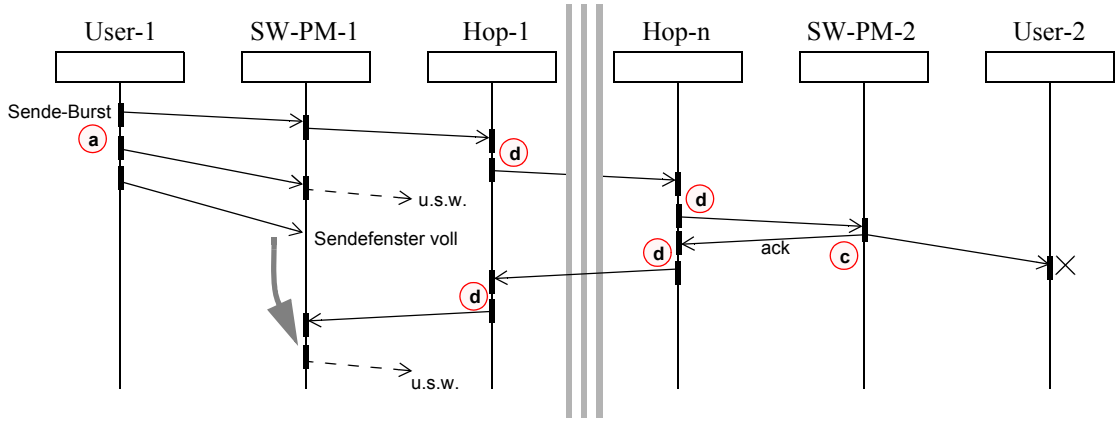


Abbildung 4-15: Vom Activity-Thread-Modell abweichende Szenarien

Vergleicht man anhand der Implementierung dieses Systems zunächst die Wirksamkeit der bisher eingeführten Optimierungen, so stellt man fest, dass das ständige Abreißen von Activity-Threads durch das damit verbundene Suchen nach einem schaltbaren Modul (also nach einem anderen Activity-Thread) die Auswahleffizienz erheblich verschlechtert.

Der Sliding-Window-Benchmark wird anhand folgender Parameter ausgewertet, wobei zur Steigerung der Wirkung der Independent-Module-Optimierung alle `ACTIVITY`-Module[72] als `INDEPENDENT` attributiert wurden, so dass alle Module unabhängig sind.

70. In jedem Fall erfolgt der Weitertransport in einem Hop über das Schalten mehrerer Transitionen.

71. Die `DELAY`-Transitionen der Protokollmaschinen sind in der Ausgangsspezifikation in ein Kindmodul ausgelagert und werden nachrichtenbasiert gesteuert. Dadurch entstehen weitere Verzweigungen des Activity-Threads.

72. Benutzer-, Protokollmaschinen- und Netzwerkmodul sind `ACTIVITY`-attributiert, also mussten nur die Hop-Module und die Timer-Module innerhalb der Protokollmaschinen `INDEPENDENT` attributiert werden.

- Anzahl der Runden: 10
- Anzahl Nachrichten pro Runde: 10
- Anzahl Hops (*H*): 3
- Sliding-Window-Fenstergröße: 20
- Medium-Delay [ms] 0
- Retransmission-Timeout [ms] 300

Dabei zeigt sich, dass durch die Independent-Module-Optimierung zusammen mit der Ereignissteuerung eine Verbesserung der Modulauswahleffizienz von 19,0 % auf 55,3 % erreicht wird (siehe Tabelle 4.8), diese aber offensichtlich noch nicht optimal ist. Die Ursache für die begrenzte Wirkung der Optimierung ist, dass die eigentlichen Activity-Threads zwar mit optimaler Effizienz ablaufen, beim häufig auftretenden *Ende eines Threads* jedoch relativ ungerichtet nach einem schaltbaren Modul gesucht wird. Diese Suche erfolgt gemäß der jeweils vorliegenden Reihenfolge der Module in den Kindmodul- bzw. Independent-Module-Listen, welche wiederum Folge der zuvor geschalteten Transitionen bzw. durch die Ereignissteuerung favorisierten Module sind.

Tabelle 4.8: Modulauswahleffizienz SW-Benchmark (ACTIVITY) mit Independent-Module-Optimierung

Auswahlverfahren	getestet	gefeuert	Effizienz
Basis[a]	8.178	1.510	19,0 %
ereignisgesteuert[b]	4.493	1.510	34,5 %
independent Modules[c]	2.094	1.510	72,1 %
independent Modules und ereignisgesteuert[d]	2.729	1.510	55,3 %

a. Laufzeit-Optionen: -notreesleep ***-no*imodopt -eventdrive 0**
b. Laufzeit-Optionen: -notreesleep ***-no*imodopt -eventdrive 1**
c. Laufzeit-Optionen: -notreesleep **-imodopt -eventdrive 0**
d. Laufzeit-Optionen: -notreesleep **-imodopt -eventdrive 1**

Bei einfachen Activity-Threads (ohne zusätzliche Events durch Verzweigungen, Multiplikation, etc.) befinden sich am Ende die beteiligten Module in umgekehrter Reihenfolge ihres Schaltens in diesen Listen.[73] Entsprechend erfolgt nach dem Ende des Threads die Suche nach dem nächsten schaltbaren Modul zunächst rückwärts entlang des letzten Threads. Dadurch werden, wie man in Abb. 4-16 erkennt, nach der eigentlichen Activity-Thread-Phase, welche mit 100 % Effizienz verläuft, in der „Rücklaufphase“ fast ebenso viele Module erfolglos getestet und somit die Gesamtauswahleffizienz gegen 50 % gesenkt (im Benchmark 55,3 %).

Ein interessantes Randergebnis zeigt sich bei der nicht-ereignisgesteuerten Auswahl. Hier wird bei Independent-Module-Optimierung mit 72,1 % Auswahleffizienz ein besseres Ergebnis erreicht. Ursache ist das völlig andere Ausführungsmuster der Spezifikation. Aufgrund der fehlenden Ereignissteuerung werden die zuletzt gefeuerten Module immer bevorzugt, wodurch ein

73. Ursache für die Reihenfolge ist, dass die bei der Auswahl von der Liste `Module::Children` in die Liste `Module::SelectedChildren` verschobenen Module nach der Schaltphase zurück verschoben und dabei vorne in dieser Liste eingekettet werden (siehe Abschnitt 3.4.3). Man kann diese auch hinten in die Liste einketten, jedoch hat dies massive Nachteile bzgl. der Modulauswahleffizienz bei spontan feuernden Folgetransitionen im selben Modul.

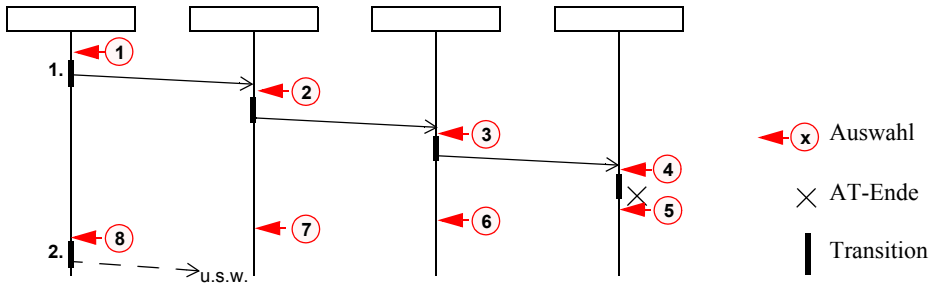


Abbildung 4-16: Auswahl beim Ende eines Activity-Threads (ereignisgesteuert)

Paketburst,[74] wie er auch im Testszenario verschickt wird, immer als geschlossene Gruppe von Nachrichten von Modul zu Modul transportiert wird (siehe Abb. 4-17). Dadurch werden bei einem Burst der Länge n und einer Activity-Thread-Länge von m Hops $m \cdot (n+1)$ Transitionen geschaltet und incl. „Rücklauf“ $(m+1) \cdot (n+1) + (n \angle 1)$getestet. Für große m und n geht die Auswahleffizienz somit gegen 100 %:

$$\lim_{m,\, n \to \infty} \frac{(m+1) \cdot (n+1) + (n \angle 1)}{m \cdot (n+1)} = 1$$

Die Wirkung dieses Effektes ist allerdings durch die oben bereits beschriebenen Störeffekte in den Activity-Threads beschränkt, wie man anhand der Benchmark-Ergebnisse sehen kann. In stärker synchronisierten Szenarien wie beim Ping-Pong-Benchmark (siehe Tabelle 4.7 auf Seite 141) oder bei verkleinerten Sliding-Window-Fenstergrößen ist die resultierende Effizienz entsprechend gering.

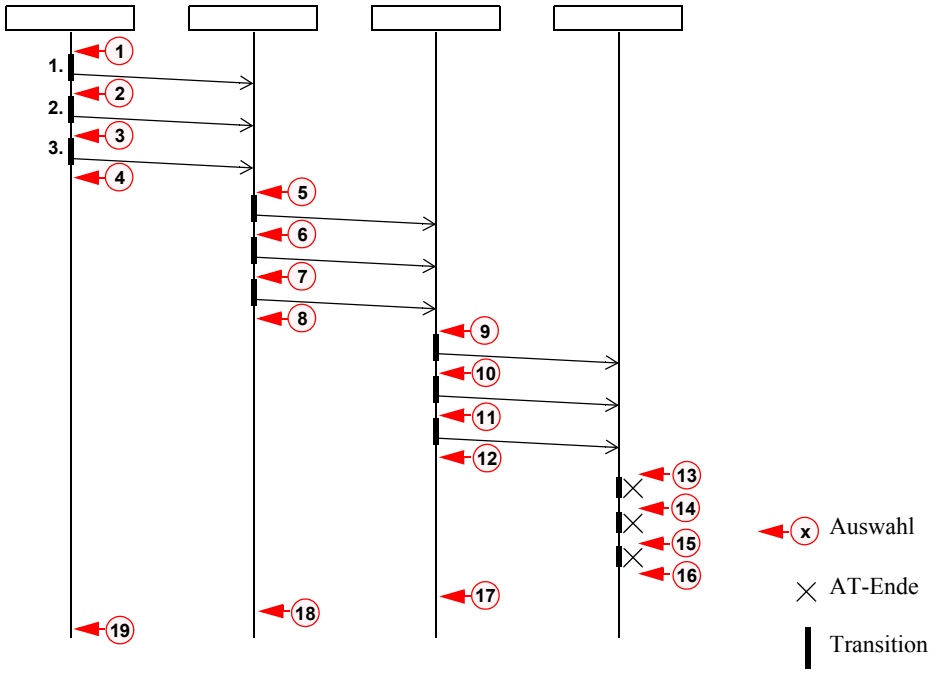


Abbildung 4-17: Auswahl beim Ende eines Activity-Threads (nicht ereignisgesteuert)

Kommen wir also wieder zurück zum ereignisgesteuerten Auswahlmodell. Zur Verbesserung seiner Wirkung sind also offensichtlich die oben beschriebenen Störungen der Activity-Threads zu berücksichtigen. Die meisten Störungen lassen sich auf folgende Probleme reduzieren:

74. also eine unsynchronisiert verschickte Folge von Nachrichten, die durch einmaliges oder (wie im Fall des Sliding-Window-Benchmarks) mehrmaliges Schalten von Transitionen erzeugt werden

- Es muss mehr als ein Activity-Thread zu einem Zeitpunkt verfolgt werden.
- Das Ende eines Activity-Threads muss erkannt werden. In diesem Fall wird ggf. ein anderer Thread weiter verfolgt.

Dazu unterscheidet das XEC-Laufzeitsystem[75] zwischen einem *primären* und einer Menge von *sekundären* Ereignissen (bzw. durch die Ereignisketten definierten Activity-Threads). Tritt in einer Ausführungsphase ein erstes Ereignis[76] auf, so wird dieses als primäres Ereignis über den in Abschnitt 4.2.2.2 dargestellten Mechanismus in die Modulinstanzhierarchie eingeprägt, so dass das betroffene Modul im nächsten Durchlauf favorisiert wird. Zusätzlich wird die Variable „`Specification::bPrimaryEvent`“ auf `true` gesetzt, um das Auftreten eines primären Ereignisses in dieser Phase anzuzeigen. Weitere Ereignisse werden durch die Aufnahme der Zielmodulinstanz in die Liste „`XPtrList<Module> Specification::SecondaryEventQueue`“ behandelt.

In jeder Auswahlphase wird im Sinne der Ereignissteuerung das über die Modulinstanzhierarchie eingeprägte Ereignis *verbraucht*. Entsprechend wird zu Beginn „`bPrimaryEvent`“ jeweils auf `false` zurückgesetzt. Wird die Variable im Verlauf der nächsten Ausführungsphase nicht auf `true` gesetzt, so ist kein Ereignis aufgetreten und der verfolgte Activity-Thread hat geendet. In diesem Fall wird eines der u. U. früher aufgetreten sekundären Ereignisse aktiviert, indem zu Beginn der Auswahlphase eine Modulreferenz aus der `SecondaryEventQueue` entnommen und nun das referenzierte Modul im Sinne eines neuen primären Ereignisses *favorisiert* wird.

Die `SecondaryEventQueue` wird dabei so betrieben,[77] dass jedes Modul höchstens einmal Mitglied der Liste sein kann. Dadurch wird verhindert, dass eine Mehrzahl von Ereignissen im selben Modul zu einer fortschreitenden Füllung der Queue führt.

Durch diese Unterstützung multipler Activity-Threads durch die Ereignissteuerung können bereits die oben beschriebenen Szenarien *Multiplikation* (b) und *Verzweigung* (c) eines Activity-Threads (siehe auch Abb. 4-15) vollständig behandelt werden. Daraus ergibt sich eine Verbesserung der Auswahleffizienz von 55,3 % auf 65,8 % (siehe Tabelle 4.9).

Tabelle 4.9: Modulauswahleffizienz SW-Benchmark (ACTIVITY) mit Independent-Module-Optimierung

Auswahlverfahren	getestet	gefeuert	Effizienz
Basis[a]	8.178	1.510	19,0 %
independent Modules und ereignisgesteuert (einfach)[b]	2.729	1.510	55,3 %
independent Modules und ereignisgesteuert (mehrfach)[c]	2.293	1.510	65,8 %

75. gesteuert durch das Compiler-Makro `OPT_SEC_EVENT` (für „secondary event optimization“) und die Kommandozeilenoption „`-eventdrive` *n*“ mit $n \geq 2$

76. d.h. die Menge der sekundären Ereignisse ist leer und seit der letzten Auswahlphase ist kein Ereignis aufgetreten (erkennbar an `Specification::bPrimaryEvent=false`)

77. Der Listenknoten `XPtrListNode<Module> SecondaryEventQueueNode` ist dazu stark in das Modulobjekt aggregiert, so dass bei einem erneuten Einfügen automatisch die alte Listenmitgliedschaft überdeckt wird.

a. Laufzeit-Optionen: -notreesleep ***-no*imodopt -eventdrive 0**
b. Laufzeit-Optionen: -notreesleep **-imodopt -eventdrive 1**
c. Laufzeit-Optionen: -notreesleep **-imodopt -eventdrive 2**

Analysiert man nun die Auswahlsequenzen, so zeigt sich, dass der verbleibende Overhead praktisch ausschließlich auf die spontane Erzeugung von Nachrichten (a) zurückgeht. Dies wirkt sich deshalb so stark aus, da die Ereignissteuerung ja den Betrieb von Activity-Threads favorisiert und entsprechend jede vom sendenden Benutzermodul emittierte Nachricht zunächst das gesamte System durchläuft, bis keine aktiven Threads mehr vorliegen. Da nunmehr das sendende Benutzermodul als einziges noch (spontan) schaltbar ist, ergibt sich die bereits oben beschriebene Situation, dass diese Modulinstanz über eine konventionelle Suche gefunden werden muss (siehe Abb. 4-16).

In solchen Situationen wäre es vorteilhaft, wenn beim Auslaufen aller konventionellen Eventquellen nachrangig auch spontan feuernde Module gezielt favorisiert werden könnten. Die naheliegendste Lösung, einfach nach jedem Feuern eines Moduls dessen fortgesetzte Schaltbarkeit durch eine erneute Prüfung zu ermitteln, würde (unter der Annahme, dass zu einem Zeitpunkt meist nur ein kleiner Teil der Module schaltbar ist) tendenziell zu einer Verdopplung der Prüfungen und damit zu einer Beschränkung der Modulauswahleffizienz auf 50 % führen.

An dieser Stelle setzen wir eine weitere Heuristik ein: Module, die in der Vergangenheit häufig *spontan*[78] schaltbar wurden, bieten oft weiterhin spontan schaltbare Transitionen an.

Um keine zusätzlichen Transitionstests für die Ermittlung dieser „*häufigen spontanen Schaltbarkeit*“ ausführen zu müssen, wird diese Eigenschaft nur *passiv* anhand der ohnehin ausgeführten Tests betrieben. Dabei werden nur solche Modultests berücksichtigt, bei denen das Modul seit dem vorangegangenen Test nicht durch *modulexterne Ereignisse* in seiner Schaltbarkeit beeinflusst wurde.[79] Über die Quote, mit der eine Modulinstanz gemäß dieses Kriteriums seit dem letzten Test schaltbar geblieben ist oder nicht, wird dann in der Instanzvariablen „`int Module::nRepeatedly`“ ein *gleitender Mittelwert*[80] gebildet. Überschreitet diese einen vorgegebenen Schwellwert, so gilt die Modulinstanz als *spontan schaltend*.

Gemäß der o. g. Heuristik wird ein derart qualifiziertes Modul nach seinem Schalten mit hoher Wahrscheinlichkeit sofort wieder schaltbar sein. Entsprechend wird es nach dem Schalten sofort (wieder) in die Liste `SecondaryEventQueue` aufgenommen, das Schalten impliziert also gewissermaßen ein neues Ereignis im selben Modul und kann entsprechend bereits der Beginn eines neuen Activity-Threads sein.

Damit bereits laufende Activity-Threads gegenüber derartigen neuen Threads vorrangig bearbeitet werden, erfolgt die Aufnahme in die Liste im Gegensatz zu anderen Ereignissen durch Anfügen ans Ende der Liste. Somit werden erst nach Ablauf aller anderen Activity-Threads neue verfolgt, was die Ausbildung markanter und effizient verfolgbarer Ereignisketten fördert.

---

78. also ohne äußeren Stimulus im Sinne der ereignisgesteuerten Auswahl

79. Wir liefern die technischen Grundlagen für diesen Test im Abschnitt 4.2.2.6 nach.

80. Die Auswahl entsprechender Parameter ist letztlich von den spezifischen Eigenschaften der Spezifikation abhängig. Als günstiger Ausgangswert haben sich ein *Startwert* von -10, ein *Schwellwert* von 0, ein *Bonus* und *Malus* von +4 bzw. -1 und eine *Saturierung* auf das Intervall [+30 ; -30] bewährt.

Diese Optimierung[81] liefert nach einer kurzen „Lernphase“, in der die spontan schaltenden Module durch Entwicklung des gleitenden Mittelwertes detektiert werden, eine sehr zutreffende Erkennung der spontan schaltenden Modulinstanzen. Untersucht man die Auswahlsequenzen genauer, so kann man feststellen, dass dadurch bereits nach wenigen Zyklen eine ungezielte Suchphase bei der Modulauswahl (e-g in Abb. 4-16) nicht mehr auftritt und stattdessen ausschließlich gezielt das Modul mit der nächsten schaltbaren Transition gefunden wird. Es ergibt sich dabei mit 98,4 % Modulauswahleffizienz beim Sliding-Window-Benchmark ein nahezu optimaler Wert (siehe Tabelle 4.10).

Tabelle 4.10: Modulauswahleffizienz SW-Benchmark (ACTIVITY) mit Independent-Module-Optimierung

Auswahlverfahren	getestet	gefeuert	Effizienz
Basis[a]	8.178	1.510	19,0 %
independent Modules und ereignisgesteuert (einfach)[b]	2.729	1.510	55,3 %
independent Modules und ereignisgesteuert (mehrfach)[c]	2.293	1.510	65,8 %
independent Modules und ereignisgesteuert (mehrfach und wiederholt)[d]	1.535	1.510	98,4 %

a. Laufzeit-Optionen: -notreesleep ***-no*imodopt -eventdrive 0**
b. Laufzeit-Optionen: -notreesleep **-imodopt -eventdrive 1**
c. Laufzeit-Optionen: -notreesleep **-imodopt -eventdrive 2**
d. Laufzeit-Optionen: -notreesleep **-imodopt -eventdrive 3**

4.2.2.6 Erkennung von Inaktivität bei Modulinstanzen

Unsere bisherigen Überlegungen zur ereignisgesteuerten Auswahl zielten darauf ab, mit einem minimalen Overhead eine Modulinstanz zu ermitteln, die eine schaltbare Transition anbietet. Als Mittel dazu haben wir *Auswahlheuristiken* eingesetzt, die auf Basis früherer Ereignisse Modulinstanzen favorisieren, in denen eine solche Transition vermutet wird.

In bestimmten Situationen ist jedoch allein die geschickte Auswahl von Testkandidaten nicht effektiv bzgl. der Optimierung der Anzahl getesteter Module. Neben der Problematik des Versagens von Auswahlheuristiken,[82] die wir in den vorangegangenen Abschnitten mehrmals gesehen haben, ist vor allem der Übergang von der Frage nach der Existenz *eines beliebigen* schaltbaren Moduls aus einer Menge zur Frage nach *allen schaltbaren* Modulinstanzen bzw. der *Nichtexistenz einer solchen Modulinstanz* eine relevante Fragestellung.[83]

81. ebenfalls gesteuert durch das Compiler-Makro `OPT_SEC_EVENT` (für „secondary event optimization“) und die Kommandozeilenoption „`-eventdrive n`“ mit $n \geq 3$

82. Sie führt letztlich zu einer *ungerichteten* Suche.

83. Der Übergang entspricht dem von der (effizienten) Aufzählbarkeit *eines* schaltbaren Moduls zur (effizienten) Entscheidbarkeit *aller* schaltbaren Module.

Ein typisches Szenario zur letzteren Fragestellung entsteht, wenn im gesamten System keine schaltbaren Transitionen mehr vorliegen. Dies geschieht neben der endgültigen Systemtermination[84] auch bei *vorübergehender* Inaktivität, in der bis zum Ablaufen eines Timers keine schaltbaren Transitionen auftreten. Um einen solchen Zustand zu detektieren, muss die Nichtschaltbarkeit aller Module ermittelt werden.

Bei genauerer Betrachtung ist die effiziente Auswahl in diesem Szenario nur bedingt kritisch: Die zur Prüfung aufgewandte Zeit kann ohnehin nicht zum Schalten von Transitionen und damit für die eigentlichen Protokolloperationen genutzt werden, da die Auswahl bis zum Ablaufen der Timer ja per Definition erfolglos bleibt.[85] Wie wir aber bereits in Abschnitt 2.2.1 gesehen haben, gibt es neben der reinen Ausführungsdauer von Protokolloperationen auch andere Optimierungskriterien (z. B. die Minimierung der CPU-Belegung der Ausführungsplattform), so dass auch hier eine effiziente Auswahl von Nutzen sein kann.

In Umkehrung dieser Argumentation kann diese ohnehin nicht zur Ausführung der eigentlichen Protokolloperationen nutzbare Zeit auch zur *proaktiven Gewinnung von Informationen* genutzt werden, um später eine effizientere Auswahl zu ermöglichen.

Wir haben beide Fragestellungen aber auch im Zusammenhang mit der Vater-Sohn-Priorität bereits kennen gelernt und durch verschiedene Mittel zur Senkung der impliziten Modulsynchronisationen auf Spezifikationsebene entschärfen können. Ist dies nicht möglich, so muss effizient getestet werden können, ob die Menge der Vatermodule nicht schaltbar ist. Analog dazu sind bei `PROCESS`-attributierten Modulen *alle* schaltbaren Kindmodulinstanzen zu ermitteln, da diese ja synchron ausgeführt werden müssen.

Die Grundidee zur Optimierung der Fragestellungen besteht entsprechend nicht wie bisher darin *den Suchraum so weit wie möglich zu verkleinern*, um so im Extremfall einer erfolglosen Suche (wie beim Versagen der Auswahlheuristiken) oder einer zwingenden vollständigen Suche (wie bei den Kindmodulen eines `PROCESS`-attributierten Moduls) die Anzahl der zu testenden Modulinstanzen zu minimieren.

Dazu werden Modulinstanzen, die einmal als nicht schaltbar (also als „*inaktiv*“) erkannt wurden, erst dann erneut geprüft, wenn durch ein Ereignis die Schaltbarkeit beeinflusst worden sein kann. Bis zu einem solchen Ereignis müssen keine erneuten lokalen Modulauswahlen erfolgen, und entsprechend können dann auch keine Transitionen geschaltet werden.

---

84. Die Estelle-Semantik sieht keine Systemtermination im eigentlichen Sinne vor.
Es kann jedoch in Berechnungen Systemzustände geben, in denen kein neuer nächster Zustand im Sinne der *next-state*-Relation existiert, da keine Transition mehr ausgewählt oder gefeuert werden kann und auch keine Timer mehr laufen (siehe Abschnitt 2.1.2). Wir betrachten ein solches System dann als terminiert und entsprechend beendet sich dann auch die von XEC erzeugte Implementierung (siehe Abschnitt 3.5). Spontane Reaktivierungen durch indeterministische Schaltbedingungen werden dabei nicht berücksichtigt.

85. Wir gehen hier von einem *realzeitbezogenen* Ausführungsmodell aus, in dem ein direkter Zusammenhang zwischen der Ausführung von Operationen und der erforderlichen Zeit besteht. Die Estelle-Semantik hat dagegen einen deutlich schwächeren Zeitbegriff. Zudem sind *indeterministische* Aspekte der Auswahl hier nur soweit berücksichtigt, dass sie auf ein spezifisches Ergebnis beschränkt werden und somit durch mehrfache Auswertung (ohne zwischenzeitliche Schaltphase) keiner Ergebnisvariation unterliegen.

Die in Frage kommenden Ereignisse, die wir bereits zu Beginn von Abschnitt 4.2.2.2 identifiziert haben, müssen dabei im Sinne einer semi-Entscheidbarkeit zuverlässig[86] erkannt werden. Umgekehrt ist ein falsch-positives Ergebnis zwar bzgl. der Effizienz des Verfahrens unerwünscht, aber bzgl. seiner Korrektheit unkritisch.

Die Implementierung des Verfahrens[87] greift auf die ebenfalls in Abschnitt 4.2.2.2 eingeführte Methode `Module::event()` zurück, welche ja bereits zur Unterstützung der ereignisgesteuerten Auswahl beim Empfang einer Nachricht oder dem Ablaufen eines Timers der Modulinstanz aufgerufen wird.

Es bleibt also noch die Behandlung von externen Aktivierungen durch Wertänderungen von gemeinsamen Variablen. Diese können bei einem bereits inaktivierten Modul nur durch das Schalten des Vatermoduls (also bzgl. der eigenen exportierten Variablen) oder das Schalten eines Kindmoduls (also bzgl. der von ihm exportierten Variablen) verursacht werden. Diese beiden Situationen werden entsprechend in `Module::fire()` analysiert und führen ggf. ebenfalls zu einem Aufruf der Methode `Module::event()`.

Diese Analyse des Zugriffs auf gemeinsame Variablen erfolgt dabei bisher noch sehr grob, indem alle Vater- oder Kindmodule, mit denen das schaltende Modul gemeinsame Variablen besitzt, aktiviert werden. Eine Einengung dieser maximalen Zielgruppe ist jedoch auf Basis transitionsbasierter Analysen möglich (siehe Abschnitt 4.3).

Um diese neuen Ereignisse von den Nachrichten- und Delaytimer-basierten aus Abschnitt 4.2.2.2 unterscheiden zu können, wird `Module::event()` bei den variablenbezogenen Ereignissen (im Gegensatz zu den vorherigen, siehe Beispiel 4.27) mit dem optionalen Parameter `bImportant=false` aufgerufen. Anhand dieser Parametrierung werden derartige „unwichtige“ Ereignisse nicht bei der ereignisgesteuerten Modulauswahl berücksichtigt.[88]

Zur Darstellung des Inaktivitäts-Zustandes wird in der Klasse `Module` die Instanzvariable `bLocalSleep` eingeführt. Diese wird bei jeder erfolglosen lokalen Transitionsauswahl auf `true` gesetzt und bei jedem Aufruf der Methode `event()` auf `false` zurückgesetzt. Offensichtlich zeigt diese Variable die oben beschriebene Form von Inaktivität der Modulinstanz an und entsprechend kann eine lokale Transitionsauswahl unterbleiben, wenn sie gesetzt ist.

Über dieses Flag zur Anzeige der lokalen Inaktivität hinaus enthält die Klasse `Module` auch eine Instanzvariable `bSubTreeSleep`, die bei jeder erfolglosen rekursiven Auswahl im Modulteilbaum unterhalb der jeweiligen Modulinstanz auf `true` gesetzt wird. Diese Variable zeigt an, dass der gesamte hier beginnende Modulhierarchie-Teilbaum[89] inaktiv ist, und entsprechend kann die gesamte rekursive Auswahl erfolglos abgebrochen werden.

---

86. also ohne falsch-negative Ergebnisse

87. gesteuert durch das Compiler-Makro `OPT_TREESLEEP` und die Kommandozeilenoption „`-[no]treesleep`“

88. Es wäre durchaus auch eine Nutzung des Zugriffs auf exportierte Variablen zur Ereignissteuerung im Sinne der ereignisgesteuerten Auswahl denkbar. Die bisher implementierte relativ grobe Analyse der Zugriffe beeinträchtigt jedoch den Nutzen dieser Maßnahme aufgrund zu vieler Aktivierungen, die letztlich nicht zu schaltbaren Modulen führen (s. u.).

89. Als Teilbaum in diesem Sinne gilt natürlich nur der Teil, der von einer rekursiven Auswahl betroffen wäre, also ausschließlich die möglicherweise unabhängig operierenden Kindmodule und die darunter liegenden Teilbäume (siehe Abschnitt 4.2.2.3 und 4.2.2.4).

Das Zurücksetzen dieser Variablen erfolgt dabei ebenfalls beim Aufruf der Methode `Module::event()`, wobei hier ggf. die Variable auch in allen Vatermodulen (bis hinauf zum nächsten Independent-Module) zurückgesetzt werden muss, um bei einer späteren Auswahl den Zugriff in diesen Teilbaum zu ermöglichen. Dabei ist bzgl. der Auswahleffizienz zu beachten, dass beim Eingang eines Events in einen inaktiven Modulteilbaum nur bei dem unmittelbar betroffenen Modul das Flag `bLocalSleep` zurückgesetzt wird (siehe Abb. 4-18) und daher in der nächsten Auswahl auch nur dieses Modul getestet werden muss. Im Falles eines erfolglosen Tests werden dann beim Rücklauf der Modulauswahlrekursion alle `bSubTreeSleep`-Flags wieder gesetzt.

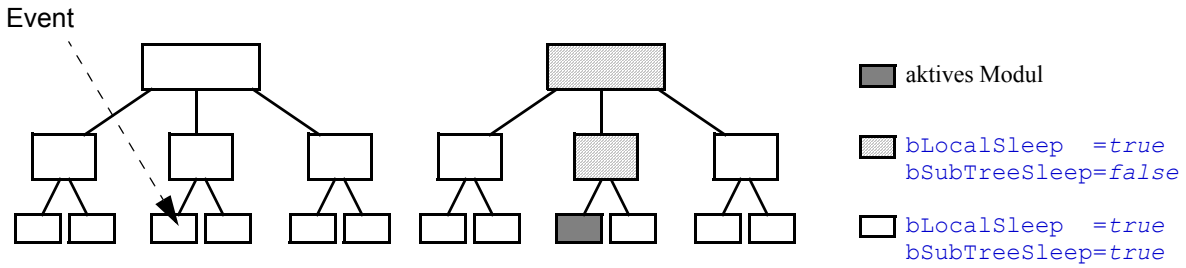


Abbildung 4-18: Wirkung eines Events auf die Aktivitäts-Flags in einem Modulteilbaum

Die Wirkung dieser Optimierung kann man sehr schön anhand des Sliding-Window-Benchmarks zeigen, sobald man eine Verzögerung beim Transport über die einzelnen Hop-Module im Netzwerkmodul aktiviert. Während in der bisher betrachteten Konfiguration (also ohne Verzögerung) die Inaktivitätserkennung erwartungsgemäß das ohnehin schon sehr gute Ergebnis der obigen Optimierungen nicht mehr verbessern kann, kommt es nun durch die ständigen Unterbrechungen im Activity-Thread zu erheblichen Störungen der ereignisgesteuerten Auswahl (siehe Szenario *d* in Abschnitt 4.2.2.5).

Die Messungen an der verzögerten Version des Benchmarks erfolgen dabei mit den folgenden Parametern, die lediglich bzgl. des von 0 ms auf 10 ms erhöhten Medium-Delays abweichen.[90] Um die Messungen deterministisch zu machen, setzen wir dabei die Zeitsimulation ein (Kommandozeilenoption „`-tsim`“, siehe Abschnitt 3.5.2).

- Anzahl der Runden: 10
- Anzahl Nachrichten pro Runde: 10
- Anzahl Hops (*H*): 3
- Sliding-Window-Fenstergröße: 20
- Medium-Delay [ms] **10**
- Retransmission-Timeout [ms] 300

Wir untersuchen zunächst die Wirkung der bisherigen Optimierungstechniken auf diese Spezifikation ohne die Inaktivitätserkennung (siehe Tabelle 4.11). Wie zu erwarten war verschlechtern sich die Auswahleffizienzen im Vergleich zu den Ergebnissen des vorherigen Benchmarks (Tabelle 4.10) erheblich (40,4 % anstatt früher 98,4 % als bestes Ergebnis).

Mit Unterstützung der Inaktivitätserkennung (siehe Tabelle 4.12) ergeben sich merkliche Steigerungen, die jedoch nicht mehr die Ergebnisse bei dem unverzögerten Benchmark erreichen (58,7 % als bestes Ergebnis).

90. Dadurch wird jedoch auch die Anzahl der geschalteten Transitionen beeinflusst.

Tabelle 4.11: Modulauswahleffizienz SW-Benchmark (ACTIVITY, verzögert) *ohne* Inaktivitätserkennung

Auswahlverfahren	getestet	gefeuert	Effizienz
Basis[a]	11.150	1.710	15,3 %
independent Modules und ereignisgesteuert (einfach)[b]	4.392	1.710	38,9 %
independent Modules und ereignisgesteuert (mehrfach)[c]	4.626	1.710	37,0 %
independent Modules und ereignisgesteuert (mehrfach und wiederholt)[d]	4.237	1.710	40,4 %

a. Laufzeit-Optionen: -tsim -*notreesleep* **-*no*imodopt -eventdrive 0**
b. Laufzeit-Optionen: -tsim -*notreesleep* **-imodopt -eventdrive 1**
c. Laufzeit-Optionen: -tsim -*notreesleep* **-imodopt -eventdrive 2**
d. Laufzeit-Optionen: -tsim -*notreesleep* **-imodopt -eventdrive 3**

Tabelle 4.12: Modulauswahleffizienz SW-Benchmark (ACTIVITY, verzögert) *mit* Inaktivitätserkennung

Auswahlverfahren	getestet	gefeuert	Effizienz
Basis[a]	4.265	1.710	40,1 %
independent Modules und ereignisgesteuert (einfach)[b]	3.125	1.710	54,7 %
independent Modules und ereignisgesteuert (mehrfach)[c]	3.300	1.710	51,8 %
independent Modules und ereignisgesteuert (mehrfach und wiederholt)[d]	2.911	1.710	58,7 %

a. Laufzeit-Optionen: -tsim -*treesleep* **-*no*imodopt -eventdrive 0**
b. Laufzeit-Optionen: -tsim -*treesleep* **-imodopt -eventdrive 1**
c. Laufzeit-Optionen: -tsim -*treesleep* **-imodopt -eventdrive 2**
d. Laufzeit-Optionen: -tsim -*treesleep* **-imodopt -eventdrive 3**

Die Ursache dafür ist darin zu suchen, dass die Inaktivitätssteuerung nur Wirkung bei Modulen zeigt, die häufiger negativ als positiv getestet werden. Sie kann insbesondere nur dann Modultests einsparen, wenn tatsächlich ein inaktives Modul getestet werden soll, also dieses Modul mindestens zweimal hintereinander negativ getestet würde[91]. Gerade diese Szenarien werden jedoch durch die früheren Optimierungsmaßnahmen reduziert, so dass die Kombination der Optimierungstechniken oft nur begrenzten Erfolg hat. Entsprechend wird der größte Speedup durch die Inaktivitätssteuerung bei ansonsten unoptimierten Auswahlverfahren (Basis-Auswahlverfahren) erreicht.

Weiterhin ist in Systemen, die eine einigermaßen gleichmäßige Test- und Schaltaktivität über alle Module zeigen, nur dann eine deutliche Verbesserung zu erwarten, wenn die Auswahleffizienz ohne die Inaktivitätssteuerung deutlich unter 50 % lag.

91. genauer: Wenn zwischenzeitlich zudem auch keine Ereignisse auf das Modul eingewirkt haben.

Um dies zu verdeutlichen, ändern wir die Benchmarkspezifikation so ab, dass neben dem bisherigen aktiven Kommunikationsszenario vier weitere identische aber im Beobachtungszeitraum weitgehend inaktive Kommunikationsszenarien[92] im Spezifikationsmodul existieren. Derartige Situationen entstehen in realen Systemen häufig aufgrund der unterschiedlichen Beanspruchungen und Aktivitäten von Kommunikationspfaden und den zugehörigen Protokollinstanzen.

Dadurch entstehen neben den 10 bisherigen Modulinstanzen noch 36 weitere, die im konkreten Fall während der Benchmarkausführung keine schaltbaren Transitionen anbieten. Ohne die Inaktivitätssteuerung verschlechtert sich das Ergebnis nun deutlich von 15,3 % auf 2,2 % (Basisauswahl) bzw. 40,4 % auf 14,9 % (volle Optimierung, siehe Tabelle 4.13).

Tabelle 4.13: Modulauswahleffizienz SW-Benchmark (ACTIVITY, verzögert, teilinaktiv) *ohne* Inaktivitätserkennung

Auswahlverfahren	getestet	gefeuert	Effizienz
Basis[a]	76.630	1.710	2,2 %
independent Modules und ereignisgesteuert (mehrfach und wiederholt)[b]	11.473	1.710	14,9 %

a. Laufzeit-Optionen: -tsim -*notreesleep* **-*no*imodopt -eventdrive 0**
b. Laufzeit-Optionen: -tsim -*notreesleep* **-imodopt -eventdrive 3**

Durch die wesentlich größere Lokalität der schaltbaren Module greift dagegen die Inaktivitätssteuerung optimal und erreicht praktisch die selben Ergebnisse wie in der früheren Spezifikation (siehe Tabelle 4.14).

Tabelle 4.14: Modulauswahleffizienz SW-Benchmark (ACTIVITY, verzögert, teilinaktiv) *mit* Inaktivitätserkennung

Auswahlverfahren	getestet	gefeuert	Effizienz
Basis[a]	4.301	1.710	39,8 %
independent Modules und ereignisgesteuert (mehrfach und wiederholt)[b]	2.947	1.710	58,0 %

a. Laufzeit-Optionen: -tsim -*treesleep* **-*no*imodopt -eventdrive 0**
b. Laufzeit-Optionen: -tsim -*treesleep* **-imodopt -eventdrive 3**

Dieses Ergebnis stellt sich übrigens in ähnlicher Form ebenfalls ein, wenn die Module nicht dauerhaft, sondern zeitweise inaktiv sind: Nach einer kurzen Lernphase (alle Instanzen werden einmal getestet) haben diese Module für den Zeitraum ihrer Inaktivität keinen negativen Einfluss mehr auf die Performance des Gesamtsystems.

Eine Einflussgröße, die in den bisherigen Benchmarks noch keine Rolle gespielt hat, ist die Auswirkung von exportierten Variablen auf die Effektivität der Inaktivitätssteuerung. Offensichtlich vergrößern diese die Anzahl der durch ein Ereignis aktivierten Modulinstanzen (s. o.) und damit auch die Anzahl der zu testenden Module. Durch eine präzisere Auswertung der potentiell betroffenen Variablen und Modulinstanzen kann die Auswirkung zwar verringert werden, grundsätzlich sollten jedoch exportierte Variablen auf ein unvermeidliches Minimum be-

92. Also jeweils ein kompletter Satz an Benutzermodulen, Protokollmaschinen und Basisdienstmodulen.

schränkt und vorzugsweise durch nachrichtenbasierte Synchronisation ersetzt werden, zumal dies auch das Potential für mögliche Parallelimplementierungen vergrößert. Wir kommen in Abschnitt 4.3 nochmals auf diese Fragestellung zurück.

4.2.2.7 Einwirkung systemfremder Ereignisse

Unsere Betrachtungen zur ereignisgesteuerten Optimierungen des Auswahlverfahrens basierten bisher ausschließlich auf dem durch die Estelle-Semantik vorgegebenen Modell zur Zustandsmanipulation. Bei der praktischen Anwendung automatisch generierter Implementierungen ist jedoch gelegentlich die Interaktion mit darüber hinausgehenden Ereignisquellen erforderlich.

Ein typisches Beispiel ergibt sich bei der Ankopplung eines in Estelle spezifizierten Systems an Kommunikationsmechanismen der Implementierungsplattform[93], z. B. die Kommunikation über UNIX-Sockets. Dies wird häufig pragmatisch durch die Definition *primitiver Funktionen und Prozeduren* und deren Einsatz u. a. als Schaltbedingungen realisiert, welche dann durch Polling diese externen Ereignisse detektieren sollen.

Solche Zugriffe auf *systemfremde Zustandsraumkomponenten und Ereignisquellen* liegen zunächst außerhalb des Ereignismodells der von XEC erzeugten Implementierung, d.h. Änderungen in der Schaltbarkeit von Transitionen werden nicht mehr notwendigerweise ausschließlich von systeminternen Ereignissen induziert. Entsprechend kann z. B. durch die oben diskutierte Inaktivitätssteuerung ein Modul, dessen Aktivität von solchen externen Ereignissen abhängt, vom Auswahlverfahren irrtümlich als *dauerhaft inaktiv* erkannt werden, so dass externe Ereignisse nicht mehr wahrgenommen werden. Hier sind gegebenenfalls zur Sicherung der Korrektheit spezifischere Anpassungen an das Laufzeitsystem oder die Deaktivierung der Inaktivitätssteuerung erforderlich.[94] Eine Integration in das Ereignismodell von XEC kann darüber hinaus auch zur effizienten Anbindung dieser externen Ereignisse dienen und so z. B. den Start eines neuen Activity-Threads als Folge eines externen Ereignisses unterstützen.

Ein anderes interessantes Beispiel ist der Zugriff auf eine *Echtzeituhr* in der Estelle-Spezifikation von XTP 4.0. Diese wird in einer primitiven Funktion[95] für XEC auf die Methode `Timer::now()` abgebildet, welche Zugriff auf die dem Estelle-System zu Grunde liegende Uhr[96] in der Einheit der Systemzeitskala bietet (siehe Abschnitt 3.5.2). Wird eine solche Methode über eine primitive Funktion in einer Schaltbedingung verwendet, so kann auch hier ohne Einwirken eines systeminternen Ereignisses eine Änderung der Schaltbarkeiten in einem Modul erfolgen, indem z. B. Klauseln wie „`PROVIDED now() >= tTimeout`“ verwendet werden. Auch hier sind die Interaktionen mit der Inaktivitätssteuerung genau zu untersuchen.[97]

---

93. Siehe auch die Diskussion zur Umgebungsinteraktion in der Einführung zu Open-Estelle in Kapitel 7.

94. z. B. per Kommandozeilenoption „`-notreesleep`“

95. `FUNCTION local_time:INTEGER`

96. siehe auch Abschnitt 5.3.5 in [ISO97]

97. In der konkreten XTP-Spezifikation geht von der primitiven Funktion keine Gefahr aus, da sie nicht direkt oder indirekt aus einer Transitions*klausel* heraus aufgerufen wird. Der Aufruf aus einem Transitions*rumpf* heraus ist dagegen unkritisch, da hier ohnehin ein Zustandsübergang erwartet wird. Entsprechend wäre es sinnvoller, in der XTP-Spezifikation diesen Zugriff über eine (primitive) *non-pure*-Prozedur zu realisieren, da dadurch explizit gekennzeichnet wäre, dass sie nicht aus einer Transitionsklausel heraus aufgerufen werden kann.

4.3. Modullokale Transitionsauswahl

Im vorangegangenen Abschnitt 4.2 haben wir verschiedene Techniken zur Effizienzsteigerung bei der *globalen Auswahl* – also der Auswahl eines Moduls mit schaltbaren Transitionen – untersucht. Nun wenden wir uns der *modullokalen Auswahl* zu, also der Frage nach der effizienten Bestimmung einer schaltbaren Transition innerhalb einer vorgegebenen Modulinstanz.

Wir werden dabei sehen, dass sich einige der Grundideen der globalen Auswahl durchaus auf die modullokale Auswahl übertragen lassen, häufig jedoch das Erreichen einer optimalen Auswahleffizienz durch die meist ungleich höhere Verschränkung der Abhängigkeiten zwischen den Transitionen eines Moduls beeinträchtigt wird.

4.3.1 Auswahleffizienz

In Abschnitt 4.2 haben wir die *globale Modulauswahleffizienz* Eff_{mod} als Quotienten zwischen der Anzahl der geschalteten (nm_{fired}) und der getesteten *Modulinstanzen* (nm_{tested}) definiert:

$$Eff_{mod} = nm_{fired} / nm_{tested}$$

Analog führen wir nun als Qualitätsindikator für die Transitionsauswahl die *(globale) Transitionsauswahleffizienz* Eff_{trans} als Quotient zwischen der Anzahl der geschalteten (nt_{fired}) und der getesteten Transitionen (nt_{tested}) ein:

$$Eff_{trans} = nt_{fired} / nt_{tested}$$

Hierbei ist zu beachten, dass die modullokale Auswahl immer nur null oder eine Transition liefert und die globale Auswahl eine als schaltbar identifizierte Transition auch schließlich immer schaltet ($nm_{fired} = nt_{fired}$), für eine einzelne lokale Modulauswahl also $nt_{fired} \in \{0, 1\}$ gilt.[98] Entsprechend nutzen wir den Begriff der Transitionsauswahleffizienz im Allgemeinen nicht für eine einzelne lokale Auswahl, sondern nur für einen ganzen globalen Auswahlzyklus oder eine Menge von Auswahlzyklen (z. B. einen gesamten Systemlauf), um aussagekräftige Werte zu erhalten.

4.3.1.1 Modullokale Transitionsauswahleffizienz

Betrachten wir nur *eine einzelne lokale Transitionsauswahl*, so ist die Anzahl (nt_{tested}) getesteter Transitionen eine aussagekräftigere Größe. Wie wir später sehen werden, ergeben sich oft fundamental unterschiedliche Ergebnisse bei *erfolgreichen* oder *erfolglosen* modullokalen Auswahlen. Daher differenzieren wir diesen Wert in beiden Fällen zu nt^{+}_{tested} (also der Anzahl der zu testenden Transitionen, um *eine schaltbare* Transition zu finden) und nt^{-}_{tested} (also der Anzahl der zu testenden Transitionen um nachzuweisen, dass *keine schaltbare* Transition existiert).

98. Also gilt $nm_{fired} = nt_{fired}$. Dazu werden u. a. die Modulinstanzen immer in der Reihenfolge absteigender Prioritäten getestet, so dass ein einmal als schaltbar erkanntes Modul auch immer geschaltet werden kann.

Dividiert durch die Gesamtzahl der Transitionen des Moduls (*nt*) kann man so bereits einen Effizienzbegriff im Sinne eines *Optimierungsgewinns* für einzelne lokale Transitionsauswahlen bilden, der von der Anzahl der geschalteten Transitionen unabhängig ist:

$$opt^{+}_{trans} = 1 \angle \frac{nt^{+}_{tested}}{nt} \quad \text{bzw.} \quad opt^{-}_{trans} = 1 \angle \frac{nt^{-}_{tested}}{nt}$$

Dieser Effizienzbegriff bewertet letztlich die Quote der nicht getesteten Transitionen eines Moduls und ist besonders für eine analytische Bewertung eines Auswahlverfahrens geeignet.

Für eine gegebene Wahrscheinlichkeit *p* für eine erfolgreiche Auswahl ergibt sich ein mittleres nt_{tested} als gewichteter Mittelwert und somit ein entsprechendes opt_{trans}:

$$nt_{tested} = p \cdot nt^{+}_{tested} + (1 \angle p) \cdot nt^{-}_{tested}$$

$$opt_{trans} = 1 \angle \frac{nt_{tested}}{nt}$$

In all diesen Fällen wirken offensichtlich die Effizienz der modullokalen Transitionsauswahl und die globale Transitionsauswahleffizienz zusammen, da eine gute globale Auswahleffizienz dazu führt, dass weniger Module und somit insgesamt auch weniger Transitionen erfolglos getestet werden müssen (siehe auch Abschnitt 4.3.7).

4.3.1.2 Globale Transitionsauswahleffizienz

Ein anderer Zugang zur Effizienz der modullokalen Auswahlen ist über die Gesamtzahl der getesteten Transitionen über *eine Menge von Systemzyklen und Modulen* (z. B. das gesamte System über die gesamte Laufzeit) möglich, wie sie z. B. als statistische Ergebnisse eines Benchmark-Laufs gewonnen werden können.

Hier kann man die *mittlere modullokale Transitionsauswahleffizienz* eff_{trans} als Quotient aus den getesteten Modulen (nm_{tested}) und den der dabei jeweils getesteten Transitionen (nt_{tested}) definieren:

$$eff_{trans} = nm_{tested} / nt_{tested}$$

Dieser Wert ist also der Kehrwert der Anzahl der Transitionen, die in den von der globalen Auswahl vorgegebenen Modulen im Schnitt getestet wurden und berücksichtigt dabei auch das Verhältnis der erfolgreichen und erfolglosen modullokalen Auswahlen, wobei auf gefeuerte Transitionen keinerlei Bezug genommen wird.

Der selbe Wert kann über mehrere Auswahl- und Ausführungszyklen hinweg wegen $nm_{fired} = nt_{fired}$ auch als Quotient der entsprechenden Effizienzen berechnet werden:

$$eff_{trans} = Eff_{trans} / Eff_{mod} \qquad \text{also} \qquad Eff_{trans} = eff_{trans} \cdot Eff_{mod}$$

Somit ergibt sich die globale Transitionsauswahleffizienz als Produkt der mittleren modullokalen Transitionsauswahleffizienz und der globalen Modulauswahleffizienz.

Bei dieser Darstellung ist zu bedenken, dass die Effizienz der globalen Modulauswahl *nicht orthogonal* zur mittleren modullokalen Transitionsauswahleffizienz ist. Entsprechend ist eff_{trans} auch von den spezifischen Eigenschaften der globalen Modulauswahl abhängig. So sind – wie wir sehen werden – z. B. erfolglose lokale Auswahlen meist aufwändiger als erfolgreiche. Entsprechend führt eine schlechte Modulauswahleffizienz, die vermehrt erfolglose lokale Transitionsauswahlen verursacht, auch zu einer schlechteren lokalen Transitionsauswahleffizienz und belastet damit die globale Transitionsauswahleffizienz doppelt (siehe Abschnitt 4.3.7).

Wird diese Größe der im Mittel bei einer lokalen Auswahl getesteten Transitionen *analytisch abgeschätzt*, so ist auch hier eine Aufspaltung in die Anteile bei erfolgreicher ($\overline{eff}^{+}_{trans}$) und erfolgloser Auswahl ($\overline{eff}^{-}_{trans}$) sinnvoll. Der Zusammenhang der Werte ergibt sich dann ebenfalls aus der Wahrscheinlichkeit *p* einer erfolgreichen Auswahl:

$$eff_{trans} = p \cdot \overline{eff}^{+}_{trans} + (1 \angle p) \cdot \overline{eff}^{-}_{trans}$$

Hierbei ist jedoch zu beachten, dass die o. g. statistisch ermittelten Werte meist den Durchschnitt einer Vielzahl von Modulen bilden, während die analytisch ermittelten Werte sich typischerweise auf ein einzelnes Modul beziehen.

Auch nimmt diese Größe keinen Bezug auf die Anzahl der in Frage kommenden Transitionen, sondern zeigt nur an, wie viele Transitionen pro Modulinstanz getestet wurden. ggf. ist hier eine entsprechende Bewertung anhand einer unoptimierten Referenzimplementierung (z. B. im Sinne eines *Speedups*) sinnvoll.

4.3.1.3 Rechenzeitbezogene Effizienz

Neben der besprochenen Auswahleffizienz ist natürlich auch der *Rechenzeitaufwand* zur Bestimmung schaltbarer Transitionen ein relevantes Kriterium. Hier spielt zusätzlich der Anteil des Overheads zur Durchführung der jeweiligen Optimierungsmaßnahmen eine Rolle. Hat in einer Auswahlphase ein einzelner Transitionstest im Mittel einen Zeitbedarf von $\overline{t_{test}}$, so ergibt sich bei sequentieller Bearbeitung zusätzlich ein zeitlicher Overhead $t_{testoverhead}(n)$ für den Test von *n* Transitionen, insgesamt also

$$T_{test}(n) = n \cdot \overline{t_{test}} + t_{testoverhead}(n)$$

Dabei besteht häufig kein linearer Zusammenhang zwischen n und dem Overhead, weshalb wir letzteren hier zunächst als allgemeine Funktion formulieren.

Analog ergibt sich beim Ausführen von *n* Transitionen ein Zeitbedarf von

$$T_{fire}(n) = n \cdot \overline{t_{fire}} + t_{fireoverhead}(n)$$

Die rechenzeitbezogene Effizienz ist somit

$$\frac{T_{fire}(nt_{fired})}{T_{fire}(nt_{fired}) + T_{test}(nt_{tested})} = 1 \angle \frac{T_{test}(nt_{tested})}{T_{fire}(nt_{fired}) + T_{test}(nt_{tested})}$$

Diese gibt an, wie groß der Gesamtzeitbedarf des Schaltens von Transitionen an der gesamten Ausführungszeit ist (linke Seite) bzw. (in Analogie zu Abschnitt 4.3.1.1) welcher Zeitanteil nicht mit Auswahlen verbracht wurde (rechte Seite).[99] Im Idealfall geht diese Effizienz gegen 100 %.[100] Die beiden Größen $T_{test}(n)$ und $T_{fire}(n)$ können direkt vom integrierten Statistikmodul des XEC-Laufzeitsystems (siehe Abschnitt 3.6.2) ausgegeben werden und sind somit leicht zu bestimmen.

---

99. Zeiten, die sich aus dem systemweiten „Abwarten“ eines Delay-Timeouts ohne Schaltaktivitäten ergeben, sind hier nicht berücksichtigt.

100. Alternativ kann hier der Overhead beim Feuern der Transitionen statt dem Zähler dem Nenner zugeschlagen werden.

4.3.2 Optimierungsansätze

Wir beginnen analog zur globalen Auswahl (Abschnitt 4.2) mit der Untersuchung der Effizienz der Transitionsauswahl in Bezug auf das Zahlenverhältnis der getesteten zu den geschalteten Transitionen.

In Abschnitt 3.4 haben wir bereits die grundlegenden in XEC realisierten Mechanismen zum Testen von Transitionen auf Schaltbarkeit und die darauf aufbauende modullokale Auswahl kennen gelernt. Dazu haben wir ein *Referenzauswahlverfahren* definiert, welches sequentiell die Liste der Transitionsinstanzen der betrachteten Modulinstanz durchläuft, jeweils die Methode „`bool SimpleTrans::test()`“ aufruft und so prüft, ob und welche Transitionen bereit[101] sind. Aus diesen wird eine Transition mit maximaler Priorität auswählt (siehe Beispiel 3.22-a auf Seite 91).

Dieses Auswahlverfahren ist universell, es testet jedoch immer alle Transitionen. Damit ist in einer Modulinstanz mit *n* Transitionsinstanzen eine lokale Auswahleffizienz eff_{trans} von *1/n* zu erwarten. So ergibt sich z. B. in einem System mit jeweils 10 Transitionen pro Modul eine lokale Auswahleffizienz von 10 %, d.h. selbst bei einer optimalen globalen Modulauswahleffizienz von 100 % beträgt die globale Transitionsauswahleffizienz maximal 10 %.

Ausgehend von unseren Erfahrungen bei der globalen Modulauswahl untersuchen wir nun zwei grundsätzliche Vorgehensweisen zur Verbesserung der Auswahleffizienz:

(i) die *Verkleinerung des Suchraums* durch den Ausschluss nicht in Frage kommender Transitionen und

(ii) die Optimierung der *Transitionstest-Reihenfolge*, um möglichst nicht sämtliche in Frage kommenden Transitionen testen zu müssen.

Beide Vorgehensweisen haben auf den ersten Blick ähnliche Eigenschaften, da sie die Anzahl der zu testenden Transitionen verringern sollen, sie unterscheiden sich jedoch insbesondere anhand ihrer Wirkung auf *inaktive* Module: Während bei einem Modul, das keine schaltbare Transition anbietet, eine *Verkleinerung des Suchraums (i)* die Anzahl der zu testenden Transitionen effektiv verringert, hat eine optimierte *Transitionstest-Reihenfolge (ii)* in diesem Fall keine positive Wirkung. Erst beim Test eines *aktiven* Moduls (also eines Moduls, das mindestens eine schaltbare Transition enthält) kann letztere ihre Vorteile ausspielen. Die bedingte Wirksamkeit dieses zweiten Verfahrens ist auch die Ursache dafür, dass erfolglose modullokale Auswahlen aufwändiger sind als erfolgreiche (s. o.).

Im Grunde beruhen alle von uns untersuchten modullokalen Optimierungsverfahren auf diesen beiden Grundideen, die in ähnlicher Form bereits bei der globalen Auswahl gute Erfolge gezeigt haben. Wie auch dort gibt es jedoch Abhängigkeiten zwischen den auszuwählenden Komponenten (hier den Transitionen eines Moduls), die der Anwendung dieser Prinzipien entgegenstehen. Wir untersuchen diese Effekte im Folgenden anhand einiger konkreter Auswahlverfahren in XEC.

Zur kompakten Implementierung und flexiblen Nutzung verschiedener Transitionsauswahlverfahren wird in der XEC-Laufzeitbibliothek jedes Verfahren in einer eigenen Klassendefinition als Spezialisierung der Basisklasse `LocalTransSelector` realisiert (siehe Abschnitt 3.4.2). Jede Modulinstanz referenziert dann eine eigene Instanz einer dieser Auswahlklassen, in der u. a. auch dynamisch oder vorab ermittelte Profilinformationen[102] zur Op-

101. Die Methode `SimpleTrans::test()` berücksichtigt auch die Erfülltheit einer möglichen `DELAY`-Klausel (in ihrer Spezialisierung in `DelayedTrans`, siehe Abschnitt 3.3.7.7).

timierung der Auswahl gesammelt werden können. Insbesondere kann man gegebenenfalls für jede Modulinstanz ein *individuelles Auswahlverfahren* nutzen. Zudem können diese Auswahlobjekte auch dynamisch zur Laufzeit ausgetauscht werden, um z. B. auf wechselnde Verhaltensweisen oder Erkenntnisse aus der Verhaltensanalyse reagieren zu können. Die Basisklasse `LocalTransSelector` implementiert dabei als Default das Referenzauswahlverfahren.

4.3.3 Auswahloptimierungen und `DELAY`-Transitionen

Die Grundidee der Auswahloptimierung besteht – wie wir gesehen haben – darin, über verschiedene Wege bei der modullokalen Auswahl nicht mehr alle Transitionen zu testen. Daraus ergibt sich jedoch ein kritisches Problem bzgl. der Steuerung der Timer von `DELAY`-Transitionen.

Aus der Estelle-Semantik ergibt sich, dass bei einer lokalen Auswahl auch die Timer aller `DELAY`-Transitionen gemäß der aktuellen Schaltbereitschaft[103] ggf. gestartet oder gestoppt werden müssen. Da im XEC-Laufzeitsystem das *Ablaufen* des Timers zentral erfasst und behandelt wird (siehe Abschnitt 3.5.2), genügt es, bei einer Änderung der Schaltbereitschaft den Start bzw. Stop des Timers zu initiieren (siehe auch Abb. 3-14 auf Seite 88). Diese Funktionalität ist in die Methode `DelayedTrans::test()` bereits integriert, so dass eine entsprechende Prüfung der `DELAY`-Transitionen genügt.

Die *Einsparung* des Tests einer `DELAY`-Transition bei der lokalen Auswahl ist daher nur dann bzgl. der Konformität mit der Estelle-Semantik unkritisch, wenn sich ihre Schaltbereitschaft seit der letzten Auswahl nicht geändert hat. Dies zu ermitteln erfordert jedoch im Allgemeinen gerade den Test der Transition.

Bei Verfahren zur Optimierung der *Transitionstest-Reihenfolge* bedeutet dies im Allgemeinen, dass sämtliche in Frage kommenden `DELAY`-Transitionen getestet werden müssen, obwohl man möglicherweise bereits einen (besseren) Kandidaten gefunden hat. Dies widerspricht natürlich der Grundidee dieses Optimierungsansatzes. Ein typisches Beispiel ergibt sich bei `DELAY`-Transitionen, die aufgrund geringerer Priorität[104] bzgl. einer anderen schaltbaren Transition gar nicht ausgewählt werden können (siehe z. B. Abschnitt 4.3.4 zur Prioritätssteuerung).

Bei Verfahren zur *Verkleinerung des Suchraums* müssen die Timer der dabei *ausgeschlossenen* `DELAY`-Transitionen ggf. gestoppt werden.[105] Ein typisches Beispiel ist hier die tabellengesteuerte Auswahl bzgl. des Kontrollzustandes (siehe Abschnitt 4.3.5). Hier müssen bei einer Auswahl nach einem Kontrollzustandswechsel alle dabei deaktivierten `DELAY`-Transitionen dennoch getestet werden.

---

102. siehe auch Diskussion zu *Profildaten* in Abschnitt 3.5.4.2

103. Eine Transition ist *(schalt-) bereit* („*enabled*“), wenn ihre `WHEN`-, `PROVIDED`- und `FROM`-Klauseln erfüllt sind (siehe Abschnitt 9.6.2 von [ISO97] und Abschnitt 3.4).

104. Die Priorität einer Transition hat keinen Einfluss auf die Schaltbereitschaft einer Transition und somit auch nicht auf die Steuerung der Delay-Timer.

105. Wir gehen hier davon aus, dass die ausgeschlossenen Transitionen nachweislich nicht bereit sind (z. B. wegen unerfüllter `FROM`-Klausel). Dies kann jedoch z. B. im Falle eines Ausschlusses auf Grund von Prioritätsbeziehungen unzutreffend sein. Für die in diesem Text betrachteten Optimierungen ist die Annahme jedoch gültig.

Die zuverlässige und effiziente Erkennung solcher Sonderfälle stellt offensichtlich bei komplexeren Auswahlverfahren eine nicht-triviale Herausforderung dar. Aus diesen Gründen schließen wir in den folgenden Betrachtungen DELAY-Transitionen aus der Auswahloptimierung aus, indem sie in einer separaten Liste geführt und in jeder lokalen Transitionsauswahl zwingend (d.h. über das Referenzauswahlverfahren) getestet werden. Nach der Anwendung des jeweiligen optimierten Auswahlverfahrens auf die übrigen Transitionen wird dann ggf. zwischen den Ergebnissen beider Auswahlen eine Transition maximaler Priorität gewählt (Abb. 4-19).[106]

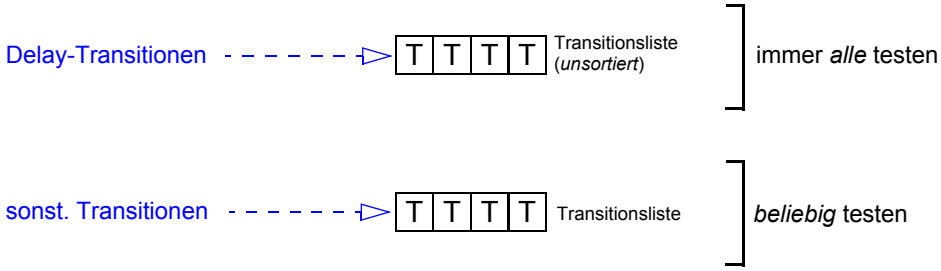


Abbildung 4-19: Sonderbehandlung von DELAY-Transitionen bei der Auswahl

Diese Vereinfachung reduziert die Komplexität der lokalen Auswahlverfahren um den Preis, dass die DELAY-Transitionen in jeder lokalen Auswahl getestet werden und somit die erreichbare Effizienz abhängig von deren Anzahl verringert wird. Es ist entsprechend vorteilhaft, wenn in einem System möglichst wenige DELAY-Transitionsinstanzen existieren bzw. diese sich in Modulinstanzen befinden, in denen selten eine Auswahl stattfindet. Ein Negativbeispiel wäre hier entsprechend ein stark frequentiertes Modul oder (bei wirksamer[107] Vater-Sohn-Priorität) dessen Vatermodul.

Betrachten wir dazu den Anteil der DELAY-Transitionsinstanzen in verschiedenen Benchmarkspezifikationen (siehe Tabelle 4.15). Hier zeigt sich, dass in großen Spezifikationen wie dem XTP 4.0-Protokoll der Anteil der DELAY-Transitionsinstanzen mit 4,9 % verhältnismäßig gering ausfällt. Im Sliding-Window-Benchmark sind dagegen 41 % der Transitionsinstanzen DELAY-Transitionen. Die Ursache dafür liegt in der in der Spezifikation realisierten *selektiven Paketwiederholung*, durch die für jedes unquittierte Paket ein eigener Timer benötigt wird. Daraus ergeben sich bei der im Beispiel verwendeten Sliding-Window-Fenstergröße von 20 gerade die 20 DELAY-Transitionsinstanzen[108] (siehe Abschnitt 2.3.3).

Wir weisen bei den im Folgenden untersuchten Optimierungsmaßnahmen den Anteil der DELAY-Transitionen bei der Transitionsauswahl jeweils separat aus und kommen dann in Abschnitt 4.3.8 nochmals auf die Optimierung der DELAY-Transitionsauswahl zurück.

106. Diese Sonderbehandlung erfolgt innerhalb der bisher implementierten Spezialisierungen der Basisklasse LocalTransSelector und kann entsprechend durch andere Auswahlverfahren abweichend gehandhabt werden.

107. Wenn diese nicht z. B. durch die Independent-Module-Erweiterung (siehe Abschnitt 4.2.2.4 auf Seite 141) aufgehoben wurde.

108. Die Timer dieser DELAY-Transitionen im Sliding-Window-Benchmark werden zwar regelmäßig gestartet und gestoppt, aufgrund des Anwendungsszenarios (kein Verlust im Basisdienst) laufen sie jedoch nie ab.

Tabelle 4.15: Anteil DELAY-Transitionen an allen Transitionen

Benchmark	Anteil DELAY-Transitionen bzgl. ...			
	Transitions-definitionen		Transitions-instanzen	
Ping-Pong[a]	0 / 13	*= 0 %*	0 / 43	*= 0 %*
Sliding-Window[b]	3 / 25	*= 12 %*	20 / 49	*= 41 %*
XTP 4.0[c]	16 / 219	*= 7,3 %*	29 / 587	*= 4,9 %*

a. Parameter gemäß Abschnitt 4.2.2.2
b. Parameter gemäß Abschnitt 4.2.2.5 (kein Medium-Delay)
c. Parameter gemäß Abschnitt 4.3.7.3

4.3.4 Prioritätsgesteuerte Auswahl

Wir beginnen unsere Untersuchung mit einer Methode zur *Optimierung der Transitionstest-Reihenfolge*. Diese baut auf dem o. g. Referenzauswahlverfahren auf. Ziel ist es dabei, die modullokale Suche nach einer schaltbaren Transition so zu gestalten, dass möglichst *nicht alle* Transitionen getestet werden müssen, sondern die Suche bereits vorzeitig (erfolgreich) abgebrochen werden kann.

Hintergrund des Prioritätsauswahlverfahrens ist, dass von den Transitionen, deren FROM-, WHEN-, PROVIDED und DELAY-Klauseln erfüllt sind, nur eine solche ausgewählt werden kann, die maximale Priorität hat. Sortiert man entsprechend die Transitionen nach absteigender Priorität[109], so kann die Suche beim ersten Treffer erfolgreich abgebrochen werden (Abb. 4-20).

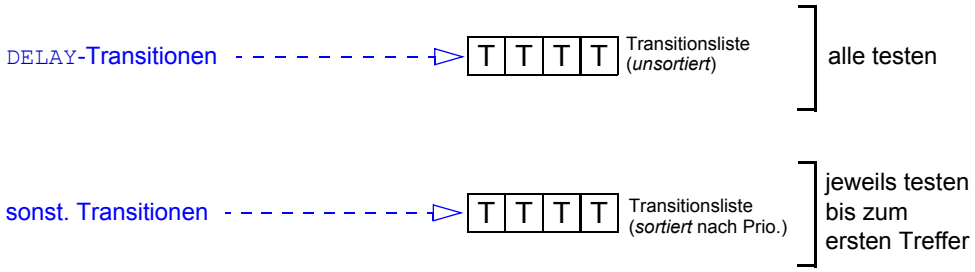


Abbildung 4-20: Struktur der prioritätsgesteuerten Auswahl

Die einfachste Implementierung ergibt sich bei einem Modul, in dem überhaupt keine Transitionen mit PRIORITY-Klauseln auftreten.[110] Die Liste der Transitionen ist hier implizit bereits prioritätssortiert, und somit genügt ein entsprechender Abbruch der Suchschleife bei Erfolg. Gibt es unterschiedliche Prioritäten, so genügt es zur Implementierung auf Basis des Referenz-Implementierungsverfahrens, die Liste der Transitionen absteigend nach Priorität zu sortieren.

109. also nach aufsteigenden Werten der Parameter der Prioritätsklauseln (siehe Abschnitt 3.3.7.4)

110. angezeigt durch den Modul-Hint HINT_NOPRIO (siehe Abschnitt 3.4.4)

Im Ping-Pong- und dem Sliding-Window-Benchmark ist ein explizites Sortieren entsprechend nicht erforderlich, da keinerlei PRIORITY-Klauseln vorkommen, im XTP-Benchmark gilt dies jedoch nur für 60 % der Moduldefinitionen (siehe Tabelle 3.3 auf Seite 96).

Die Wirksamkeit dieses Optimierungsverfahrens hängt davon ab, an welchem Punkt der prioritätssortierten Liste eine schaltbare Transition gefunden wird. Gehen wir zunächst von einer Gleichverteilung über alle *n* Transitionen aus, so wird bei einer erfolgreichen Auswahl im Mittel nach der Hälfte der Transitionstests eine schaltbare Transition gefunden, es ergibt sich also eine lokale Effizienz eff^{+}_{trans} von *2/n*, die Anzahl der erforderlichen Tests hat sich im Vergleich zum Referenzmodell halbiert. Bei erfolgloser Auswahl müssen jedoch weiterhin alle Transitionen getestet werden, und es ergibt sich eine lokale Effizienz eff^{-}_{trans} von *1/n*.

Die jeweilige Prioritätsverteilung kann jedoch erheblichen (insbesondere negativen) Einfluss auf das Verfahren haben, wenn z. B. meist nur Transitionen der niedrigsten Prioritätsstufe schaltbar werden. In diesem Fall fällt die Effizienz des Verfahrens auf die des Referenzmodells zurück ($eff^{+}_{trans} = 1/n$), da auch die erfolgreiche Auswahl meist (fast) alle Transitionen testen muss.

Entsprechend kann man hier bereits als *Spezifikationsstilregel* festhalten, dass solche Szenarien vermieden, also die am häufigsten schaltbaren Transitionen nur von möglichst wenigen anderen Transitionen bzgl. ihrer Priorität übertroffen werden sollten.

Die Implementierung des Verfahrens[111] erfolgt in der Klasse `LTS_Prio`, die eine Spezialisierung von `LocalTransSelector` ist. Sie enthält zwei Listen mit Transitionsreferenzen, in denen die `DELAY`-*Transitionen* bzw. die *übrigen Transitionen* enthalten sind. Letztere sind dabei nach Prioritäten absteigend sortiert. Die Auswahl erfolgt in der ersten Liste immer vollständig und in der zweiten (prioritätssortierten) Liste bis zum ersten erfolgreichen Test.[112] Somit wird unter den o. g. Randbedingungen immer eine Minimalzahl von Transitionen getestet und im Erfolgsfall eine Transition mit maximaler Priorität zurückgeliefert.

4.3.5 Kontrollzustandstabellen

Ein Verfahren zur *Verkleinerung des Suchraums* ergibt sich aus der modulweiten Auswertung des jeweiligen Kontrollzustandes. Dazu wird für jeden möglichen Kontrollzustand eine (schwach aggregierende) Liste derjenigen Transitionsinstanzen erstellt, deren `FROM`-Klausel diesen Kontrollzustand referenziert. Bei der Transitionsauswahl kann dann ausgehend von einem konkreten Kontrollzustand die betroffene Liste (typischerweise durch einen Indexzugriff in eine entsprechende *Kontrollzustandstabelle*) ermittelt werden und die weitere Auswahl dann beschränkt auf die enthaltenen Transitionsinstanzen ausgeführt werden (siehe Abb. 4-21).

111. Die Aktivierung des prioritätsgesteuerten Auswahlverfahrens erfolgt durch den Kommandozeilenparameter "`-localselect prio`" der generierten Implementierung.

112. Wurde aus der ersten Liste bereits eine Transition ausgewählt, so bricht die Suche in der zweiten Liste beim Erreichen der selben Prioritätsstufe ab, da nachfolgende Transitionen geringere Priorität als die schon gefundene (`DELAY`-) Transition haben und daher ohnehin nicht mehr ausgewählt werden können.

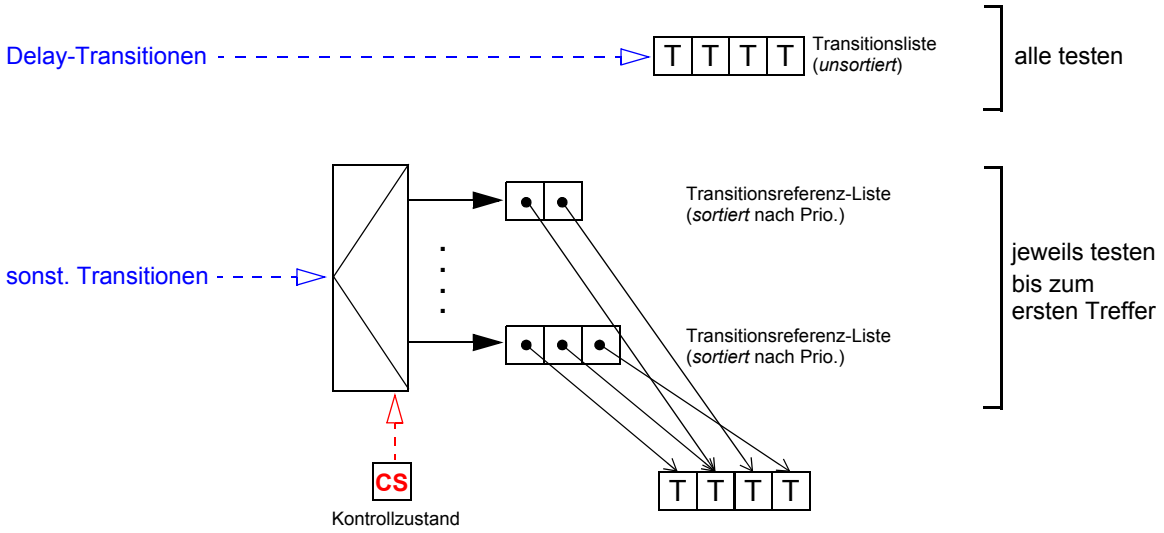


Abbildung 4-21: Struktur der kontrollzustandsbasierten Auswahl

Die Implementierung des Verfahrens[113] erfolgt in der Klasse `LTS_CST` („*control state table*“), die eine Spezialisierung der im letzten Abschnitt vorgestellten Klasse `LTS_Prio` (und damit indirekt auch von `LocalTransSelector`) ist. Dadurch kann sie auf die bereits etablierte und auch hier erforderliche Trennung zwischen `DELAY`-*Transitionen* und den *übrigen Transitionen* aufbauen. Letztere werden über ein dynamisch anhand der Anzahl der Kontrollzustände erzeugtes Array von Transitionslisten (`XPtrList<SimpleTrans>`, also schwache Aggregation[114]) nach dem oben beschriebenen Muster gemäß ihrer `FROM`-Klauseln referenziert. Die einzelnen Listen werden dabei im Sinne der prioritätsgesteuerten Auswahl angelegt und entsprechend optimiert ausgewertet (siehe Abschnitt 4.3.4).

Durch diese Kombination der beiden Optimierungsverfahren ist dieses Auswahlverfahren dem rein prioritätsgesteuerten in praktisch allen Fällen vorzuziehen. Der geringfügig höhere Aufwand bei der Initialisierung und der Aufwand beim Indexzugriff auf die Tabelle während der Auswahl sind unter normalen Umständen vernachlässigbar gering.

Der zu erwartende *Gewinn* durch das Verfahren hängt letztlich sehr stark von der *Selektivität* des Kontrollzustandes ab, also davon, wie viele Transitionen jeweils einem konkreten Kontrollzustand angehören. Transitionen ohne `FROM`-Klausel gehören dabei allen Kontrollzuständen an. Entsprechend sollten zur Steigerung der Wirksamkeit möglichst viele Transitionen eine `FROM`-Klausel besitzen und dort möglichst wenige Zustände referenzieren. Aber auch die Verteilung der Transitionen auf die Kontrollzustände und die Häufigkeit eines Kontrollzustandes bei den Transitionsauswahlen ist relevant: In den (bei den Auswahlen) am häufigsten vorliegenden Kontrollzuständen sollten möglichst wenige `FROM`-Klauseln erfüllt sein, damit auch nur wenige Transitionen in Betracht gezogen werden müssen.

Betrachten wir als Beispiel ein Modul mit 10 Transitionen, die jeweils 2 von ebenfalls 10 Kontrollzuständen gleichverteilt in ihren `FROM`-Klauseln referenzieren. Im Mittel müssen dann also pro Auswahl auch 2 Transitionen betrachtet werden, zusammen mit der prioritätsgesteuerten Auswahl ergibt sich also eine zu erwartende Auswahleffizienz eff^{+}_{trans} von 66,6 %. Fügt man

113. Die Aktivierung des prioritätsgesteuerten Auswahlverfahrens erfolgt durch den Kommandozeilenparameter “`-localselect CST`“ der generierten Implementierung.

114. Wenn eine `FROM`-Klausel mehrere Zustände referenziert, ist sie auch in mehreren Listen enthalten.

diesem System 10 weitere Transitionen hinzu, die jedoch *keine* FROM-Klauseln haben, so steigt die Anzahl der pro Auswahl im Mittel zu betrachtenden Transitionen auf 12 an, die zu erwartende Auswahleffizienz eff^{+}_{trans} sinkt auf 15,4 %[115] und verschlechtert sich somit um dem Faktor 4,3.

Offensichtlich sind gerade Transitionen ohne FROM-Klauseln besonders kritisch bzgl. der Wirksamkeit des Verfahrens. Eine typische Ursache für solche Transitionen in Protokollspezifikationen ist die starke Bezugnahme auf andere Zustandsraumkomponenten (speziell Variablen) über PROVIDED-Klauseln. Dies ist besonders dann kritisch, wenn sie zur Modellierung von Teilautomaten innerhalb eines Moduls eingesetzt werden. Dazu werden meist Aufzählungstyp-Variablen[116] oder boolesche Variablen als Ersatz für einen Kontrollzustand modelliert und die entsprechenden FROM- und TO-Klauseln durch PROVIDED-Klauseln[117] bzw. Wertzuweisungen im Transitionsblock[118] ersetzt. Auch findet man häufig Situationen vor, in denen komplexere (aber ebenfalls variablenbezogene) Bedingungen in PROVIDED-Klauseln eine FROM-Klausel ersetzen. Ein geradezu klassisches Beispiel ist eine PROVIDED-Klausel, die die Schaltbarkeit einer Empfangs- oder Sendetransition vom Zustand[119] einer als Array modellierten Warteschlange abhängig macht, wie man sie insbesondere im Sliding-Window- und dem XTP-4.0-Benchmark vorfindet.

4.3.6 Messagequeue-basierte Auswahl

Die primäre Zielsetzung bei der Spezifikation von Kommunikationssystemen ist der Austausch (asynchroner) Nachrichten und ihre Verarbeitung in Protokollmaschinen. Entsprechend enthalten Estelle-Spezifikationen aus dieser Anwendungsdomäne häufig eine erhebliche Anzahl von *Eingabetransitionen*, also Transitionen mit WHEN-Klauseln, durch die über (externe oder interne) Interaktionspunkte Nachrichten empfangen werden können. Dazu referenziert jede WHEN-Klausel einen *Interaktionspunkt*[120] und einen *Nachrichtentyp*, der über diesen Interaktionspunkt empfangen werden kann (siehe auch Abschnitt 3.3.7.6). Dabei können auch verschiedene Transitionsinstanzen um die selbe Interaktion konkurrieren.

Die einzelnen Interaktionspunkte speichern eingehende Nachrichten in einer zugeordneten FIFO-Queue zwischen, bis sie von einer Eingabetransition empfangen werden können. Dabei können sie jeweils eine eigene Queue besitzen („`individual queue`“), oder eine zentrale Queue der Modulinstanz nutzen („`common queue`“). Im letzteren Fall bleibt jedoch jede Nachricht in der zentralen Queue ihrem ursprünglichen Interaktionspunkt zugeordnet und kann nur über diesen empfangen werden.

---

115. Wir gehen davon aus, dass (nicht nur aufgrund der FROM-Klausel) jeweils eine Transition in dem Modul schaltbar ist und somit im Mittel 6,5 Transitionen getestet werden müssen.
116. z. B.: `subState: (sending, receiving, waiting)`
117. z. B.: `PROVIDED (subState = sending)`
118. z. B.: `subState := waiting`
119. bzgl. der Prädikate „*ist voll*“ (also kein Empfang möglich) oder „*ist leer*“ (also kein Senden möglich)
120. Es können auch über Interaktionspunktarrays wechselnde Interaktionspunkte referenziert werden, (siehe Abschnitt 3.3.7.6).

Die Zuordnung eines Interaktionspunktes zu einer zentralen oder individuellen Queue, die dabei im Modul auch gemischt erfolgen kann, entscheidet darüber, ob eingehende Interaktionen nur in der Reihenfolge ihres Eintreffens oder in beliebiger Reihenfolge verarbeitet werden können.

Offensichtlich ergibt sich hier eine Möglichkeit zu einer *Verkleinerung des Suchraums*, indem die Menge der in Frage kommenden Eingabetransitionen bei der Transitionsauswahl auf diejenigen reduziert wird, die die jeweils vordersten Elemente in den einzelnen Warteschlangen (Messagequeues) überhaupt empfangen können. Dazu wird jedem Interaktionspunkt eine Liste derjenigen Transitionen zugeordnet, die (nur) aus diesem Interaktionspunkt eine konkrete Nachricht eines Typs empfangen können. Diese Liste wird zudem nach dem jeweiligen Nachrichtentyp in (disjunkte) Unterlisten aufgeteilt (siehe Abb. 4-22).

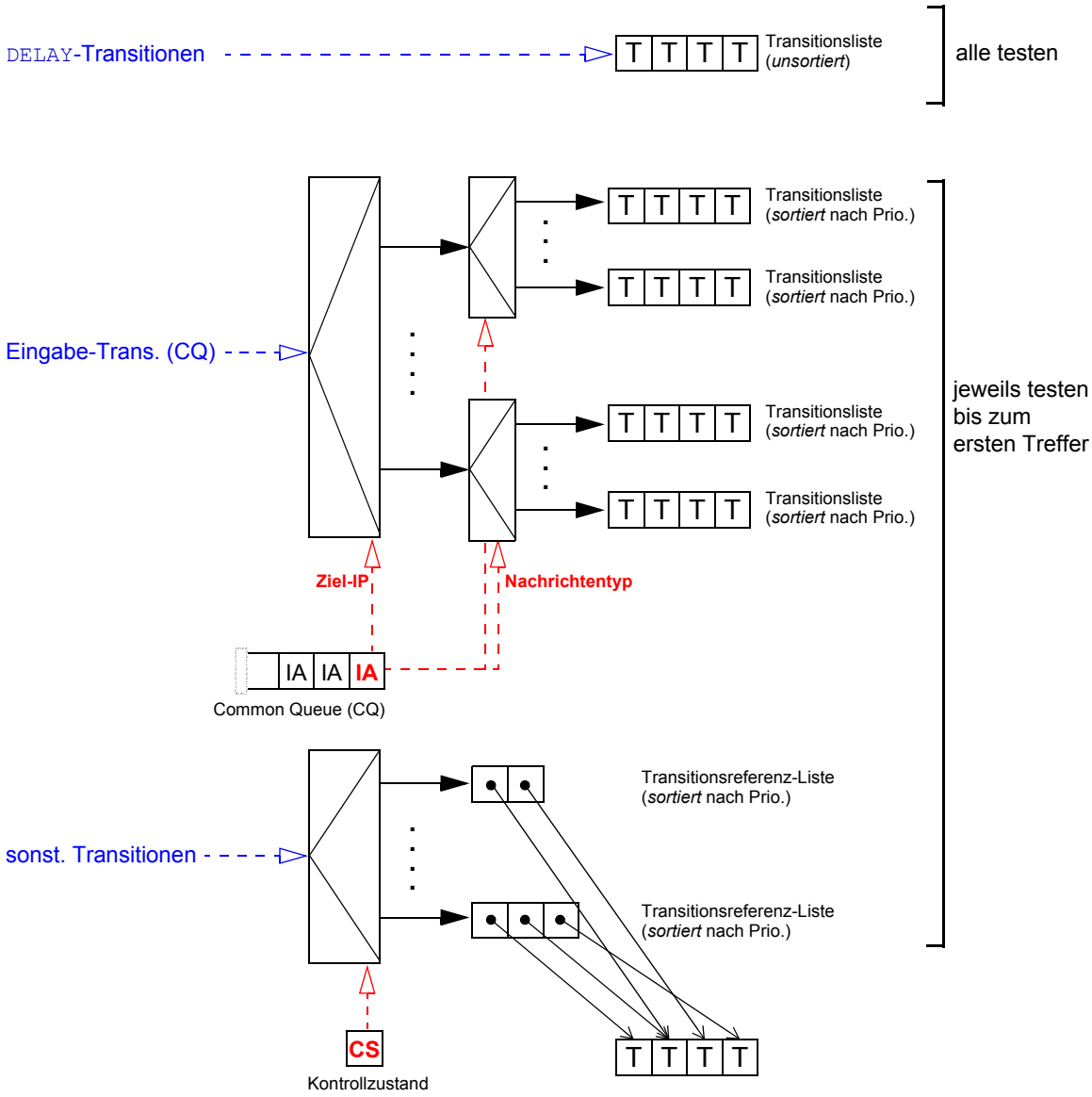


Abbildung 4-22: Struktur der Messagequeue-basierten Auswahl

Bei der modullokalen Auswahl werden nun die derart zugeordneten Eingabetransitionen nicht mehr über die bisher betrachteten Mechanismen geprüft, sondern es werden ausgehend von den Warteschlangen und den jeweils vordersten Interaktionen die o. g. Unterlisten der potentiellen Empfängertransitionen ermittelt, und zwar auf Basis des referenzierten Interaktionspunktes und des Nachrichtentyps. Die Auswahl erfolgt dann beschränkt auf diese Unterlisten.

Aus diesem Auswahlverfahren müssen jedoch diejenigen Transitionen ausgeschlossen werden, die aufgrund eines variablen Indexwertes für ein Interaktionspunkt-Array nicht eindeutig einem einzelnen Interaktionspunkt zugeordnet werden können (siehe Abschnitt 3.3.4 und Abschnitt 3.4.4).[121] In der Implementierung von XEC schränken wir die Auswertung zudem auf die *zentrale Warteschlange* der Modulinstanz ein, da ansonsten durch die Vielzahl der zu testenden Warteschlangen ein ungünstiges Kosten-/Nutzen-Verhältnis entsteht.[122] Entsprechend werden Transitionen, die sich auf eine individuelle Warteschlange beziehen, ebenfalls ausgeschlossen und in der Liste der nicht-`DELAY`-Transitionen behandelt (siehe „*sonstige*“ in Abb. 4-22).

Insgesamt erfolgt hier wie auch schon in vorherigen Verfahren eine getrennte Auswahl gemäß den nunmehr drei Auswahlverfahren und es wird aus den jeweiligen Ergebnissen eine Transition mit maximaler Priorität ausgewählt.[123]

Die Implementierung des Verfahrens[124] erfolgt in der Klasse `LTS_QST` („*queue state table*“), die eine Spezialisierung der im letzten Abschnitt vorgestellten Klasse `LTS_CST` (und damit indirekt auch von `LocalTransSelector`) ist. Dadurch kann sie auf die bereits etablierte und auch hier erforderliche Trennung zwischen `DELAY`-*Transitionen* und den *sonstigen Transitionen* aufbauen. Aus der letzteren Liste werden die in Frage kommenden[125] `DELAY`-Transitionen aussortiert und in einer mehrstufigen Struktur analog zu Abb. 4-22 jeweils in eine Transitionsliste übertragen.[126] Die einzelnen Listen werden dabei im Sinne der prioritätsgesteuerten Auswahl angelegt und entsprechend optimiert ausgewertet (siehe Abschnitt 4.3.4).

Da bei der kontrollzustandsorientierten Auswahl die Transitionen jeweils einer Transitionslisten um die selben Nachrichten (selber Typ am selben Interaktionspunkt) konkurrieren, enthalten diese Listen meist nur eine sehr geringe Anzahl von Transitionen, meist sogar nur genau eine. Entsprechend wurde hier auf das Nachschalten einer kontrollzustandsgesteuerten Auswahl (siehe Abschnitt 4.3.5) verzichtet. Hier wäre ggf. eine Erweiterung zur dynamischen Anpassung des Auswahlverfahrens an die konkreten Gegebenheiten der Spezifikation bzw. bzgl. der konkreten Menge von Transitionen in den einzelnen Listen denkbar.

Die Effizienz des Verfahrens hängt entsprechend davon ab, wie viele Transitionen über das Verfahren abgewickelt werden können und wie viele davon jeweils um eine Interaktion konkurrieren. Enthält ein Modul nur Eingabetransitionen und sind diese für jeweils einen Interaktionspunkt allein verantwortlich, so kann die Menge der in Frage kommenden Transitionen auf

121. In keiner der bisher von uns ausgewerteten „realen“ Spezifikationen ist dieser Fall jedoch aufgetreten.

122. Die Optimierungsmethode basiert auf einer Datenstruktur, deren Aufbau und Auswertung eine nicht unerhebliche Komplexität hat (siehe Abb. 4-22). Da sich auf eine individuelle Queue (also auf einen Interaktionspunkt) meist nur einige wenige Eingabetransitionen beziehen, übersteigt der mit der Auswertung der Datenstruktur verbundene Aufwand schnell den einer direkten Suche. Die Spezifikation als *gemeinsame* Warteschlangen bietet hier die weitaus effektivere Lösung. Wir verzichten daher vorläufig auf diese Optimierung, sie kann jedoch leicht dem Laufzeitsystem hinzugefügt werden.

123. Auch hier wird nach dem vollständigen Test in der `DELAY`-Transitionsliste die Auswahl in den beiden folgenden Listen optimiert, indem die Suche entlang der prioritätssortierten Listen bei der Prioritätsstufe möglicher Zwischenergebnisse aus vorherigen Teilauswahlen abgebrochen wird.

124. Die Aktivierung des prioritätsgesteuerten Auswahlverfahrens erfolgt durch den Kommandozeilenparameter “`-localselect QST`“ der generierten Implementierung.

125. Also Transitionen, bei denen der Hint `HINT_FIXED_IP` gesetzt ist (siehe Abschnitt 3.4.4).

126. Im Unterschied zur Kontrollzustandstabelle kann hier jede Transition nur in einer Liste enthalten sein.

eine reduziert werden und es ist sowohl für eine erfolgreiche wie auch für eine erfolglose Auswahl nur ein Transitionstest erforderlich. Dieses Szenario ist für reaktive Kommunikationsprotokolle nicht untypisch und stellt insbesondere eine der Voraussetzungen für die Anwendung des *Activity-Thread-Modells* dar ([Svo89], [HeMiKö97], siehe auch Abschnitt 4.2.1.2).

4.3.7 Evaluierung der Optimierungsverfahren

Wir evaluieren nun die verschiedenen modullokalen Transitionsauswahlverfahren zusammen mit den bereits in Abschnitt 4.2 entwickelten globalen Modulauswahlverfahren anhand der in Abschnitt 2.3 vorgestellten Benchmarkspezifikationen.

4.3.7.1 Auswertung des Ping-Pong-Benchmarks

Die Anwendung des modullokalen Transitionsauswahlverfahrens auf den Ping-Pong-Benchmark unter Anwendung der maximalen globalen Optimierung (vgl. Zeile „*independent Modules und ereignisgesteuert*“ in Tabelle 4.7 auf Seite 141) ergibt eine Verbesserung der *(globalen) Transitionsauswahleffizienz* Eff_{trans} von 49,7 % bei der Referenzauswahlmethode und auf 66,1 % bei der prioritätsgesteuerten Auswahlmethode (siehe Tabelle 4.16). Dies entspricht recht gut dem Erwartungswert von 66,6 %[127].

Tabelle 4.16: Lokale Auswahleffizienz Ping-Pong-Benchmark (ACTIVITY, mit max. globaler Optimierung)

Transitions-Auswahlverfahren	Module		Transitionen (global)			mittlere lokale T.-Eff.
	getestet	Eff.[a]	getestet	gefeuert	Eff.	
Referenz[b]	3.822	99,4 %	7.644	3.800	**49,7 %**	50,0 %
prioritätsgesteuert[c]	3.822	99,4 %	5.744	3.800	**66,1 %**	66,6 %
kontrollzustandsgesteuert[d]	3.822	99,4 %	5.744	3.800	**66,1 %**	66,5 %
Messagequeue-basiert[e]	3.822	99,4 %	3.802	3.800	**99,9 %**	100,5 %

a. Die Anzahl der gefeuerten Module ist identisch zur Anzahl der gefeuerten Transitionen
b. Laufzeit-Optionen: -tsim -treesleep -imodopt -eventdrive 3 -*localselect* straight
c. Laufzeit-Optionen: -tsim -treesleep -imodopt -eventdrive 3 -*localselect* prio
d. Laufzeit-Optionen: -tsim -treesleep -imodopt -eventdrive 3 -*localselect* CST
e. Laufzeit-Optionen: -tsim -treesleep -imodopt -eventdrive 3 -*localselect* QST

Die zusätzliche kontrollzustandsgesteuerte Auswahl kann das Ergebnis nicht verbessern, da die Spezifikation keine Kontrollzustände einsetzt. Erst die Messagequeue-basierte Auswahl liefert mit 99,9 % ein praktisch optimales Ergebnis. Hintergrund ist, dass die `WHEN`-Klauseln im Ping-Pong-Benchmark sehr stark selektiv wirken (zu einem Zeitpunkt ist genau eine solche Klausel erfüllt).

127. Es existieren zwei prioritätsgleiche Transitionen pro Modul, also werden im Mittel 1,5 Transitionen getestet.

Berechnet man die *mittlere modullokale Transitionsauswahleffizienz* eff_{trans}, also den Quotienten aus den (global) getesteten Modulen (nm_{tested}) und den dabei jeweils getesteten Transitionen (nt_{tested}), so stellt man fest, dass bei der Messagequeue-basierten Auswahl ein Wert von über 100 % erreicht wird, also nicht bei jedem Modultest überhaupt auch nur eine Transition getestet wurde. Dies ist eine Folge der hier optimalen Effizienz des lokalen Auswahlverfahrens, durch das bei einer erfolglosen Auswahl[128] alle Transitionen bereits aufgrund der leeren Nachrichtenwarteschlange ausgeschlossen werden.

Offensichtlich erreichen wir durch diese Art der teilweisen Zentralisierung der modullokalen Transitionsauswahl die begrifflichen Grenzen, an denen der Test einer einzelnen Transition als zählbares Ereignis noch existiert. Dies wird noch deutlicher, wenn man den selben Versuch ohne globale Auswahloptimierung durchführt (siehe Tabelle 4.17, vgl. Zeile „*Basis*“ in Tabelle 4.7 auf Seite 141).

Tabelle 4.17: Lokale Auswahleffizienz Ping-Pong-Benchmark (ACTIVITY, ohne globale Optimierung)

Transitions-Auswahlverfahren	Module		Transitionen (global)			mittlere lokale T.-Eff.
	getestet	Eff.[a]	getestet	gefeuert	Eff.	
Referenz[b]	41.714	9,1 %	75.826	3.800	**5,0 %**	55,0 %
prioritätsgesteuert[c]	41.714	9,1 %	73.926	3.800	**5,1 %**	56,4 %
kontrollzustandsgesteuert[d]	41.714	9,1 %	73.926	3.800	**5,1 %**	56,4 %
Messagequeue-basiert[e]	41.714	9,1 %	7.395	3.800	**51,4 %**	4058 %

a. Die Anzahl der gefeuerten Module ist identisch zur Anzahl der gefeuerten Transitionen
b. Laufzeit-Optionen: -tsim -notreesleep -noimodopt -eventdrive 0 -*localselect* straight
c. Laufzeit-Optionen: -tsim -notreesleep -noimodopt -eventdrive 0 -*localselect* prio
d. Laufzeit-Optionen: -tsim -notreesleep -noimodopt -eventdrive 0 -*localselect* CST
e. Laufzeit-Optionen: -tsim -notreesleep -noimodopt -eventdrive 0 -*localselect* QST

Dabei fällt die *(globale) Modulauswahleffizienz* Eff_{mod} auf 9,1 %. Dies senkt natürlich insgesamt die *globale Transitionsauswahleffizienz* Eff_{trans} auf 5,0 % bzw. 5,1 %. Aus diesen Werten ergibt sich als Quotient eine *mittlere modullokale Transitionsauswahleffizienz* eff_{trans} von 55 % für die (lokale) Referenzauswahlmethode,[129] aber nur 56 % für die prioritätsgesteuerte Auswahlmethode. Der Unterschied zeigt nochmals, dass die Wirksamkeit von Optimierungsmaßnahmen auf Basis der *Transitionstest-Reihenfolge* in ihrer Wirksamkeit von der Effizienz der globalen Auswahl abhängt (siehe Abschnitt 4.3.2).

Bei der Messagequeue-basierten Auswahlmethode erreichen wir nun eine *globale Transitionsauswahleffizienz* von 51,4 % und dadurch ein Verhältnis von über 40:1 (4058 %) der getesteten Module zu den insgesamt getesteten Transitionen (s. o.). Offensichtlich kompensiert hier die

128. Hier treten ohnehin allein durch die Suche nach einem ersten schaltbaren Modul bzw. bei der Deadlockerkennung erfolglose Modultests auf.

129. Die Verbesserung der mittleren modullokalen Transitionsauswahleffizienz für die Referenzauswahlmethode von 50 % auf 55 % beruht auf den hinzugekommenen modullokalen Auswahlen in Modulinstanzen ohne Transitionen.

sehr wirksame Messagequeue-basierte (lokale) Auswahlmethode die Schwächen der globalen Auswahlmethode zu einem erheblichen Teil. Welche Auswirkungen dies auf die Ausführungszeiten hat, werden wir später in Abschnitt 4.4 untersuchen.

4.3.7.2 Auswertung des Sliding-Window-Benchmarks

Betrachten wir nun den bzgl. der modullokalen Transitionsstruktur deutlich komplexeren *Sliding-Window-Benchmark*. Wir evaluieren die modullokale Transitionsauswahl mit den für Tabelle 4.8 auf Seite 148 bereits eingesetzten Parametern und in Kombination mit der maximalen globalen Optimierung (siehe Tabelle 4.18).

Tabelle 4.18: Lokale Auswahleffizienz SW-Benchmark (ACTIVITY, mit max. globaler Optimierung)

Transitions-Auswahlverfahren	Module		Transitionen (global)			mittlere lokale T.-Eff.
	getestet	Eff.[a]	getestet	gefeuert	Eff.	
Referenz[b]	1.535	98,4 %	8.042	1.510	**18,8 %**	19,1 %
prioritätsgesteuert[c]	1.535	98,4 %	6.242	1.510	**24,2 %**	24,6 %
kontrollzustandsgesteuert[d]	1.535	98,4 %	6.242	1.510	**24,2 %**	24,6 %
Messagequeue-basiert[e]	1.535	98,4 %	5.577	1.510	**27,1 %**	27,5 %

a. Die Anzahl der gefeuerten Module ist identisch zur Anzahl der gefeuerten Transitionen
b. Laufzeit-Optionen: -tsim -treesleep -imodopt -eventdrive 3 -*localselect* straight
c. Laufzeit-Optionen: -tsim -treesleep -imodopt -eventdrive 3 -*localselect* prio
d. Laufzeit-Optionen: -tsim -treesleep -imodopt -eventdrive 3 -*localselect* CST
e. Laufzeit-Optionen: -tsim -treesleep -imodopt -eventdrive 3 -*localselect* QST

Hier ergeben sich deutlich schlechtere mittlere lokale Transitionsauswahleffizienzen von 19,1 % bis 27,5 %, die sich aufgrund der fast optimalen globalen Modulauswahleffizienz in nahezu identischen Werten für die globale Transitionsauswahleffizienz niederschlagen.

Eine genauere Analyse der vom XEC-Laufzeitsystem gelieferten Statistikdaten zeigt, dass sich unter den getesteten Transitionen in allen Fällen ein konstanter Anteil von *4060 Delay-Transitionstests* (also jeweils mehr als die Hälfte aller getesteter Transitionen) befindet. Dies deckt sich mit den in Abschnitt 4.3.3 bereits vorhergesagten Effekten der Estelle-Semanik auf die Optimierbarkeit von Transitionstests bei `DELAY`-Transitionen.

So enthalten die Sliding-Window-Protokollmaschinen in der Benchmarkspezifikation jeweils ein spezielles Kindmodul, in dem alle `DELAY`-Transitionen angeordnet sind (siehe Abb. 4-14 auf Seite 147 bzw. Zeile 268-308 in Anhang B.4). Aufgrund der selektiven Wiederholungsstrategie muss die Anzahl der Delay-Timer der Sliding-Window-Fenstergröße entsprechen. Daraus ergeben sich bei einer Sliding-Window-Fenstergröße von 20 und zwei Protokollmaschineninstanzen insgesamt 40 `DELAY`-Transitionsinstanzen. Nun wird jede Aktivierung und Deaktivierung eines Timers durch jeweils eine Transition vom Vatermodul angezeigt. Dies ergibt bei insgesamt 100 Datenübertragungen insgesamt 200 Interaktionen an eine der Kindmodulinstanzen. Der kritische Punkt besteht an dieser Stelle nun darin, dass jede dieser 200 Interaktionen zu ei-

ner lokalen Auswahl in der Kindmodulinstanz führt und dabei jeweils alle 20 DELAY-Transitionsinstanzen getestet werden müssen (siehe Abschnitt 4.3.3). In der Summe ergeben sich somit bereits 4000 DELAY-Transitionstests.

Bereinigt man die Statistik[130] um diese DELAY-Transitionen, so ergeben sich die in Tabelle 4.19 dargestellten Werte.

Tabelle 4.19: Lokale Auswahleffizienz SW-Benchmark (ACTIVITY, mit max. globaler Optimierung) ohne DELAY-Trans.

Transitions-Auswahlverfahren	Module		Transitionen (global)			mittlere lokale T.-Eff.
	getestet	Eff.[a]	getestet	gefeuert	Eff.	
Referenz[b]	1.535	98,4 %	3.982	1.510	**37,9 %**	38,5 %
prioritätsgesteuert[c]	1.535	98,4 %	2.182	1.510	**69,2 %**	70,3 %
kontrollzustandsgesteuert[d]	1.535	98,4 %	2.182	1.510	**69,2 %**	70,3 %
Messagequeue-basiert[e]	1.535	98,4 %	1.517	1.510	**99,5 %**	101,2 %

a. Die Anzahl der gefeuerten Module ist identisch zur Anzahl der gefeuerten Transitionen
b. Laufzeit-Optionen: -tsim -treesleep -imodopt -eventdrive 3 -*localselect* straight
c. Laufzeit-Optionen: -tsim -treesleep -imodopt -eventdrive 3 -*localselect* prio
d. Laufzeit-Optionen: -tsim -treesleep -imodopt -eventdrive 3 -*localselect* CST
e. Laufzeit-Optionen: -tsim -treesleep -imodopt -eventdrive 3 -*localselect* QST

Offensichtlich war speziell die Messagequeue-basierte Auswahloptimierung ohne Berücksichtigung der DELAY-Transitionen nahezu optimal und ergänzt somit die ebenfalls nahezu optimale globale Auswahl sehr gut.

DELAY-Transitionen stellen also offenbar ein durchaus relevantes Problem bei der effizienten modullokalen Auswahl dar, auch wenn die im Sliding-Window-Benchmark vorzufindende Situation in dieser Hinsicht schon außergewöhnlich erscheint. Wir werden in Abschnitt 4.3.8 mögliche Lösungen dieser Problematik diskutieren.

4.3.7.3 Auswertung des XTP-Benchmarks

Wir vervollständigen nun unsere Messreihe mit der Auswertung des XTP-4.0-Applikationsbenchmarks (siehe Abschnitt 2.3.4). Zusammen mit den erforderlichen Benutzer- und Netzwerkmodulen enthält die Benchmarkspezifikation insgesamt 219 Transitionsdefinitionen und gehört damit zu den komplexesten bekannten Estelle-Spezifikationen.

Bereits die Instanziierung des minimalen Testszenarios mit zwei Benutzermodulinstanzen, zwei Protokollmaschineninstanzen und einer Netzwerkmodulinstanz (siehe Abb. 2-6 auf Seite 20) führt zu einer Gesamtzahl von 620 Transitions- in 24 Modulinstanzen, im Durchschnitt besitzt also jede Modulinstanz knapp 26 Transitionsinstanzen. Offensichtlich bietet dieser Benchmark eine erhebliche Komplexität bzgl. der lokalen wie auch der globalen Auswahlverfahren.

Wir betreiben den Benchmark mit den folgenden Parametern:

130. Wir haben hier nur die Transitionstests bearbeitet. Die Modulauswahlen wurden nicht verändert.

- Nutzdatenmenge maximal [Bytes]: 10
- Nutzdatenmenge verwendet [Bytes]: 10
- Datenpakete pro Testphase: 1000
- Synchronisationspunkte (Anzahl Pakete): 3
- Anzahl Testphasen: 3
- Verhalten des Mediums: unverzögert und verlustfrei

Daraus ergibt sich nach der Verbindungsaufbauphase eine abwechselnde Übertragung eines Paketbursts vom *initiierenden Nutzermodul* zum empfangenden und von dort zurück. Nach jeweils drei Paketen wartet der Empfänger zur Synchronisation auf die ausstehenden Antwortpakete, bevor er weiter sendet. Insgesamt werden so 1000 Pakete verarbeitet und dann die Verbindung geschlossen. Dieser *Testzyklus* wird insgesamt dreimal ausgeführt.

In dieser Konfiguration werden[131] zum Transport der insgesamt 6000 Sendeaufträge eines Benutzermoduls aufgrund der hohen Komplexität der XTP-Protokollmaschine (siehe Abschnitt 2.3.4) insgesamt 114.190 Interaktionen erzeugt, wobei sich zu einem Zeitpunkt jedoch nur maximal 8 gleichzeitig in Übertragung befinden. Es wird 19.130 mal ein Delay-Timer (innerhalb der XTP-Protokollmaschinen) gestartet und (im Gegensatz zu den vorherigen Benchmarks) läuft auch 9.030 mal ein Timer ab. Das Ablaufen der Timer ist zur internen Koordination der XTP-Protokollmaschinen erforderlich, da intern eine Zwischenpufferung zur Paketzusammenführung erfolgt. Das Ablaufen der Timer ist dabei immer auch mit einer temporären Inaktivität des Gesamtsystems (`InternalClock::sleepUntil(...)`), siehe Abschnitt 3.5.2) verbunden. Insgesamt werden 149.305 Transitionen ausgeführt.

Tabelle 4.20: Lokale und globale Auswahleffizienz XTP-Benchmark (ACTIVITY)

Optimierung der Auswahlverfahren	Module		Transitionen (global)			mittlere lokale T.-Eff.
	getestet	Eff.[a]	getestet	gefeuert	Eff.	
keine[b]	2.314.469	**6,5 %**	55.223.831	149.305	**0,3 %**	**4,2 %**
nur global[c]	553.075	**27,0 %**	16.213.323	149.293	**0,9 %**	**3,4 %**
nur lokal[d]	2.306.489	**6,5 %**	9.220.082	149.305	**1,6 %**	**25,0 %**
global und lokal[e]	550.093	**27,1 %**	2.427.581	149.296	**6,1 %**	**22,7 %**

a. Die Anzahl der gefeuerten Module ist identisch zur Anzahl der gefeuerten Transitionen
b. Laufzeit-Optionen: -tsim -**no**treesleep -**no**imodopt -eventdrive **0** -localselect **straight**
c. Laufzeit-Optionen: -tsim -treesleep -imodopt -eventdrive **3** -localselect **QST**
d. Laufzeit-Optionen: -tsim -**no**treesleep -**no**imodopt -eventdrive **0** -localselect **straight**
e. Laufzeit-Optionen: -tsim -treesleep -imodopt -eventdrive **3** -localselect **QST**

In Tabelle 4.20 sind die Effizienzen der verschiedenen Kombinationen lokaler und globaler Auswahlverfahren zusammengestellt. Wir verzichten dabei auf die Differenzierung der einzelnen Zwischenschritte und vergleichen direkt die einfachsten und fortgeschrittensten Auswahlverfahren.

131. Die Zahlenangaben beziehen sich auf die Referenzimplementierungsmethoden. Zusammen mit den optimierten Auswahlverfahren können sich aufgrund der variierenden Auswahlergebnisse und der damit verbundenen Verhaltensänderungen geringfügige Abweichungen ergeben.

Die Optimierung der globalen Auswahl steigert deren Effizienz von 6,5 % auf 27 %, also um den Faktor 4,1. Dieser Gewinn ist vor dem Hintergrund der Komplexität des Systems und der durch das XTP-Protokoll und die o. g. Timer bedingten schwächeren Ausprägung der Activity-Threads durchaus erwartungsgemäß.

Die mittlere lokale Transitionsauswahleffizienz wird durch ihre Optimierung von 4,2 % auf 25,0 % fast um den Faktor 6 verbessert. Die Ursache für die Begrenztheit der durch die lokale Optimierung erreichbaren Effizienz ergibt sich durch die große Zahl von Transitionen, deren Schaltbereitschaft vollständig oder überwiegend durch `PROVIDED`-Klauseln gesteuert wird. Ihre Auswahl wird bisher nur im Zusammenhang mit der *Transitionstest-Reihenfolge* optimiert (siehe Abschnitt 4.3.2). Die `DELAY`-Transitionen haben hier dagegen mit ca. 33 % (bei maximaler Optimierung) zwar eine merkliche, aber keine dominierende Wirkung wie beim Sliding-Window-Benchmark.

In der Kombination steigern die lokalen und globalen Optimierungsverfahren die globale Transitionsauswahleffizienz von 0,3 % auf 6,1 % um den Faktor 22,7. Dies bedeutet, dass global statt 370 jetzt nur noch durchschnittlich 16 Transitionen getestet werden müssen, um eine schaltbare Transition auszuwählen.

Abschließend untersuchen wir noch die Auswirkungen des Einsatzes der in Abschnitt 4.2.2.4 eingeführten *Independent-Module-Erweiterung*. Die globale Auswahloptimierung konnte in der Spezifikation bisher bereits 5 unabhängige Modulinstanzen[132] identifizieren. Attributieren wir alle `ACTIVITY`-attributierten Module entsprechend als „`INDEPENDENT ACTIVITY`“, so vergrößert sich die Anzahl auf 18, d. h. alle (potentiell) aktiven Modulinstanzen sind unabhängig. Die hinzugekommenen Modulinstanzen liegen dabei alle in den XTP-Protokollmaschinen.

Tabelle 4.21: Lokale und globale Auswahleffizienz XTP-Benchmark (INDEPENDENT ACTIVITY)

Optimierung der Auswahlverfahren	Module		Transitionen (global)			mittlere lokale T.-Eff.
	getestet	Eff.	getestet	gefeuert	Eff.	
keine	2.314.469	**6,5 %**	55.223.831	149.305	**0,3 %**	4,2%
nur global	459.980	**32,5 %**	12.734.931	149.295	**1,2 %**	3,6%
nur lokal	2.306.489	**6,5 %**	9.220.082	149.305	**1,6 %**	25,0%
global und lokal	460.978	**32,4 %**	2.383.190	149.298	**6,3 %**	19,3%

Wie aus Tabelle 4.21 zu ersehen ist, kann dadurch die globale Auswahleffizienz von 27,1 % auf 32,4 % gesteigert werden, woraus sich eine nur geringfügige Verbesserung der globalen Transitionsauswahleffizienz von 6,1 % auf 6,3 % ergibt. Eine der Ursachen ist, dass der Verbesserung der globalen Auswahleffizienz interessanterweise eine Verschlechterung der mittleren lokalen Transitionsauswahleffizienz von 22,7 % auf 19,3 % entgegenwirkt. Hier wurden offenbar Modultests eingespart, die früher ohnehin lokal überdurchschnittlich effizient realisiert werden konnten und damit den Durchschnitt verbesserten.[133]

132. Dies sind die jeweils zwei Benutzermodul- und Protokollmaschineninstanzen sowie die Netzwerkmodulinstanz (siehe Abb. 2-6 auf Seite 20). Das Systemmodul selbst hat keine Transitionen und wurde entsprechend automatisch ausgeschlossen (siehe Abschnitt 4.2.2.3).

4.3.8 Optimierung der DELAY-Transitionsauswahl

Bei der modullokalen Auswahl haben sich DELAY-Transitionen als problematisch erwiesen, da aufgrund der Semantik der *impliziten* Starts und Stopps der zugehörigen Timer die Einsparung von Transitionstests bei DELAY-Transitionen kritisch und nur mit erheblichem Aufwand möglich ist. Entsprechend werden bei jeder lokalen Auswahl alle DELAY-Transitionen des Moduls getestet. Dies führt zu einer deutlichen Verschlechterung der Auswahleffizienz, wenn

1. sich die DELAY-Transitionen in einer Modulinstanz befinden, die häufig getestet wird und
2. es eine große Anzahl von DELAY-Transitionsinstanzen gibt.

Betrachten wir dazu nochmals den Sliding-Window-Benchmark. Die Tatsache, dass sich hier alle DELAY-Transitionen einer Protokollmaschine in einem separaten Kindmodul befinden (siehe Zeile 268-308 in Anhang B.4) lässt zunächst vermuten, dass hier gerade ein Modul gefunden wurde, in dem nur geringe sonstige Aktivität vorliegt, die obige Bedingung (1) also nicht erfüllt ist.

Diese Modulinstanz erhält jedoch eine sehr große Anzahl von Interaktionen, durch die die Delay-Timer gestartet und gestoppt werden sollen, so dass sich zusammen mit ihrer großen Anzahl der in Abschnitt 4.3.7.2 beschriebene Overhead ergibt.

Geht man davon aus, dass man weder die dort verwendete selektive Wiederholungsstrategie aufgeben, noch eine andere Fenstergröße (und damit eine geringere Anzahl von DELAY-Transitionsinstanzen) nutzen möchte, so sind die häufigen Starts und Stopps der Timer und deren Anzahl unvermeidlich.

Eine Lösung *auf Spezifikationsebene* besteht z. B. darin, jeder DELAY-Transitionsinstanz eine eigene Modulinstanz zu geben, in der sie zusammen mit den bekannten beiden Transitionen zum interaktionsgesteuerten Starten und Stoppen liegt. Der Unterschied zur Lösung in der Spezifikation des Sliding-Window-Benchmarks besteht hier letztlich nur darin, dass die Vervielfältigung nicht per ANY-Klauseln bzgl. der einzelnen Transitionen, sondern per mehrfacher Instanziierung und über ein Interaktionspunktarray erfolgt.

Diese Lösung ist bzgl. der vorgestellten Optimierungsverfahren recht effektiv, da die lokale Transitionsauswahl nur noch bei einem externen Ereignis stattfindet, also bei Start, dem Stopp oder dem Ablaufen des Timers. Es ergeben sich dabei 1707 Transitionstests, von denen sich nur noch 190 (statt der früheren 4060) auf eine DELAY-Transitionsinstanz beziehen. Die resultierende globale Transitionsauswahleffizienz steigt damit von 27,1 % auf 88,5 %.

Ein alternativer Ansatz besteht in einer *Estelle-Erweiterung*, die genau einen solchen, explizit (also per Nachrichtenkommunikation oder per zusätzlichen Statements) steuerbaren Timer als Sprachprimitiv anbietet.[134] Dadurch können direkt die in Tabelle 4.19 vorhergesagten Effizienzen von bis zu 99,5 % erreicht werden.

133. Dies betrifft speziell die nur wenig aktiven Kontextmodule der XTP-Protokollmaschinen (siehe auch Abb. 2-5 auf Seite 20).

134. Dies entspricht konzeptionell insbesondere weitgehend dem Timer-Ansatz von SDL [ITU94].

4.4. Ausführungszeitbezogene Bewertung

Die bisherigen Auswertungen von Optimierungsmodellen basierten auf *zählbaren Größen* wie der Anzahl der getesteten oder geschalteten Transitionen. Diese konnten wir sowohl *analytisch* und als auch *experimentell* ermitteln, ohne dabei Bezug auf spezifische Eigenschaften der Plattform (z. B. die Ausführungsdauer für einzelne Operationen) nehmen zu müssen[135] oder statistische Fehler bei den Messungen zu betrachten (siehe auch Abschnitt 2.3)

Wie wir jedoch schon in Abschnitt 2.2 gesehen haben, beziehen sich fast alle wesentlichen Optimierungsziele auf die *Ausführungsdauer* von Protokolloperationen. Entsprechend ist deren Evaluierung der eigentliche Maßstab aller Optimierungsmaßnahmen.

Dies gilt ganz besonders vor dem Hintergrund, dass mit zunehmend komplexeren Optimierungstechniken die Übertragbarkeit der zählbaren Größen auf die Ausführungsdauer der dabei ausgeführten Operationen abnimmt, da hierbei der mit komplexen Optimierungstechniken verbundene zusätzliche Overhead zu berücksichtigen ist. Ein typisches Beispiel sind die bereits eingeführten Optimierungen der lokalen Auswahl, durch die teilweise sogar ohne einen einzigen Transitionstest festgestellt werden kann, dass keine der Transitionen schaltbar ist (siehe auch Abschnitt 4.3.7.2). Diese Auswertung erfordert jedoch ebenfalls Zeit. Ein anderer bisher nicht berücksichtigter Störfaktor ist der unterschiedliche Zeitbedarf der gezählten Größen (wie z. B. dem Ausführen eines Transitionsblocks).

Entsprechend werten wir in diesem Abschnitt einige der bisherigen Experimente auch realzeitbezogen aus. Wir verwenden dabei zur Vermeidung von Rückwirkungen der *Effizienz* einer Implementierung auf ihr *Verhalten*[136] jedoch weiterhin die in XEC integrierte Zeitsimulation (Abschnitt 3.5.2.2) für das Estelle-Zeitmodell. Die Messungen erfolgen dagegen natürlich anhand der real verbrauchten Zeit.

Die folgenden Messwerte wurden auf der Evaluierungsplattform[137] mit dem Makefile-Template „`optstat`“ erzeugt, das die maximalen Basis-Optimierungen nutzt, dabei jedoch weiterhin die Ermittlungen von Statistikdaten durch XEC aktiviert. Bei dem besser optimierenden Template „`optimize`“ (siehe Abschnitt 3.1.2.3) stünden diese Werte nicht mehr zur Verfügung, und die Messungen müssten durch manuell in die Spezifikation integrierte Plattformzugriffe realisiert werden.

In Tabelle 4.23 sind für den XTP-Benchmark die Summen der Ausführungszeiten aller Auswahlen und aller Ausführungsphasen wiedergegeben.[138] Die Werte wurden analog zu den Ergebnissen in Abschnitt 4.3.7.3 ermittelt und können daher direkt zueinander in Relation gesetzt werden. Zur besseren Übersichtlichkeit haben wir in Tabelle 4.22 diese früheren Ergebnisse (siehe Tabelle 4.20 auf Seite 175) in einer angepassten Darstellung vorangestellt.

---

135. Durch den Einsatz der Zeitsimulation (siehe Abschnitt 3.5.2.2) ist die Ausführung der Implementierungen vollständig performanceunabhängig.

136. So könnte z. B. bei einer besonders ineffizienten Implementierung ein Übertragungs-Timeout auftreten, durch den ein völlig neues Verhalten verursacht würde.

137. Plattform: Intel P4, 1x 2,524 GHz CPU, 1 GByte RAM, Debian Linux, Kernel 2.4.26, gcc 2.95.4, pet 2.01, xec 1.3.3(+) mit „`-m optstat`“

138. Die angegebenen Toleranzen ergeben sich aus den Standardabweichungen der zur Beseitigung statistischer Fehler durchgeführten Versuchswiederholungen und aus den Genauigkeiten der Zeitmessungen.

Tabelle 4.22: Vergleichswerte zählbarer Größen des XTP-Benchmarks (ACTIVITY)

Optimierung der Auswahlverfahren	Auswahl		Ausführung	
	Anzahl (Trans.)	Speedup	Anzahl (Trans.)	Speedup
keine	55.223.831	= 1,0	149.305	= 1,0
nur global	16.213.323	3,4	149.293	1,0
nur lokal	9.220.082	6,0	149.305	1,0
global und lokal	2.427.581	22,7	149.296	1,0

Tabelle 4.23: Realzeitmessung XTP-Benchmark (ACTIVITY)

Optimierung der Auswahlverfahren	Auswahl (Summe)		Ausführung (Summe)	
	Zeit [sec]	Speedup	Zeit [sec]	Speedup
keine	3,8287 ± 0,0011	= 1,0	0,1314 ± 0,0003	= 1,0
nur global	1,1588 ± 0,0011	3,3	0,1327 ± 0,0005	1,0
nur lokal	0,9679 ± 0,0008	4,0	0,1110 ± 0,0004	1,2
global und lokal	0,3321 ± 0,0025	11,5	0,1220 ± 0,0011	1,1

Dazu stellen wir der Summe der Ausführungs- und Auswahlzeiten des Systems (Tabelle 4.23) die Anzahl der dabei jeweils ausgeführten oder getesteten Transitionen (Tabelle 4.22) gegenüber. Vergleicht man nun den „Speedup“ der Auswahl- und Ausführungszeiten mit dem Verbesserungsfaktor[139] der gezählten Größen, so zeigt sich, dass speziell die reine globale Optimierung fast exakt die aus den Zählungen resultierenden Erwartungswerte erreicht (3,4 bzw. 3,3). Deutliche Abweichungen treten dagegen bei der lokalen Optimierung auf. Hier werden vom erwarteten Speedup nur 2/3 erreicht (4,0 statt 6,0). Noch deutlicher fällt dies bei der Kombination beider Optimierungsebenen aus: Der hier erreichte Speedup beträgt nur etwa die Hälfte des erwarteten.

Tabelle 4.24: Realzeitmessung XTP-Benchmark (ACTIVITY) pro Transition

Optimierung der Auswahlverfahren	Auswahl (pro Transition)		Ausführung (pro Transition)	
	Zeit [10^{-6} sec]	rel.	Zeit [10^{-6} sec]	rel.
keine	0,0693 ± 0,0002	= 100 %	0,8803 ± 0,0015	= 100 %
nur global	0,8888 ± 0,0029	103 %	0,8888 ± 0,0029	101 %
nur lokal	0,1050 ± 0,0002	151 %	0,7435 ± 0,0023	85 %
global und lokal	0,1368 ± 0,0011	197 %	0,8170 ± 0,0066	93 %

139. Wir bezeichnen diesen ebenfalls als „*Speedup*“.

Diese Ergebnisse bestätigen sich, wenn man die Auswahl- und Ausführungszeiten auf die einzelnen Transitionen umrechnet (siehe Tabelle 4.24).

Offenbar wirkt sich hier bereits der höhere Aufwand für das Management der Optimierungsverfahren aus. Ein anderer Faktor ergibt sich aus der Zusammensetzung der jeweils getesteten Transitionen: Gerade die lokale Optimierung führt dazu, dass speziell diejenigen Transitionen, deren Schaltbereitschaft nur oder fast nur von ihren (zum Teil sehr aufwändigen) `PROVIDED`-Klauseln abhängen, einen größeren Anteil an den getesteten Transitionen einnehmen (Abschnitt 4.3.7.3). Dies führt dann zu einer Verschlechterung der mittleren Ausführungsdauer.

Eine interessante Randbeobachtung ergibt sich aus der Auswertung der Ausführungszeiten. Entgegen den Erwartungen, dass die Optimierungsmaßnahmen bei der Auswahl aufgrund des mit ihnen verbundenen Verwaltungsoverheads[140] eine Verschlechterung der Ausführungszeiten bewirken, ist dieser Effekt bei den Messdaten sogar invertiert: Der Zeitaufwand pro ausgeführter Transition *sinkt* signifikant um bis zu 15 % (siehe Tabelle 4.24), obwohl die Menge der geschalteten Transitionen praktisch[141] identisch ist. Die Ursache für diesen Effekt ist vermutlich in den Optimierungsmechanismen der zu Grunde liegenden Hardwareplattform zu suchen. So bewirkt die effizientere Auswahl, dass weniger unterschiedliche Transitionen getestet werden müssen und somit auch ein kleinerer Workingset für Code und Daten erforderlich ist. Dies kann die Wirksamkeit von hardwarebasierten Optimierungstechniken (*Code- und Daten-Cache, Translation Lookaside Buffers*, etc.) verbessern und so zu dem beschriebenen Effekt führen. Wir werden später in Kapitel 6 ähnliche Effekte bei der Handhabung großer Datenmengen feststellen können.

Welche Bedeutung die Optimierung der Transitionsauswahlphase besitzt, lässt sich anhand der gemessenen Zeiten daran erkennen, dass im nicht optimierten Fall die Transitionsauswahl mehr als 29 mal mehr Zeit beansprucht, als die Ausführung der Transitionen. Entsprechend könnte eine reine Optimierung der Ausführungsphase keine nennenswerten Leistungssteigerung erreichen, da sie insgesamt nur ca. 3 % der gesamten Ausführungszeit in Anspruch nimmt.

Durch die vorgestellten Optimierungen konnte das Verhältnis auf 2,7 : 1 verbessert werden, wodurch mögliche Optimierungen der Ausführungsphase (nun ca. 25 % Zeitanteil) sinnvoll werden. Bei einfacheren Spezifikation ist das Verhältnis meist sogar noch deutlich ausgeglichener.

Insgesamt bestätigt die realzeitbezogene Auswertung mit dem erreichten Speedup von 11,5 die Wirksamkeit der Optimierungsmaßnahmen. Dies ist insbesondere auch vor dem Hintergrund der angewandten XTP-Benchmarkspezifikation zu betrachten, die aufgrund ihrer hohen Komplexität und des nur wenig leistungsorientierten Spezifikationsstils keine extrem performanten Implementierungen erwarten ließ.

140. Ein Teil der Verwaltungsoperationen (speziell bei der ereignisgesteuerten Optimierung, siehe Abschnitt 4.2.2) findet während der Schaltphase statt.

141. Der Anteil der Abweichungen liegt unter 10^{-5}.

4.5. Zusammenfassung

Wir haben verschiedene Techniken zur Optimierung der Kontrollflusssteuerung von automatisch generierten Estelle-Implementierungen entwickelt und mit Hilfe des in Kapitel 3 eingeführten Estelle-Implementierungsgenerators XEC implementiert und evaluiert. Diese Optimierungsansätze können in drei Ebenen untergliedert werden:

- die Transitionsausführung,
- die globale Modulauswahl und
- die modullokale Transitionsauswahl.

Die Optimierung der *Transitionsausführung* haben wir auf die Effizienz der dabei auszuführenden Basisoperationen zurückgeführt. Auf Grund des von XEC verfolgten Implementierungskonzepts der möglichst direkten Abbildung von Estelle-Operationen (speziell auf der weitgehend von Pascal abstammenden Statement-Ebene) kann zunächst einmal davon ausgegangen werden, dass der gewonnene C++-Code weitgehend dem bei einer Handimplementierung der gleichen Operationen eingesetzten entspricht (s. u.), und somit weitergehende Optimierungsgewinne nur durch konventionelle Compileroptimierungstechniken möglich wären. Zudem hat die Transitionsausführungsphase am gesamten Zeitverbrauch oft nur einen sehr geringen Anteil (3 % beim XTP-Benchmark ohne Auswahloptimierung), so dass eine Optimierung hier praktisch keinen Effekt hätte.

Hinsichtlich der spezifischen Problemstellungen bei der Implementierung formaler Spezifikationstechniken sind die *Auswahlfragestellungen* wesentlich relevanter, zumal hier durch die Anwendung *heuristischer Optimierungen* erhebliche Leistungssteigerungen erreicht werden konnten. Dazu haben wir die Evaluierung der einzelnen Optimierungsverfahren zunächst auf der Basis ausführungszeitunabhängiger Kriterien durchgeführt, da diese neben den experimentellen Ergebnissen aus der Anwendung auf Benchmarkspezifikationen auch analytische Vorhersagen der Effizienz ermöglichen.

Die hier vorgestellte Optimierung der *globalen Modulauswahl* basiert auf der Übertragung der Grundideen des Activity-Thread-Modells auf das von XEC gemäß der Estelle-Semantik realisierte Server-Modell. Das daraus entwickelte hybride, ereignisgesteuerte Auswahlmodell bietet die Effizienz des Activity-Thread-Modells, ohne dessen extreme stilistische Beschränkungen zu erzwingen. Die Effizienz des hybriden Modells *skaliert* dabei mit dem Grad der Einhaltung dieser Beschränkungen. Weitere Optimierungsansätze dienen dazu, die negativen Auswirkungen der Nichteinhaltung der genannten Vorbedingungen zu reduzieren oder gar völlig zu kompensieren. Sehr gute Ergebnisse konnten dabei durch die Einführung der *Independent-Module-Erweiterung* erreicht werden, die eine Senkung der Modulsynchronisation ermöglicht. Aber auch schon allein auf Basis des Auswahlverfahrens konnten mit der Inaktivitätserkennung und den erweiterten Aktivitätssteuerungen wirksame Optimierungen realisiert werden. So konnten bzgl. der Modultests im Vergleich zum Referenzauswahlverfahren Speedups von 10 und mehr erreicht werden.

Bei der auf die globale Auswahl aufbauenden *modullokalen Transitionsauswahl* konnten mehrere Verfahren entwickelt werden, die jeweils eine Reduktion der zu betrachtenden Transitionen oder eine Optimierung durch eine geschickte Reihenfolge bei den Transitionstests erreichen. Die Verfahren bauen teilweise aufeinander auf bzw. lassen sich sogar auf verschiedenen Ebenen kombinieren. Die Effizienz der modullokalen Auswahl wird bei globaler Betrachtung (der

Anzahl der systemweit getesteten Transitionen) durch eine nicht optimale globale Auswahl beeinträchtigt; jedoch zeigte sich, dass insbesondere die lokale *warteschlangenbasierte Auswahl* teilweise die Schwächen der übergeordneten globalen Modulauswahl kompensieren kann.

Durch eine experimentelle Evaluierung der Optimierungsverfahren anhand unterschiedlich komplexer Spezifikationen konnte ihre Wirksamkeit nicht nur bezüglich der zählbaren Größen (z. B. der Anzahl der getesteten oder geschalteten Module und Transitionen), sondern auch bezüglich der *Ausführungszeiten* nachgewiesen werden. Insbesondere konnte z. B. beim XTP-Benchmark der Anteil der Transitionsauswahl an der Gesamtausführungszeit um den Faktor 11,5 verringert werden.

Die quantitative Evaluierung der Transitionsauswahl- und -Ausführungszeiten haben zudem nochmals unseren ursprünglichen Ansatz bestätigt, unsere Optimierungsbemühungen auf die effiziente Auswahl von Transitionen auf Managementebene zu fokussieren, da diese (beim XTP-Benchmark) ohne die Optimierungen fast 30 mal mehr Zeit beansprucht hat als die Transitionsausführung und letztere somit zunächst praktisch keinerlei Optimierungspotential bot. Die bisher untersuchten Optimierungen haben dieses Verhältnis um mehr als den Faktor 10 auf 2,7 : 1 verringert. Inwieweit das hier offensichtlich immer noch vorhandene Optimierungspotential genutzt werden kann, bleibt zu erforschen.

Viele der hier vorgestellten Optimierungskonzepte können auch auf andere automatenbasierte formale Spezifikationstechniken wie z. B. SDL [ITU94] übertragen werden. Besonders interessant sind hier die Techniken zur ereignisgesteuerten Optimierung des Auswahlprozesses auf verschieden Ebenen. Inwieweit eine solche Übertragung auf andere Beschreibungstechniken sinnvoll möglich ist hängt jedoch sehr stark von der Verfügbarkeit einer vergleichbaren Plattform zur Realisierung und quantitativen Evaluierung von Implementierungstechniken ab.

Wir beenden damit unsere Untersuchungen zur Optimierung von Managementfunktionen und wenden uns im nächsten Kapitel der Formalisierung und Abstraktion von Datentypen zu. Wir werden jedoch später in Kapitel 6 nochmals auf den Aspekt der effizienten Implementierung zurückkommen, uns dann jedoch auf die Übertragung von Nutzdaten konzentrieren.

5. Datenabstraktion und Typhierarchien

Unsere bisherigen Betrachtungen zur Ausdrucksfähigkeit und effizienten Implementierbarkeit formaler Beschreibungstechniken waren auf Aspekte des *Kontrollflusses* konzentriert. In diesem und dem folgenden Kapitel beschäftigen wir uns nun mit dem Einsatz formaler Beschreibungstechniken bei der *Übertragung von Daten* zwischen und innerhalb von Kommunikationssystemen. Dabei untersuchen wir die Datenübertragungsproblematik auf den Ebenen der formalen Spezifikation und der Möglichkeit zur Gewinnung effizienter Implementierungen zunächst anhand von Estelle und betrachten dann jeweils die Übertragbarkeit der Ergebnisse auf andere FDTs.

Wir beginnen mit der Fragestellung, wie Datenübertragungsdienste *anwendungsneutral*, *wiederverwendbar* und *problemorientiert* mit formalen Beschreibungstechniken spezifiziert werden können. Wir werden dabei sehen, dass die in Estelle verfügbaren Mittel nicht geeignet sind, solche abstrakten Dienstschnittstellen und Dienste angemessen zu formalisieren. Zur Lösung der Aufgabenstellung entwickeln wir am Beispiel von Estelle die syntaktische und semantische „*Containertyp-Erweiterung*“ und demonstrieren ihren praktischen Einsatz auf Spezifikationsebene und auf Implementierungsebene. Ziel ist es dabei, auf Spezifikationsebene die Übertragung dienstunspezifischer[1] Nutzdaten *generisch*, aber trotzdem *typsicher* beschreibbar zu machen.

Die typsichere Übertragung von Daten ist eine der wesentlichen Grundlagen für die formale (d. h. syntaktisch und semantisch eindeutige) Spezifikation von hierarchisch strukturierten Kommunikationssystemen. Wir werden in diesem Kapitel jedoch zeigen, dass diese Typsicherheit einer generischen Spezifikation von Kommunikationsdiensten und -Protokollen entgegensteht (Abschnitt 5.1). Dies wird besonders bei komplexen heterogenen Spezifikationen und vor dem Hintergrund der Wiederverwendung von Komponenten (z. B. auf Basis von *Open-Estelle*, siehe Kapitel 7) deutlich.

Zur Lösung dieses Defizits führen wir mit der *Containertyp-Erweiterung* (Abschnitt 5.2) eine kompatible syntaktische und semantische Erweiterung von Estelle ein, die eine generische Spezifikation von Datenübertragungsdiensten ermöglicht, ohne die Typsicherheit und damit die semantische Präzision der Beschreibung zu beeinträchtigen. Dazu untersuchen wir verschiedene Anwendungsaspekte der Erweiterung auf *Spezifikationsebene* (Abschnitt 5.3), indem wir anhand verschiedener praktisch eingesetzter Protokolle (wie IPv4 und IPv6) die formale Spezifikation heterogener und zum Teil auch dynamisch strukturierter Nutzdatentypen demonstrieren.

Anschließend betrachten wir den praktischen Einsatz der Erweiterung auf *Implementierungsebene* (Abschnitt 5.4) anhand einer Referenzimplementierung auf Basis des Implementierungsgenerators XEC und diskutieren schließlich noch die Übertragbarkeit der Containertyp-Erweiterung auf andere FDTs (Abschnitt 5.5).

---

1. im Sinne einer reinen Nutzlast, die uninterpretiert und unverändert vom Dienst übertragen wird

5.1. SDU-Typen in Diensthierarchien

Ein wesentliches Mittel zur Strukturierung von Kommunikationssystemen sind *Diensthierarchien*. Dabei wird ein System in eine Reihe von Schichten aufgeteilt, wobei jede Schicht der darüber liegenden Schicht einen spezifischen Dienst anbietet, den sie ggf. aufbauend auf den Diensten der darunter liegenden Schicht realisiert (siehe Abb. 5-1). Die sich ergebende *Dienstschnittstelle* zwischen einem Diensterbringer (z. B. *N*) und einem Dienstnutzer (z. B. *N+1*) abstrahiert dabei von den Details der Realisierung des Dienstes durch den Diensterbringer und die darunter liegenden Schichten. Dies erlaubt es, den Dienst und damit auch mögliche Diensterbringer unabhängig von ihrer konkreten Realisierung zu beschreiben (z. B. N^A und N^B rechts in Abb. 5-2).

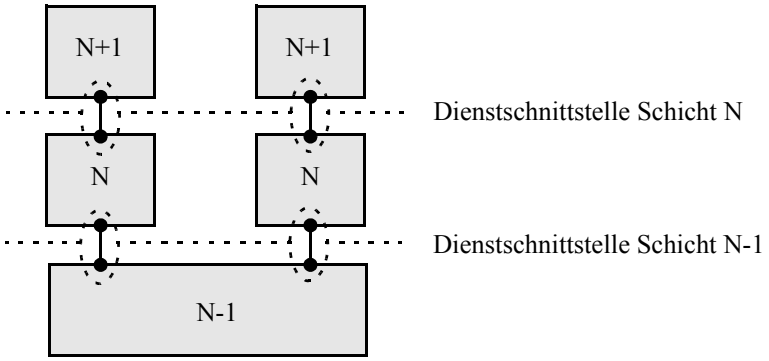


Abbildung 5-1: Schichtenmodell zur Strukturierung von Kommunikationssystemen

Umgekehrt soll die Dienstschnittstelle auch den Diensterbringer von den Interna der jeweiligen Dienstnutzer isolieren, sodass der Diensterbringer (im Rahmen der vorgesehenen Nutzung des angebotenen Dienstes) unabhängig von den späteren Dienstnutzern beschrieben werden kann. Dies gilt besonders vor dem Hintergrund, dass auf den selben Diensterbringer über die Dienstschnittstelle ganz unterschiedliche Dienstnutzer (z. B. $N+1^A$ und $N+1^B$ links in Abb. 5-2) zugreifen können, deren spezifische Eigenschaften zum Zeitpunkt der Spezifikation des Dienstes bzw. des Diensterbringers nicht notwendigerweise bereits bekannt waren.

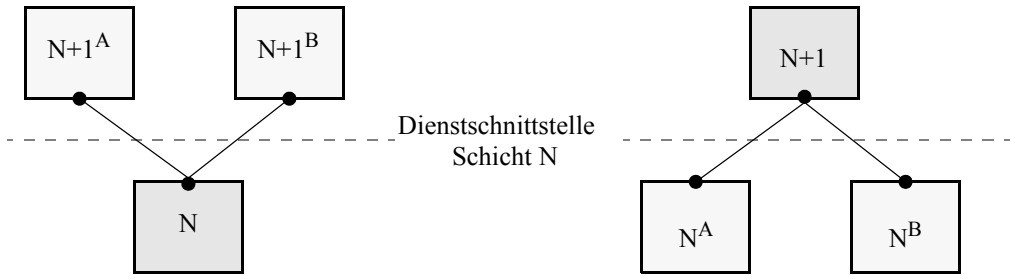


Abbildung 5-2: Unabhängigkeit von Dienstnutzern und Diensterbringern

Im OSI-Referenzmodell [ISO81] werden die Dienste des Diensterbringers der Schicht N (service provider) den Dienstnutzern der Schicht *N+1* (service user) über Dienstzugangspunkte (service access points, SAPs) zugänglich gemacht. Über diese SAPs werden zum Zugriff auf die Dienstelemente *typisierte Nachrichten* ausgetauscht, die gegebenenfalls Parameter enthalten.

5.1.1 Generische Datenübertragungsdienste

Wichtige Dienstelemente im OSI-Referenzmodell sind *generische Datenübertragungsdienste*, also das Versenden und Empfangen von Daten (`DATA.request` und `DATA.indication`), wobei die hier als Parameter der jeweiligen Sende- und Empfangsnachrichten übertragenen *Nutzdaten* („*service data unit*“, *SDU*) vom Diensterbringer typischerweise unverändert transportiert werden sollen. Solche Daten werden im Rahmen der durch den Dienst festgelegten Randbedingungen (z. B. max. Größe) in Struktur und Inhalt meist allein vom Dienstnutzer definiert; ihre interne Struktur ist für die Erbringung des Transportdienstes unerheblich und bleibt daher den potenziellen Diensterbringern verborgen. Entsprechend *abstrahiert* die Dienstschnittstelle die SDUs meist als abstraktes Datum oder (in implementierungsnaher Sicht) als unstrukturierte Byte-Sequenz endlicher Länge. Gemäß der geforderten Datenabstraktion zwischen den Schichten kann der Diensterbringer auf Inhalt und Struktur der SDU in einem solchen Fall keinen direkten Zugriff nehmen; lediglich Eigenschaften der unstrukturierten binären Darstellung (wie Anzahl der Bytes oder Prüfsummen) sind ableitbar.

Die durch den SAP der Dienstschnittstelle zwischen Diensterbringer (*N*) und Dienstnutzer (*N+1*) ausgetauschte SDU (*N*-SDU) wird bei einem Datentransportdienst typischerweise zusammen mit protokollspezifischen Zusatzinformationen zu einer neuen Datenstruktur, der *Protokolldateneinheit* („*protocol data unit*“, *PDU*, siehe Abb. 5-3) aggregiert (*Framing*) und an die Partnerinstanz des Diensterbringers der Schicht *N* auf dem Zielknoten weitergereicht, indem sie als *N-1*-SDU an die nächst tiefere Schicht weitergeleitet wird.

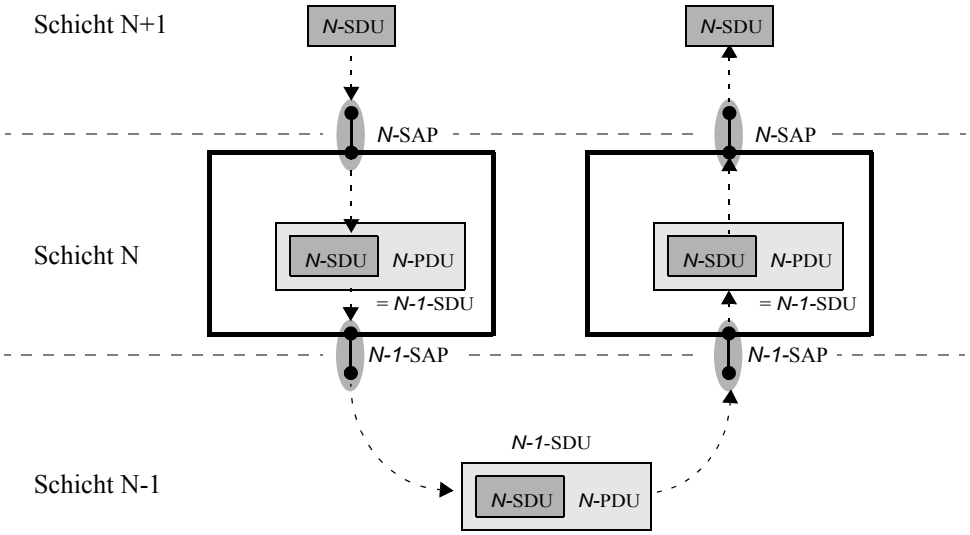


Abbildung 5-3: Verschachtelung von SDUs in PDUs bei Datentransportdiensten

Aus implementierungsnaher Sicht eines Dienstes (Bytekodierungsebene) wird der Vorgang der Bildung einer *N*-PDU im einfachsten Fall als die Konkatenation der jeweiligen Byte-Sequenzen eines *N*-PDU-Headers, der übergebenen *N*-SDU und eines *N*-PDU-Trailers[2] interpretiert (*einfache SDU-Aggregation* bzw. *Framing*, siehe Abb. 5-4). Die sich dabei ergebende Byte-Sequenz enthält die *N*-SDU-Bytesequenz und kann als *N-1*-SDU zur Weitergabe an die nächst tiefere Schicht dienen. In komplexeren Diensten treten aber auch aufwändigere SDU-Abbildungen auf, die von Fragmentierungen der SDU-Bytesequenz in mehrere Teilstücke bis zur auf-

2. PDU-Header und -Trailer können dabei jeweils auch leer sein (Byte-Sequenz der Länge 0).

wändigen Datenkompression oder kryptografischen Kodierung der SDUs reichen, durch die in der übertragenen *N*-PDU u. U. nicht einmal mehr Fragmente der zu übertragenden Byte-Sequenz (*N*-SDU) zu finden sind.

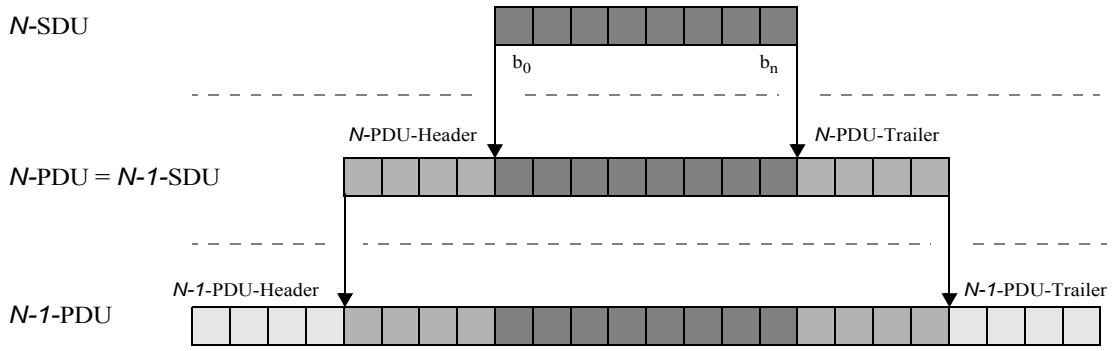


Abbildung 5-4: Verschachtelung von SDUs in PDUs auf Bytekodierungsebene

In den folgenden Abschnitten betrachten wir zunächst Dienste mit reiner SDU-Aggregation, bei denen die *N*-PDU die vom Dienstnutzer übergebene *N*-SDU unverändert enthält; mit den komplexeren Diensten beschäftigen wir uns dann später im Zusammenhang mit der effizienten Implementierung von Datenübertragungen in Kapitel 6.

5.1.2 Typsicherheit in FDTs

Ein wichtiger Aspekt bei der Verarbeitung von Daten in formal beschriebenen Systemen ist die Kompatibilität der beteiligten Typen. Beispiele sind die Zuweisungen von Werten auf Variablen oder die Parametrierung von Prozeduren, Funktionen und Interaktionen. In formalen Beschreibungstechniken wie Estelle wird hier das Grundprinzip der *strengen Typprüfung* angewandt (siehe auch Abschnitt 6.4, Annex C von [ISO97]). Dies bedeutet, dass Typen grundsätzlich nur dann *zuweisungskompatibel* sind, wenn sie auf die (identisch) selbe Definition zurückzuführen sind. Diese strenge Typprüfung ist eine der Grundlagen der formalen Syntax und Semantik der FDTs, da sie die Semantik der Wertzuweisung durch die Beschränkung auf typgleiche Datenquellen und Zuweisungsziele auf eine einfache und implementierungsunabhängige mathematische Basis stellt.

Die strenge Typprüfung wird nur an einigen wenigen Stellen aufgeweicht, indem implizite oder explizite Typkonvertierungen eingesetzt werden. Diese betreffen jedoch meist nur einfache Ordinaltypen, die im Allgemeinen semantisch nahe liegend konvertiert werden können. Ein Beispiel ist die im folgenden Estelle-Fragment auftretende, implizite Konvertierung von einem Teilbereichstyp `T` (ganze Zahlen von 0 bis 9) zu dessen Obertyp `INTEGER` (alle ganzen Zahlen):

Beispiel 5.30: Implizite Typkonvertierung in Estelle

```
TYPE T = 0..9;
VAR  x: T;
     y: INTEGER;
     (* ... *)
y := x;
```

(Ende von Beispiel 5.30)

Die Semantik dieser Zuweisung und der damit verbundenen Typumwandlung ist offensichtlich und kann für alle zulässigen Werte von x leicht auf abstrakter Ebene mathematisch formalisiert werden, da hier der Wertebereich von `T` eine Teilmenge des Wertebereichs von `INTEGER` ist. Bemerkenswerterweise existiert auch eine Regel für die implizite Typumwandlung in Gegenrichtung. Hier muss natürlich der Sonderfall behandelt werden, dass es `INTEGER`-Werte gibt, die keine zulässigen `T`-Werte sind (z. B. 10). Das Ergebnis einer solchen unzulässigen Wertzuweisung bzw. der dazu notwendigen Typumwandlung ist entsprechend *semantisch undefiniert* (Abschnitt 6.4.6, Annex C von [ISO97]). Konvertiert man jedoch einen gültigen Wert des Typs T in den Typ `INTEGER` und anschließend wieder zurück in den Typ `T`, so ergibt sich immer der Ausgangswert. Der Typ `INTEGER` kann also als eine Art *Container* für Werte des Typs `T` dienen. Auf diese Betrachtungsweise werden wir später nochmals genauer eingehen.

Anders als bei einfachen Ordinaltypen wird zwischen komplexen Typen keine implizite oder explizite Konvertierung zugelassen. Dies gilt besonders für benutzerdefinierte strukturierte Typen, wie z. B. Records oder Arrays. So ist z. B. zwischen den folgenden strukturverschiedenen Estelle-Typen `T1`, `T2` und `T3` keine nahe liegende abstrakte Konvertierung erkennbar:

Beispiel 5.31: komplexe, konvertierungsinkompatible Estelle-Typen

```
TYPE T1 = SET OF 0..7;
     T2 = RECORD
              a: REAL;
              b, c: BOOLEAN;
          END;
     T3 = ARRAY [0..9] OF CHAR;
```

(Ende von Beispiel 5.31)

Entsprechend existiert auch keine Möglichkeit zur expliziten oder impliziten Konvertierung zwischen diesen drei Typen, die nicht auf einer komponentenweisen, explizit spezifizierten Umwandlung der jeweiligen Strukturkomponenten basiert. Allgemein impliziert die Typsicherheit von Estelle, dass es zu strukturierten Typen wie z. B. Records keine davon unabhängig definierten[3], zuweisungskompatiblen Typen gibt. Wir werden im folgenden Abschnitt sehen, wie sich diese Typsicherheit bei der Definition von Diensthierarchien auswirkt.

5.1.3 Repräsentation von Diensthierarchien in Estelle

Zur Strukturierung von Systemen dient in Estelle das Konzept des Moduls. Zur Kommunikation zwischen den daraus abgeleiteten Modulinstanzen dienen externe Interaktionspunkte, durch die gemäß der zu Grunde liegenden Kanaldefinition wohldefinierte Nachrichten (*Interaktionen*) verschickt bzw. empfangen werden können. Teil der Definition der jeweiligen Interaktionstypen ist dabei auch ein fester Satz von *Interaktionsparametern* mit wohldefinierten Estelle-Typen.

Zwei Interaktionspunkte können nur dann zum Zwecke eines Datenaustauschs miteinander verbunden werden, wenn sie von der selben Kanaldefinition abstammen und darüber hinaus in diesem Kanal entgegengesetzte Rollen einnehmen. Entsprechend werden Interaktionen typerhaltend zwischen den beiden verbundenen Interaktionspunkten transportiert, und es ergibt sich von

3. d. h. die beiden Typen greifen in ihren Definitionen nicht direkt oder indirekt aufeinander zurück

der typsicheren Übergabe der Nachrichtenparameter beim Versand einer Interaktion bis zu ihrem Empfang und der Weiterverarbeitung der Parameter am Zielinteraktionspunkt eine *durchgängige Kette von typerhaltenden Übertragungsschritten*.

Zur Modellierung von Dienst- bzw. Protokollhierarchien bieten sich in Estelle zwei unterschiedliche Architekturen an. Die *protokollorientierte* Architektur hebt den Aspekt der Protokollinstanzen hervor, indem sie diese jeweils als eine Modulinstanz modelliert, welche externe Interaktionspunkte nach „oben“ (bereitgestellte Dienstschnittstelle) und „unten“ (genutzte Dienstschnittstelle) besitzt. Diese Protokollinstanzen lassen sich zu einer Diensthierarchie zusammenfügen, wie sie in Abb. 5-1 auf Seite 184 dargestellt ist.

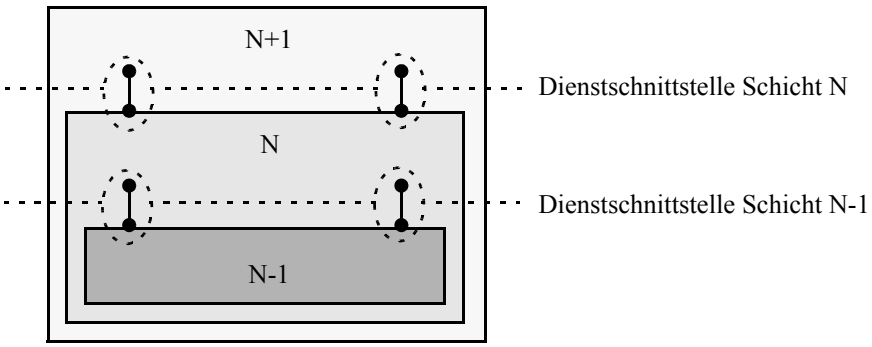


Abbildung 5-5: Verschachtelung von Diensten

Die alternative *dienstorientierte* Architektur stellt die Darstellung der Dienste selbst in den Vordergrund. Hier wird eine hierarchische Struktur aufgebaut, in der ein Dienst jeweils die ihm zu Grunde liegenden Dienste strukturell enthält. Dies wird in Estelle dadurch dargestellt, dass die Modulinstanz, die einen spezifischen Dienst bereitstellt, jeweils den von ihr genutzten Dienst als Kind-Modulinstanz enthält. Die in Abb. 5-5 dargestellte Modulinstanzhierarchie zeigt eine solche Struktur in Analogie zum vorangegangenen Beispiel.[4]

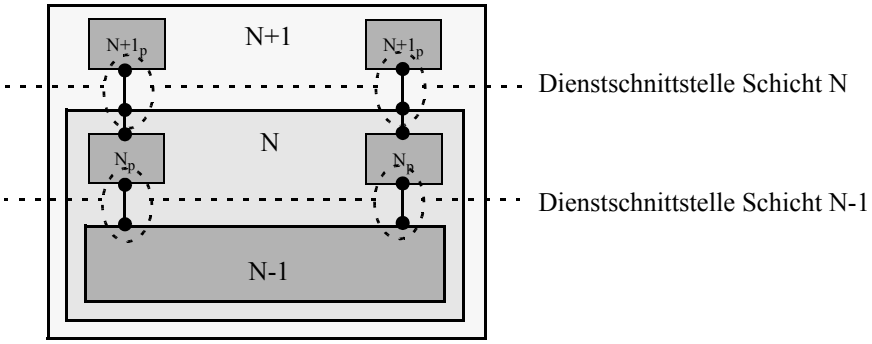


Abbildung 5-6: Protokollinstanzen in verschachtelten Diensten

Die dienstorientierte Architektur erfordert jedoch zur Modellierung der (räumlichen) Verteilung jedes Diensterbringers eine strikte Trennung der Zustände, Interaktionen und Funktionalitäten der örtlich getrennten Komponenten jedes Dienstes (mit Ausnahme des untersten Diensterbringers, *N-1* in Abb. 5-5). Dies erfolgt naheliegenderweise meist durch die Einkapselung der Protokollmaschinen in eigene Modulinstanzen innerhalb ihrer Dienst-Modulinstanz (z. B. N_p in

4. Eine technische Besonderheit von Estelle bedingt, dass zur Kommunikation mit Kind-Modulinstanzen über deren externe Interaktionspunkte jeweils eine Verbindung zu einem lokalen Hilfs-Interaktionspunkt notwendig ist.

Abb. 5-6). Damit ergibt sich eine Architektur, die bezüglich der Modulinteraktion äquivalent zu der oben genannten protokollorientierten Architektur ist und somit im Folgenden nur als Spezialfall der protokollorientierten Architektur Beachtung findet.

5.1.4 SDU-Typen in formal beschriebenen Diensthierarchien

Obwohl formale Beschreibungstechniken auf den ersten Blick gut geeignet erscheinen, um hierarchisch strukturierte Kommunikationssysteme zu spezifizieren, zeigt sich bei ihrem praktischen Einsatz, dass die Typsicherheit der FDTs einer angemessenen Spezifikation der Dienstschnittstelle entgegensteht, sobald ein *generischer Datenübertragungsdienst* (siehe Abschnitt 5.1.1) modelliert werden soll.

Die Aufgabe eines solchen Dienstes ist es gerade, ein vom Dienstnutzer frei vorgegebenes Datenobjekt zu übertragen. Die Dienstschnittstelle soll dabei unabhängig von den später eingesetzten Dienstnutzern und den konkreten Typen der von ihnen übergebenen Datenobjekte formalisiert werden. Dies impliziert, dass verschiedene unabhängig voneinander definierte Dienstnutzer auch Instanzen unterschiedlicher, unabhängig voneinander definierter Datentypen als Parameter des abstrakten Datenübertragungsdienstes einsetzen können. Wir werden nun zeigen, dass dies der geforderten Typsicherheit formaler Beschreibungstechniken widerspricht. Das Problem soll anhand eines Beispiels verdeutlicht werden:

Beispiel 5.32: Generischer Datenübertragungsdienst

In der in Abb. 5-1 auf Seite 184 dargestellten Diensthierarchie soll die Dienstschnittstelle der Schicht *N* zwei Dienstprimitive zum Senden (`D_Send(Td)`) bzw. Empfangen (`D_Recv(Td)`) von nicht dienstspezifischen Daten (`Td`) bieten. Diese Daten werden insgesamt von den beiden Protokollinstanzen *N* und dem Nachrichtentransportmedium *N-1* unverändert und verlustfrei übertragen. Ein Dienstnutzer *N+1* überträgt über diesen Dienst eine Folge von Daten des Typs `T1` (siehe Typdefinition in Beispiel 5.31 auf Seite 187).

Bei der Spezifikation des Systems in Estelle wird, wie in Abschnitt 5.1.3 beschrieben, jede Protokollinstanz durch eine Modulinstanz modelliert. Entsprechend werden die Dienstschnittstellen durch geeignete Kanaldefinitionen repräsentiert. Für die Dienstschnittstelle der Schicht N würde diese naheliegenderweise folgendermaßen aussehen:

Beispiel 5.32-a: Dienstschnittstelle eines abstrakten Datenübertragungsdienstes

```
CHANNEL ChService_N(User,Provider);
     BY User:      D_Send(Data: Td);
     BY Provider:  D_Recv(Data: Td);
```

(Ende von Beispiel 5.32-a)

Die essentielle Frage ist nun, wie der SDU-Typ `Td` zu beschreiben ist. Die Typsicherheit von Estelle bedingt, dass sich die verbundenen Interaktionspunkte von Dienstnutzer und Diensterbringer auf die selbe Kanaldefinition und damit auch auf die selbe Definition von `Td` beziehen. Da der Dienstnutzer *N+1* einen Wert des von ihm definierten Typs `T1` als Parameter des Sendeprimitivs `D_Send(Td)` übertragen soll, bietet es sich zunächst an, `Td` identisch mit `T1` zu definieren:

Beispiel 5.32-b: Anpassung der Dienstschnittstelle an einen konkreten PDU-Typ

```
TYPE T1 = SET OF 0..7;
     Td = T1;
```

(Ende von Beispiel 5.32-b)

Diese Definition von `Td` erlaubt es dann dem Dienstnutzer, Werte des Typs `T1` direkt und ohne weitere Konvertierungen als Nachrichtenparameter einzusetzen:

Beispiel 5.32-c: Dienstnutzer zu Beispiel 5.32-a

```
IP IpToProvider: ChService_N(User);
VAR x: T1;
TRANS
     BEGIN
     x := {...};
     OUTPUT IpToProvider.D_Send(x);
     END;
```

(Ende von Beispiel 5.32-c)

Die Nachteile dieser Vorgehensweise werden offensichtlich, sobald man den Dienstnutzer $N+1$ (im Folgenden auch als $N+1^A$ bezeichnet) durch einen unabhängig von ihm definierten Dienstnutzer $N+1^B$ ersetzt, der Alternativ zu $N+1^A$ auf die Dienstschnittstelle der Schicht N zugreifen soll (siehe Abb. 5-2 links), jedoch als SDU-Typ den inkompatiblen Typ `T2` einsetzt (siehe Typdefinition in Beispiel 5.31 auf Seite 187). Auf Grund der Typsicherheit von Estelle muss der Zugriff auf die Dienstschnittstelle der Schicht N bei allen Dienstnutzern auf Basis der gleichen Kanaldefinition erfolgen. Da jedoch weder `T1` und `T2` zueinander zuweisungskompatibel sind, noch ein anderer Estelle-Typ existiert, auf den beide Typen zugewiesen werden können, ist hier eine komplexere Lösung erforderlich.

(Ende von Beispiel 5.32)

Im Wesentlichen existieren zwei konventionelle Ansätze, um in solchen *heterogenen Systemen* den Dienstnutzern $N+1^A$ und $N+1^B$ mit ihren inkompatiblen SDU-Typen wahlfrei Zugriff auf den beschriebenen Dienst auf Basis der selben Kanaldefinitionen zu gewähren:

(i) die Definition unterschiedlicher Interaktionen für die verschiedenen Parametertypen oder

(ii) die Definition eines Strukturtyps (RECORD), der jeden der Parametertypen als Komponente enthalten kann.

Die erste Lösung führt für die SDU-Typen `T1` und `T2` zu folgender Kanaldefinition:

Beispiel 5.33: Dienstschnittstelle eines heterogenen Datenübertragungsdienstes

```
CHANNEL ChService_N(User,Provider);
     BY User:     D_Send_T1(Data: T1);
     BY User:     D_Send_T2(Data: T2);
     BY Provider: D_Recv_T1(Data: T1);
     BY Provider: D_Recv_T2(Data: T2);
```

(Ende von Beispiel 5.33)

Sie hat den Nachteil, dass bei *n* verschiedenen SDU-Typen sich nicht nur die Anzahl der Interaktionstypen ver-*n*-facht, sondern sich dieses auch auf alle Transitionen, Variablen, etc. fortsetzt, die sich auf einen SDU-Typ beziehen.

Die zweite Lösung definiert den einzigen SDU-Typ `Td` derart als Strukturtyp (idealerweise ein varianter RECORD), dass im Sinne einer *horizontalen Typkomposition*[5] für jeden der zu übertragenden Typen eine entsprechende Strukturkomponente (hier `d1` und `d2`) enthalten ist:

Beispiel 5.34-a: SDU-Typ eines heterogenen Datenübertragungsdienstes

```
TYPE Td = RECORD
          CASE 1..2 OF
               1: (d1: T1;);
               2: (d2: T2;);
          END;
```

(Ende von Beispiel 5.34-a)

Somit ergibt sich für den Diensterbringer eine geschlossene Darstellung für den N-SDU-Typ; lediglich die Dienstnutzer müssen beim Empfang einer solchen SDU die von ihnen genutzte Komponente herausziehen, bzw. beim Versenden den SDU-Typ generieren:

Beispiel 5.34-b: Dienstnutzer zu Beispiel 5.34-a

```
TRANS
     VAR tmp: Td;
     BEGIN
          tmp.d1 = x;
          OUTPUT IpToProvider.D_Send(tmp);
     END;
```

(Ende von Beispiel 5.34-b)

Diese Lösung hat implementierungstechnisch den Nachteil, dass sich die Größe des als SDU-Typ zu übertragenden varianten RECORDs unabhängig vom tatsächlich genutzten Inhalt meist nach dem Platzbedarf der größten Komponente bzw. der Summe aller Komponenten richtet. Zudem macht die notwendige Zuweisung des zu übertragenden Typs an eine Komponente des varianten RECORDs (`tmp.d1` im obigen Codefragment) bei der automatischen Codegenerierung eine zusätzliche Kopieroperation nahezu unvermeidbar (siehe auch Kapitel 6).

Alle bisher vorgestellten Lösungen haben jedoch den aus konzeptioneller Sicht weitaus schwerwiegenderen Nachteil gemein, dass zum Zeitpunkt der Kanaldefinition für die Dienstschnittstelle der Schicht *N* sämtliche Datentypen bereits bekannt sein müssen, die von in Frage kommenden Dienstnutzern als Parameter des Datenübertragungsdienstes später einmal eingesetzt werden könnten. Diese Einschränkung widerspricht diametral der ausgangs gemachten Forderung nach einer universellen Spezifikation eines generischen Datenübertragungsdienstes.

Noch deutlicher wird die Problematik, wenn man die Auswirkung der typsicheren SDU-PDU-Verschachtelung über eine mehrstufige Diensthierarchie hinweg betrachtet: Auf dem Weg des Sendeauftrags von der Anwendung hinunter zum Basiskommunikationsdienst entsteht eine verschachtelte PDU-Datenstruktur, die für jede Schicht *N* neben den protokollspezifischen Zusatzinformationen eine von der darüber liegenden Schicht definierte *N*-SDU enthält, welche ggf.

5. Die Typstruktur ist hier „horizontal“ in dem Sinne, dass alle Typvarianten auf gleicher Ebene Komponenten des Strukturtyps sind (vgl. „vertikale Typkomposition“, s. u.).

wiederum rekursiv eine *N+1*-PDU dieser nächst höheren Schicht darstellt. Offensichtlich entsteht auf dem Weg der SDUs von der Anwendung hinunter zum Basiskommunikationsdienst eine Aggregationshierarchie von Datentypen, deren Komplexität dem Niveau der angebotenen Dienste entgegenläuft (siehe Abb. 5-7).

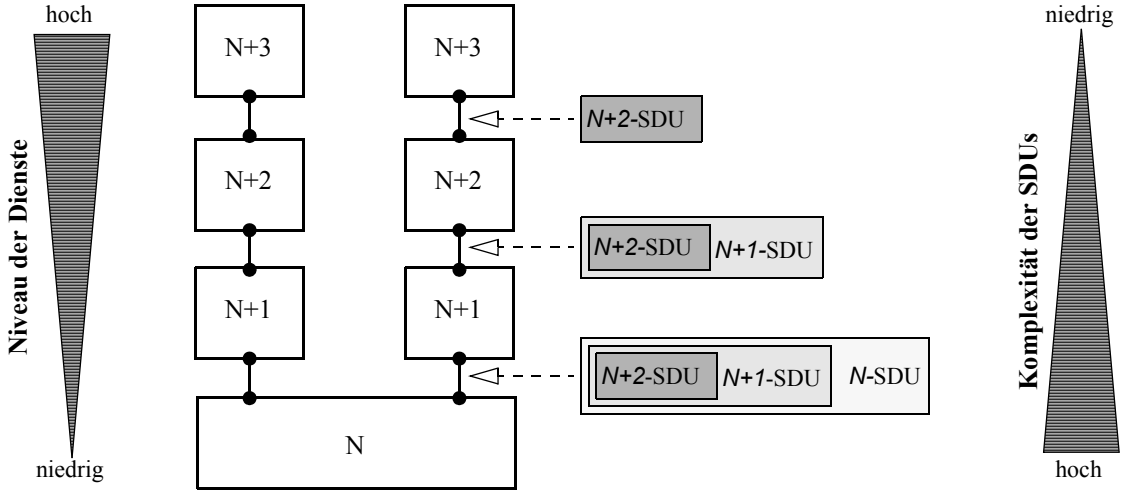


Abbildung 5-7: Gegenläufige Komplexität von Diensten und SDUs in Protokollhierarchien

Entsprechend hängt die Definition der externen Schnittstelle gerade des *unspezifischsten* Dienstes (*N* in Abb. 5-7) von den internen PDU-Definitionen aller darüber liegenden Schichten ab. Diese *vertikale Typstruktur*[6] potenziert zudem natürlich die Auswirkungen der oben beschriebenen horizontalen Typstruktur heterogener Systeme: Bei *n* Diensthierarchieebenen mit jeweils *m* PDU-verschiedenen Protokollinstanzen ergeben sich an der untersten Dienstschnittstelle $O(m^n)$ Typvarianten.

6. Die Typstruktur ist hier „vertikal“ in dem Sinne, dass die PDU-Typen durch die mehrfache PDU-SDU-Verschachtelung auf verschiedenen Ebenen der Typaggregationshierachie liegen (vgl. „horizontale Typkomposition“).

5.2. Typabstraktion und Containertypen

Bieten die oben beschriebenen Lösungsansätze in abgeschlossenen Spezifikationen mit geringer Komplexität (und insbesondere geringer Heterogenität) vielleicht noch einen akzeptablen Ausweg, so versagen sie, sobald Dienste und Protokollautomaten für einen universellen Einsatz spezifiziert werden sollen, da hier nicht auf die konkrete Definition einer geringen (oder zumindest endlichen) Zahl von SDU-Typen aller in Frage kommenden Dienstnutzer zurückgegriffen werden kann.

Besondere Relevanz erlangt die Problemstellung beim Versuch, Dienste bzw. Protokollautomaten mit Hilfe von *Open-Estelle* (siehe Kapitel 7) als wiederverwendbare offene Systeme *universell*, d. h. (innerhalb der Randbedingungen des gebotenen Dienstes) unabhängig von den SDU-Typen der späteren Dienstnutzer zu beschreiben. Ein solches offenes System hat durch die Beschreibung in Open-Estelle eine syntaktisch eindeutige äußere Schnittstelle, deren Interpretation durch den späteren Einsatzkontext nicht mehr beeinflusst wird. Entsprechend muss die Definition eines Dienstes bzw. Protokollautomaten als offenes System in Open-Estelle einerseits völlig unabhängig von den SDU-Typen der späteren Dienstnutzer erfolgen, andererseits soll im Sinne einer universellen Nutzbarkeit auch eine vollständige Kompatibilität mit allen (im Rahmen des gebotenen Dienstes zulässigen) SDU-Typen gegeben sein (siehe auch Abschnitt 7.6).

Es sei an dieser Stelle nochmals darauf hingewiesen, dass diese Problematik grundsätzlich unabhängig von Open-Estelle auch in geschlossenen Spezifikationen unter Standard-Estelle auftritt, hier jedoch aufgrund der Kenntnis der internen Details aller (in der selben Spezifikation enthaltenen) Dienstnutzer durch die bereits genannten Techniken umgangen werden kann. Diese Umgehungen führen jedoch, wie wir gesehen haben, zu einer *problemunangemessenen Modellierung der Diensthierarchie* (siehe horizontale und vertikale Typverschachtelung in Abschnitt 5.1.4).

Eine angemessene Lösung des Problems müsste hingegen das *Geheimnisprinzip* beim Zusammenwirken von Dienstnutzer und Diensterbringer besser repräsentieren: Der Diensterbringer hat bei den oben genannten Datentransportdiensten kein Interesse an der inneren Struktur der übertragenen SDU, ebensowenig wie an den darin möglicherweise eingeschachtelt enthaltenen PDUs der möglichen Dienstnutzer; aus konzeptioneller Sicht handelt es sich bei der Definition des PDU-Typs einer Protokollmaschine um einen rein internen Aspekt, der nur die entsprechenden Partnerprotokollinstanzen betrifft.

Die geforderte Einhaltung des Geheimnisprinzips kann in allgemeiner Form nur dadurch erreicht werden, dass vom konkreten Typ der von einem Dienstnutzer übergebenen SDU bei der Übergabe zum Diensterbringer abstrahiert wird, der Diensterbringer also völlig unabhängig von der von den Dienstnutzern übergebenen SDU-Struktur spezifiziert wird. Umgekehrt muss nach erfolgtem Transport der SDU-Daten bei der Übergabe vom Diensterbringer an den Dienstnutzer der abstrakte SDU-Typ wieder in genau den konkreten Typ zurück umgewandelt werden, den der Dienstnutzer ursprünglich beim Verschicken der Nachricht eingesetzt hat.

Diese *vorübergehende Typabstraktion* während der Datenübertragung entspricht auch der Sichtweise des OSI Modells, in welchem die SDU-Daten als binäre Datenobjekte von Dienstnutzern an einen Diensterbringer weitergereicht werden. Die innere Struktur der SDU-Daten und damit auch deren Bedeutung bleibt dem Diensterbringer im Allgemeinen verborgen. Aufgabe des Diensterbringers ist es bei solchen einfachen Datentransportdiensten lediglich, ein übergebenes binäres Datenobjekt *uninterpretiert* (und typischerweise auch *unverändert*) an die

empfangende Partnerinstanz des Dienstnutzers zu übertragen, wo sein Inhalt auf Grund der Kenntnis der inneren Struktur und Bedeutung durch diesen Dienstnutzer wieder interpretiert werden kann.

Im Folgenden werden wir betrachten, inwieweit Standard-Estelle bereits Mittel zum Ausdruck solcher Typabstraktionen enthält (Abschnitt 5.2.1). Anschließend definieren wir eine kompatible syntaktische und semantische Erweiterung von Standard-Estelle und Open-Estelle, die solche Typabstraktionen auf einem hohen Niveau ausdrücken kann (Abschnitt 5.2.2). Danach wenden wir uns Anwendungsaspekten dieser Erweiterung auf Spezifikationsebene (Abschnitt 5.3) und Implementierungsaspekten (Abschnitt 5.4) sowie der Frage der Übertragbarkeit auf andere formale Techniken (Abschnitt 5.5) zu.

5.2.1 Typabstraktion in Standard-Estelle

Wie wir oben bereits gesehen haben, ist Standard-Estelle [ISO97] (wie auch die Erweiterung *Open-Estelle*, siehe Kapitel 7) eine stark getypte Sprache. Dies bedeutet, dass die Zuweisungskompatibilität von Datentypen strengen Restriktionen unterliegt (siehe Abschnitt 5.1.2). Es bleibt also die Frage, inwieweit bzw. ob überhaupt mit den bereits vorhandenen Ausdrucksmitteln von Standard-Estelle die oben geforderte Datentypabstraktion an der Dienstschnittstelle möglich ist. Ziel dieser Abstraktion ist es, einem Dienstnutzer zu ermöglichen, beliebige Datenstrukturen auf den selben SDU-Typ abzubilden, der dann vom Diensterbringer weiter transportiert wird. Ebenso soll die Abbildung in die umgekehrte Richtung möglich sein.

Wie wir oben bereits gesehen haben, gibt es in Estelle keinen Typ, der mit einem beliebigen Typ zuweisungskompatibel ist. Ein nahe liegender Ansatz wäre es also, eine unvollständige Typdefinition einzusetzen. Dies wird in Estelle mit dem folgenden Konstrukt ausgedrückt:

```
TYPE T = ...;
```

Eine solche unvollständige Typdefinition ist syntaktisch korrekt, sie macht jedoch die Spezifikation, in der sie enthalten ist, zu einer *unvollständigen* Spezifikation („not well-formed“, Abschnitt 8.2.3.3 in [ISO97]), welcher keine Semantik zugeordnet ist. Die Zielsetzung einer solchen unvollständigen Typdefinition ist es, bei der Erstellung einer Spezifikation zunächst die Details eines Typs offen zu lassen, dabei jedoch eine syntaktische Prüfung zu ermöglichen. Im Rahmen einer schrittweisen Verfeinerung werden diese unvollständigen Definitionen dann sukzessive durch vollständige Typdefinitionen ersetzt. Erst wenn alle diese Ersetzungen stattgefunden haben, gewinnt die dann *vollständig* gewordene Spezifikation eine formale Semantik. Da also zur Erlangung einer formalen Systemspezifikation eine Ersetzung aller unvollständigen Definitionen notwendig ist, führt dieser Ansatz zu keiner praktikablen Lösung der Problematik.

Ein effektiverer Ansatz wird im Anhang des Estelle-Standards (Annex B von [ISO89]) genannt. Hier werden die strukturierten Datentypen des Dienstnutzers durch explizit spezifizierte Encoding- und Decoding-Prozeduren zum Zwecke des Datentransports durch den Dienst-erbringer in Byte-Arrays umgewandelt:

- Dienstschnittstelle des Diensterbringers (Auszug):

```
TYPE SDU = ARRAY [1 .. C] OF 0..255;
```

- Interne Definition des Dienstnutzers (Auszug):

```
PROCEDURE encode ( VAR out: SDU; in: usertype );
     PRIMITIVE;
PROCEDURE decode ( VAR in: SDU; out: usertype );
     PRIMITIVE;
```

Um ein Datenpaket vom Typ `usertype` an den Diensterbringer übergeben zu können, muss der Dienstnutzer dieses durch einen expliziten Aufruf der Prozedur `encode` in ein Datenobjekt des Typs `SDU` umwandeln. In Gegenrichtung ruft der Dienstnutzer die Funktionen `decode` zur Rückumwandlung des Datenobjekts auf. Die Konstante `C` ist dabei wie auch die Definition des Byte-Arrays `SDU` Bestandteil der Dienstschnittstelle des Diensterbringers. Die Encode- und Decode-Funktionen werden dagegen vom Dienstnutzer definiert. Sie sind als `PRIMITIVE`-Prozeduren gekennzeichnet, wodurch die umgebende Spezifikation nur dann semantisch vollständig und damit formal ist, wenn diesen beiden Funktionen eine mathematisch präzise Semantik zugeordnet werden kann.

Setzt man eine solche mathematisch präzise Semantik für die beiden Funktionen und einen konkreten Wert für `C` voraus, so ist diese Methode geeignet, eine formale Spezifikation einer Hierarchie von Diensten zu liefern, in der die oben beschriebenen Probleme der Verschachtelung von Typstrukturen nicht auftreten. Die Methode hat aber auch erhebliche Nachteile:

1. Bei der Definition der Konstanten `C` im Rahmen der Spezifikation der Dienstschnittstelle des Diensterbringers muss zur Erlangung einer formalen Semantik ein konkreter Zahlenwert angegeben werden. Dieser Zahlenwert beschränkt gleichzeitig die maximale Größe der binären Repräsentation der von späteren Dienstnutzern übertragbaren Datentypen. Zwar kann für `C` in einem überschaubaren Kontext mit bekannten Dienstnutzern und Typimplementierungen ein hinreichend großer Wert gefunden werden, der es erlaubt, die binäre Repräsentation aller Datentypen aufzunehmen. Es kann auf diese Art jedoch kein *universeller* Datentransportdienst definiert werden, da zu jedem endlichen Wert für `C` ein Typ gefunden werden kann, der nicht in dem oben definierten Array einkodiert werden kann.[7]
2. Die Darstellung der SDU-Typen als Byte-Array ist nur dann angemessen, wenn vom jeweiligen Dienst auch tatsächlich Zugriffe auf Byte-Ebene erfolgen. Dies liegt jedoch meist weit unterhalb des Niveaus, das als die eigentliche Domäne formaler Beschreibungstechniken anzusehen ist. FDTs sollen gerade ein Ausdrucksniveau bereitstellen, das den Spezifizierer durch hohe Abstraktion von den Details der Implementierungen befreit und stattdessen die Korrektheit der beschriebenen Funktionalität in den Mittelpunkt des Interesses stellt. Viele der mit formalen Beschreibungstechniken beschriebenen Dienste realisieren Datentransportdienste durch reine SDU-Aggregation, ohne auf byteorientierte Fragmentierung der Pakete oder gar weiter gehende Datenmanipulationen zurückzugreifen. Speziell letztere sind mit den Ausdrucksmitteln von Estelle oft auch nur sehr bedingt sinnvoll zu beschreiben. Als Beispiele seien Datenkompression, kryptografische Kodierung oder komplexe, Bit-orientierte Prüfsummenberechnungen genannt. Entsprechend ist in den Fällen, in denen gar kein Zugriff auf die Byte-Repräsentation der Daten benötigt wird, ein höheres Abstraktionsniveau wünschenswert.
3. Aber auch aus implementierungsorientierter Sicht ist die beschriebene Repräsentation inadäquat. So enthält das Byte-Array `SDU` keine expliziten oder impliziten Angaben, wie viele der Bytes zur Kodierung des jeweiligen Nutzdatentyps belegt wurden. Bei einem Dienst,

7. Man kann leicht zeigen, dass z. B. die Menge aller Werte des Typs `ARRAY [1..D] OF 0..255` mit $D > C$ mit keiner Abbildung reversibel in einer Variable des Typs `ARRAY [1..C] OF 0..255` kodiert werden kann.

dem von heterogenen Dienstnutzern unterschiedliche Nutzdatentypen in kodierter Form als Byte-Array übergeben werden, hängt die Anzahl der genutzten Bytes in diesem Byte-Array vom jeweiligen Nutzdatentyp und seiner Kodierung (und damit dem jeweiligen Dienstnutzer) ab. Bei einem universellen Einsatz eines solchen Dienstes enthalten die übergebenen Byte-Arrays folglich einen variablen Anteil an sinnvoll genutzten Daten. Wie viele das sind, ist jedoch nicht nahe liegend anhand des `SDU`-Typs in der obigen Form erkennbar. Somit muss bei der weiteren Verarbeitung unterhalb der Dienstschnittstelle immer davon ausgegangen werden, dass alle Bytes des Byte-Arrays mit Daten belegt sind und daher auch vom Dienst übertragen werden müssen. Dies bewirkt gerade bei großzügiger Auslegung der Größe des Arrays (s. o.) einen erheblichen Overhead bei der Datenübertragung.[8]

4. Bei der automatischen Generierung von Implementierungen aus solchen Spezifikationen ergibt sich neben dem vorgenannten Overhead auch noch die Problematik, dass beim Aufruf der Kodierungs- und Dekodierungsfunktionen beim Dienstnutzer jeweils ein zusätzlicher Kopiervorgang im praktischen Einsatz nahezu unvermeidbar ist (siehe Kapitel 6).

Es wird deutlich, dass die im Estelle-Standard vorgesehene Methode zur Kodierung von Benutzerdaten einige Nachteile mit sich bringt. In den nächsten Abschnitten betrachten wir zwei Estelle-Erweiterungen, die auf Spezifikationsebene eleganter und auf Implementierungsebene effizienter als die vorgenannten Lösungsansätze sind.

5.2.2 Estelle-Erweiterung „Containertyp“

Als Lösung für die beschriebene Problemstellung führen wir nun das Konzept des „*Containertyps*“ (engl.: „*Container Type*“) und seine Umsetzung als Estelle-Erweiterung ein [The03].

Wie wir bereits in Abschnitt 5.1.2 gesehen haben, gibt es zu manchen Typen `T` jeweils einen Typ `TX`, sodass ein beliebiger Wert vom Typ `T` durch eine Konvertierung in den Typ `TX` und anschließende Rückkonvertierung in den Ausgangstyp `T` nicht verändert wird, d.h. die Konvertierung von `T` nach `TX` also *vollständig reversibel* ist. Diese Beziehung gilt z. B. für den Teilbereichstyp `T=0..9` und den vordefinierten Typ `TX=INTEGER`. Einen solchen Typ `TX` bezeichnen wir als *Containertyp* für den Typ `T`, da eine Instanz des Typs `TX` einen beliebigen Wert des Typs `T` im Sinne eines Behältnisses unverändert „aufbewahren“ kann.

8. In der aktualisierten Fassung [ISO97] des Estelle-Standards wird eine nahe liegende Erweiterung der Datenstruktur zur Beschreibung der Bytesequenz um ein *Integer-Feld* zur Angabe der *Anzahl der (relevant) belegten Bytes* des Byte-Arrays eingesetzt. Dies ermöglicht eine Optimierung bei explizit spezifizierten (oder in primitiven Funktionen bzw. Prozeduren enthaltenen) Operationen auf den einzelnen Bytes des Arrays, indem nur die belegten Elemente bearbeitet werden.

   Auf die Effizienz der Handhabung des Datenobjekts als Ganzes (z. B. bei expliziten Kopieroperationen oder Übertragung als Interaktionsparameter) hat dies jedoch keinen positiven Effekt, solange bei der Implementierung der semantische Zusammenhang nicht explizit berücksichtigt wird. Dies ist bei automatisch generierten Implementierungen nur durch entsprechende Anpassungen des Implementierungssystems zur Interpretation des Feldes möglich.

Definition 5.2: Containertyp

Ein Typ `TX` *heißt* ***Containertyp*** *zu einem Typ* `T`

genau dann wenn

`TX` *und* `T` *in beide Richtungen zueinander zuweisungskompatible Typen sind und für alle Werte* `x` *vom Typ* `T` *gilt: Konvertiert man den Wert* `x` *vom Typ* `T` *in den Typ* `TX` *und anschließend wieder in den Typ* `T`*, so sind beide Konvertierungen zulässig, und der sich aus der Folge der beiden Konvertierungen ergebende Wert ist identisch mit dem Ausgangswert* `x`*.*

(Ende von Definition 5.2)

Gemäß dieser Definition ist zum Beispiel in Estelle der Typ `TX=INTEGER` ein Containertyp für den `INTEGER`-Teilbereichstyp `T=0..9`, da die Wertemenge von `T` als Teilmenge der Wertemenge von `TX` in diese eingebettet ist und daher die Typkonvertierungsfunktionen von `T` nach `TX` und zurück beschränkt auf die Wertemenge von `T` gerade die identische Abbildung ist. Umgekehrt ist der Typ `T` jedoch kein Containertyp für `TX`, da die Konvertierung des `TX`-Wertes 10 nach `T` unzulässig ist (siehe Abschnitt 6.4.6, Annex C von [ISO97]).

Wie man an diesen Beispielen bereits sieht, ist die Teilmengenbeziehung $val(T) \subseteq val(TX)$ zwischen den Wertemengen der Typen `TX` und `T` zusammen mit der identischen Abbildung als Typumwandlungsfunktion eine hinreichende Bedingung für die Erfüllung der *ist-Containertyp-für* Relation zwischen `TX` und `T`.[9] Eine solche Bedingung ist jedoch restriktiver als die oben beschriebene: Def. 5.2 setzt ausschließlich die Umkehrbarkeit der Typumwandlung vom Typ `T` in den Containertyp `TX` voraus, ohne Beschränkungen der Art der Umwandlung zu machen. So könnte z. B. die Klasse der Zeichenketten unbegrenzter Länge als Containertyp für die Klasse der ganzen Zahlen dienen, wenn bei der Umwandlung die Zahlen bei der Konvertierung in Strings durch ihre Dezimal- oder Hexadezimaldarstellung[10] repräsentiert werden und die Rückkonvertierung dies entsprechend invertiert, obwohl die Wertemengen beider Typen (Zeichenketten und ganze Zahlen) sogar disjunkt sind.

Wie man im vorangegangenen Beispiel deutlich sieht, spielt bei diesen Überlegungen die Frage nach der Konvertierung von Typen und der Zuweisungskompatibilität eine wichtige Rolle. Wir werden im Zusammenhang mit der Implementierbarkeit von Containertypen (siehe Abschnitt 5.4) nochmals auf diese Fragestellung zurückkommen.

Offensichtlich existiert zu jedem Typ T zumindest ein (trivialer) Containertyp, nämlich der Typ `T` selbst. Andererseits haben wir bereits in Abschnitt 5.1.2 gesehen, dass es in Estelle keinen *universellen* Containertyp gibt, also einen Typ, der für jeden beliebigen Typ zugleich Containertyp ist. Ziel der vorgestellten Estelle-Erweiterung ist es nun, einen solchen universellen Containertyp bereitzustellen, um diesen als SDU-Typ für die formale Definition der Dienstschnittstelle eines universellen Datentransportdienstes zu verwenden.[11]

Syntaktisch beruht diese Estelle-Erweiterung auf der Einführung eines neuen[12] TYPE-DENOTER (siehe Abschnitt 6.4.1, Annex C von [ISO97]):

---

9. Neben einfachen Teilbereichstypen ist die genannte Teilmengenbeziehung zwischen den Wertemengen der Typen bei geeigneter Abstraktion auch für Klassen und ihre Unterklassen in objektorientierten Typsystemen denkbar. Wir kommen auf diesen Aspekt in Abschnitt 5.5 zurück.

10. griech. *hexa* „*sechs*“, lat. *decem* „*zehn*“, auch *Sedezimalsystem* von lat. *sedecim* „*sechzehn*“ [Wik04]

11. Der mit der Estelle-Erweiterung eingeführte Typ ist, wie später gezeigt wird, kein wirklich *universeller* Containertyp; er erfüllt diese Anforderungen jedoch für alle in Estelle zur Beschreibung von Dienstschnittstellen nutzbaren (also alle *relevanten*) Datentypen (siehe unten).

Definition 5.3: Syntax der Containertyp-Erweiterung

```
TYPE-DENOTER = - | "ANY" "TYPE" .
```

(Ende von Definition 5.3)

Durch diese Erweiterungen kann der Ausdruck[13] „**ANY TYPE**“ ähnlich wie der Bezeichner eines vordefinierten Datentyps zur Deklaration bzw. Definition u. a. von Typen, Variablen, Prozedurparametern und nicht zuletzt Interaktionsparametern verwendet werden. Letzteres war gerade die Motivation für die Einführung dieser Erweiterung.

Beispiel 5.35: Syntaktischer Gebrauch von „ANY TYPE“

```
TYPE T = ANY TYPE;
VAR x: ANY TYPE;
PROCEDURE f(x: ANY TYPE); BEGIN (* ... *) END;
CHANNEL ChService_N(User,Provider);
     BY User:     D_Send(Data: ANY TYPE);
     BY Provider: D_Recv(Data: ANY TYPE);
```

(Ende von Beispiel 5.35)

Die Bezeichnung „**ANY TYPE**“ lässt bereits vermuten, dass der damit bezeichnete vordefinierte Typ[14] *any-type* bezüglich der Kompatibilität mit anderen Typen besondere Eigenschaften hat. Diese werden durch die Ergänzung der Definition der Zuweisungskompatibilität („assignment-compatibility“, Abschnitt 6.4.6, Annex C von [ISO97]) erreicht:

12. Die Schreibweise mit „- |“ entspricht der in Abschnitt 8.1 des Estelle-Standards [ISO97] eingeführten Notation zur Bezugnahme auf die rechte Seite der bisherigen (hier Standard-Estelle-) Definition.

13. Es ist zu beachten, dass Def. 5.3 auf der Sequenz der beiden Terminale (Token) „**ANY**“ und „**TYPE**“ und nicht auf dem (zur Vereinfachung im Text gelegentlich verwendeten) geschlossenen Terminal „**ANY TYPE**“ basiert. Entsprechend kann zwischen den beiden Schlüsselworten jede nicht-leere Folge von Token-Trennzeichen stehen (siehe Abschnitt 6.1.8 des Estelle-Standards [ISO97]) anstatt nur genau eines Leerzeichens, wie es bei „**ANY TYPE**“ der Fall wäre.
Folglich entspricht z. B. das Spezifikationstextfragment „**ANY** *(* useful *)* **TYPE**“ Def. 5.3.

14. Der neue vordefinierte Typ *any-type* wird (völlig analog z. B. zu *integer*) durch den TYPE-DENOTER „**ANY TYPE**“ lediglich *referenziert*, d.h. im Sinne der „*single point of definition*“-Regel sind die daraus abgeleiteten Typreferenzen alle identisch definiert und daher bereits auf Basis der Standard-Estelle-Zuweisungskompatibilitätsregeln („assignment-compatibility“, Abschnitt 6.4.6, Annex C von [ISO97]) zueinander kompatibel.

Definition 5.4: Erweiterte Typkompatibilität für die Containertyp-Erweiterung von Estelle

*A value of type T2 shall be designated **assignment-compatible** with a type T1 if any of the following seven statements is true:*

a .. e) *{unmodified}*
f) *T1 is the any-type and T2 is not pointer-containing.*
g) *T1 is not pointer-containing and T2 is the any-type.*

At any place where the rule of assignment-compatibility is used

a, b) *{unmodified}*
c) *it shall be an error, if T1 is the any-type and the value of T1 was not created by conversion from type T2 to the any-type.*

{ rest unmodified }

(Ende von Definition 5.4)

Der oben genannte Punkt (f) bewirkt, dass Zuweisungen auf den *any-type* nicht nur von Werten des selben Typs, sondern von jedem beliebigen Typ syntaktisch zulässig sind, solange dieser Typ nicht *pointer-containing* ist, d.h. `T2` weder selbst ein Pointer ist, noch direkt oder indirekt einen Pointer enthält (siehe Abschnitt 7.3.4.2 aus [ISO97]). Diese Einschränkung ist notwendig, damit der *any-type* angesichts seiner möglichen Nutzung als Containertyp nach einer Zuweisung von Werten anderer Typen selbst niemals Pointer-Werte enthält.

Die Beschränkungen der zu dem *any-type* zuweisungskompatiblen Typen auf solche Typen, die nicht pointer-containing sind, bewirkt leider auch, dass der *any-type* kein wirklich universeller Containertyp ist. Diese Problematik ist jedoch vor dem Hintergrund der Motivation für die Einführung dieser Estelle-Erweiterung zu sehen: Der *any-type* soll zur Spezifikation von abstrakten Datentransportdiensten eingesetzt werden, indem er als Typ der SDU-Parameter von Datenübertragungsinteraktionen eine Abstraktion von den konkreten Typen der von den Dienstnutzern übergebenen Daten leistet. In Estelle dürfen Interaktionsparameter nicht pointer-containing sein, da Pointer nur in dem lokalen Interpretationskontext derjenigen Modulinstanz, in der sie (z. B. durch Aufruf von `new`) erzeugt wurden, gültig sind. Die Übertragung eines Pointerwertes von einer Modulinstanz zu einer anderen wäre damit konzeptionell unerwünscht. Der Ausschluss von pointer-containing Typen bei der Zuweisungskompatibilität auf den *any-type* stellt die Einhaltung dieses Grundprinzips sicher.

Der Punkt (g) in der obigen Definition erlaubt analog die Zuweisung von einem *any-type* auf einen beliebigen anderen Typ, solange dieser wiederum nicht pointer-containing ist.

Die bisher erläuterten Ergänzungen der Syntax von Estelle ermöglichen es bereits, die Dienstschnittstelle eines abstrakten Datentransportdienstes syntaktisch formal zu beschreiben, indem als SDU-Typ (formaler Parameter der Datenübertragungsinteraktion) der *any-type* eingesetzt wird. Die Vorstellung dabei ist, dass ein Dienstnutzer bei der Übergabe eines zu versendenden Datenpaketes einen Wert eines beliebigen (nicht pointer-containing) Typs als aktuellen Parameter der Datenübertragungsinteraktion übergeben kann. Dieser wird dann vom Datenübertragungsdienst unverändert zur Empfängerinstanz übertragen und dort als Interaktionsparameter an den empfangenden Dienstnutzer übergeben, der wiederum dieses Datenobjekt per Zuweisung vom *any-type* in den ursprünglichen Typ zurück konvertieren kann. Da der *any-type* für die zuweisungskompatiblen (nicht pointer-containing) Typen ein *Containertyp* ist, wird auf diese Weise beim Empfänger der Wert des gesendeten Typs beim Empfänger unverändert wiederhergestellt (siehe Beispiel unten).

Offensichtlich ist, wie wir schon früher festgestellt haben, diese Aufweichung der Typsicherheit der formalen Beschreibungstechnik Estelle nur dann konzeptionell akzeptabel, wenn die Semantik der Typübergänge in einem mathematischen Sinne eindeutig beschrieben werden kann. Dies erreichen wir durch zwei Maßnahmen:

- Es gibt neben der Zuweisung und den oben genannten Konvertierungen keine Aktionen oder Operationen auf Instanzen des *any-type*s.[15]
- Erfolgt eine Konvertierung von einem beliebigen (zulässigen) Typ in den *any-type*, so kann der sich ergebende Wert vom Typ *any-type* ausschließlich in den Ausgangstyp zurück konvertiert werden.[16] Dies wird von Punkt (c) in der obigen Definition sichergestellt.[17]

Dadurch ist z. B. die letzte Zuweisung im folgenden Beispiel naheliegenderweise unzulässig:

Beispiel 5.36: Zulässigkeit von Typkonvertierungen mit Containtertyp

```
VAR  a,b: REAL;
     c:   ARRAY [0..9] OF CHAR;
     x:   ANY TYPE;
BEGIN
     a := 3.1415;
     x := a;       (* Ok: Typkonvertierung von REAL nach ANY TYPE *)
     b := x;       (* Ok: Typkonvertierung von ANY TYPE           *)
                   (*     (enthält Typ REAL) nach REAL            *)
     c := x;       (* Fehler: Typkonvertierung von ANY TYPE       *)
                   (*         (enthält Typ REAL) in anderen Typ   *)
END;
```

(Ende von Beispiel 5.36)

Die Aufgabe der korrekten Zuordnung des Konvertierungszieltyps bei der Rückkonvertierung eines *any-type*-Wertes ist in jedem Fall explizit auf Spezifikationsebene zu lösen: Es gibt auf Basis des *any-type*-Wertes selbst zunächst[18] keine Möglichkeit zur Ermittlung des enthaltenen Typs. Die zur korrekten Rückkonvertierung erforderlichen Informationen müssen *apriori* bekannt sein. Wir werden in Abschnitt 5.3 sehen, dass dies bezüglich der Paketformate bei praktisch vorkommenden Protokollen durchgängig der Fall ist.

Die Aufgabe der korrekten Typzuordnung ist dabei vergleichbar mit dem korrekten Zugriff auf die Komponenten eines varianten Records[19] oder die Dereferenzierung eines Pointers[20]: In all diesen Fällen müssen Anforderungen an die Zulässigkeit des Zugriffs auf Spezifikationsebene sichergestellt werden. Insbesondere ist eine effektive Überprüfung der Zulässigkeit einer Kon-

15. Wir werden später in Abschnitt 6.4.6 im Zusammenhang mit effizienten Implementierungen von Datenübertragungsoperationen nochmals auf die besondere Bedeutung dieses Punkts zurückkommen.

16. Durch die Einstufung einer unzulässigen Rückkonvertierung als „*error*“ (siehe Abschnitt 3.3 von [ISO97]) wird eine Spezifikation, die dies ermöglicht, *ungültig*. Alternativ wäre auch auf semantischer Ebene eine Einstufung des Ergebnisses einer solchen unzulässigen Operation als „*undefined*“ (siehe Abschnitt 3.2 von [ISO97]) denkbar.

17. Es ist zu beachten, dass bei der Konvertierung von *any-type* in den Zieltyp eine Typgleichheit und nicht nur eine Kompatibilität oder Zuweisungskompatibilität gefordert wird. Genaueres dazu folgt in Abschnitt 5.4.2.

18. Ein solcher Zugang kann auf Implementierungsebene (siehe auch Abschnitt 5.4) gegeben sein (z. B. zugreifbar über eine primitive Funktion), wird jedoch nicht vorausgesetzt.

vertierung vom *any-type* in einen anderen Typ allgemein aufgrund mangelnder Berechenbarkeit der Zustandsraumentwicklung nicht *statisch* (d.h. auf Spezifikationsebene), sondern nur für einen konkreten Zustandsübergang (d.h. *dynamisch*, z. B. auf Implementierungsebene zur Laufzeit) möglich. Wie eine solche Überprüfung und der *any-type* selbst technisch realisiert werden können ist Gegenstand von Abschnitt 5.4.

Die *Semantik* der vorgestellten Estelle-Erweiterung lässt sich auf Grund der syntaktischen Restriktionen direkt auf die oben genannte Definition des Containertyps (siehe Def. 5.2 auf Seite 197) zurückführen:

Definition 5.5: Semantik des *any-type*

The any-type is a container-type for all not-pointer-containing types.

(Ende von Definition 5.5)

Insbesondere ist zu beachten, dass eine Spezifikation durch den Einsatz des „**ANY TYPE**“-Konstrukts *nicht unvollständig* wird, wie es beim Einsatz einer Typdefinition mit „**...**“ oder einer Konstantendefinition mit „**ANY** <TYPE-DENOTER>„ der Fall wäre.

Ein weiterer wichtiger Aspekt ist, dass die vorgestellte Containertyp-Erweiterung orthogonal zu den Erweiterungen in Open-Estelle ist und daher beide problemlos kombiniert eingesetzt werden können (siehe Abschnitt 7.6).

Beispiel 5.37: Einsatz der Containertyp-Erweiterung

Zur Verdeutlichung des praktischen Einsatzes der vorgestellten Erweiterung betrachten wir das in Abb. 5-8 dargestellte Beispiel-Szenario eines abstrakten Datentransportdienstes, das sich aus einem Diensterbringer (siehe Beispiel 5.37-e) für einen abstrakten Datentransportdienst (siehe Beispiel 5.37-a) und zwei Instanzen eines Dienstnutzers (hier in zwei Varianten, siehe Beispiel 5.37-b bis 5.37-d) zusammensetzt. Der Diensterbringer hat zwei externe Interaktionspunkte, über die er eingehende Nachrichten von einem Interaktionspunkt immer zu dem anderen unverändert transportiert.[21]

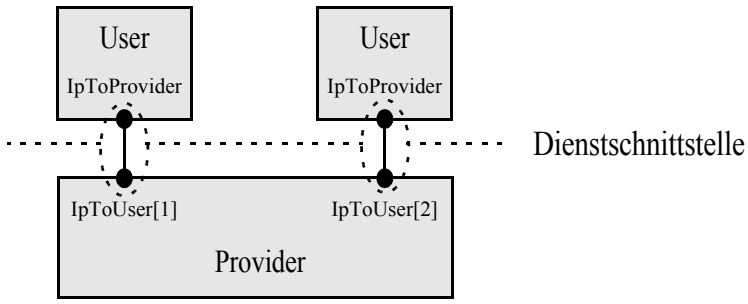


Abbildung 5-8: Modulstruktur des Datentransport-Szenarios

Der Ausgangspunkt des Beispiels ist die Definition der abstrakten Dienstschnittstelle:

19. Zu einem gegebenen Wert eines varianten Records (speziell wenn dieser keine Selektor-Komponente enthält) ist nur der Zugriff auf die aktive Variante des Records zulässig (siehe Abschnitt 6.5.3.3, Annex C von [ISO97]).

20. siehe Abschnitt 6.5.4, Annex C von [ISO97]

21. Zur besseren Übersichtlichkeit wurde in der folgenden Darstellung auf die Modulstruktur verzichtet, und es werden nur die wesentlichen Definitionen angegeben.

Beispiel 5.37-a: Abstrakte Dienstschnittstelle

```
TYPE Td = ANY TYPE;
CHANNEL ChService_N(User,Provider);
     BY User:     D_Send(Data: Td);
     BY Provider: D_Recv(Data: Td);
```

(Ende von Beispiel 5.37-a)

Diese Definition von Td erlaubt es dann dem Dienstnutzer, Werte des Typs T1 direkt und ohne weitere Konvertierungen als Nachrichtenparameter einzusetzen:

Beispiel 5.37-b: Dienstnutzer (Senderichtung)

```
IP IpToProvider: ChService_N(User);              (* modulexterner IP *)
(* ... *)

TYPE T1 = SET OF 0..7;

VAR x1: T1;
TRANS
     BEGIN
     OUTPUT IpToProvider.D_Send(x1);       (* any-type-Konvertierung *)
     END;
```

(Ende von Beispiel 5.37-b)

In Gegenrichtung wird der Dienstnutzer[22] beim Empfang der `D_Recv`-Interaktion das als Parameter übergebene Datenobjekt wieder in den vom Sender ursprünglich eingesetzten Datentyp umwandeln, um seinen Inhalt auswerten zu können:

Beispiel 5.37-c: Dienstnutzer (Empfangsrichtung)

```
TRANS
     WHEN IpToProvider.D_Recv(Data: Td)
          VAR x: T1;
          BEGIN
          x := Data;                    (* any-type-Rückkonvertierung *)
          (* werte x aus ... *)
          END;
```

(Ende von Beispiel 5.37-c)

Ein anderer Dienstnutzer kann dagegen einen völlig unterschiedlichen Datentyp als Parameter für den selben Interaktionstyp übergeben:

22. Hier wird der Einfachheit halber von einem symmetrischen Dienst ausgegangen, d.h. beide Dienstnutzer (Sender und Empfänger) sind Instanzen des selben Moduls. Die Situation ist aber leicht auf asymmetrische Konstellationen übertragbar.

Beispiel 5.37-d: Dienstnutzer (alternativer PDU-Typ)

```
TYPE T2 = RECORD
              a: REAL;
              b, c: BOOLEAN;
          END;

VAR x2: T2;
TRANS
     BEGIN
     OUTPUT IpToProvider.D_Send(x2);        (* any-type-Konvertierung *)
     END;

TRANS
     WHEN IpToProvider.D_Recv(Data: Td)
          VAR x: T2;
          BEGIN
          x := Data;                    (* any-type-Rückkonvertierung *)
          (* ... wertet x aus ... *)
          END;
```

(Ende von Beispiel 5.37-d)

Wesentlich ist dabei, dass beim Empfang einer solchen Interaktion die Typkonvertierung des „ANY TYPE“ Parameters in den korrekten Typ erfolgt. Mit der Problematik der Einhaltung dieser Anforderung in einer heterogenen SDU-Typ-Situation beschäftigen wir uns später nochmals in Abschnitt 5.3.1 im Zusammenhang mit Protokollmultiplexern.

Der Diensterbringer ist durch die Erweiterung völlig unabhängig von den konkreten SDU-Typen und transportiert den Containertyp entsprechend uninterpretiert[23] und hier auch ohne weitere Typumwandlung.[24]

Beispiel 5.37-e: Diensterbringer

```
IP IpToUser: ARRAY [1..2] OF ChService_N(Provider); (* Ext. IPs *)
(* ... *)

TRANS
     ANY i: 1..2 DO
          WHEN IpToUser[i].D_Send(Data)
               BEGIN
               OUTPUT IpToUser[3-i].D_Recv(Data);
               END;
```

(Ende von Beispiel 5.37-e)

(Ende von Beispiel 5.37)

23. Wir werden uns aber später in Abschnitt 5.3 nochmals mit erweiterten Zugriffen auf Containertypen beschäftigen.

24. Da die Dienstschnittstelle bereits abstrakt ist (als SDU-Typ also der *any-type* eingesetzt wird), erfolgen alle in diesem Zusammenhang erforderlichen Konvertierungen oberhalb der Dienstschnittstelle.

5.3. Anwendungsaspekte der Containertyp-Erweiterung auf Spezifikationsebene

Im vorangegangenen Abschnitt haben wir bereits ein erstes Beispiel für den praktischen Einsatz der vorgestellten Estelle-Erweiterung „Containertyp“ gesehen. Im Folgenden werden nun einige konzeptionelle Aspekte des Einsatzes der Containertyp-Erweiterung in komplexeren Szenarien diskutiert.

Eine der Kernfragen beim praktischen Einsatz der Containertyp-Erweiterung zur *formalen* Spezifikation von Systemen ist natürlich die Erhaltung der Typsicherheit, da diese – wie wir oben bereits gesehen haben – eine der wesentlichen Grundlagen für die Gewinnung einer eindeutigen Semantik der Spezifikation darstellt. Zweck der Typsicherheit ist es, der Konvertierung zwischen verschiedenen Typen eine im mathematischen Sinne eindeutige Semantik zu verleihen. Wir werden nun untersuchen, inwieweit die vorgestellte Erweiterung eine Beeinträchtigung der Typsicherheit von Estelle darstellt.

5.3.1 Typsicherheit bei heterogenen Typabstraktionen

Ein wichtiger Punkt zur Erhaltung der Typsicherheit bei der Konvertierung eines Wertes eines beliebigen Typs in den Containertyp *any-type* ist die Tatsache, dass auf diesem neuen Wert neben der Rückkonvertierung in den Ausgangstyp zunächst keine Operationen definiert sind. Diese Rückkonvertierung liefert aber gerade den Ausgangswert (im Ausgangstyp) und ist somit mathematisch wohldefiniert. Eine Rückkonvertierung in einen anderen Typ als den Ausgangstyp ist hingegen explizit verboten. Eine wichtige Frage ist also, wie die Einhaltung dieses Verbots sichergestellt werden kann.

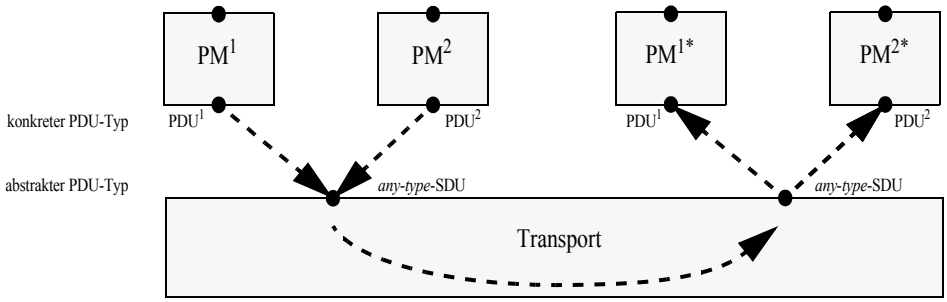


Abbildung 5-9: Heterogene Nutzung von abstrakten Datentransportdiensten

Dazu betrachten wir ausgehend von unserer ursprünglichen Motivation, der formalen Spezifikation einer heterogenen Diensthierarchie, das folgende Szenario: Zwei Protokollautomaten („PM1“ und „PM2“ in Abb. 5-9) mit unterschiedlichen PDU-Typen („PDU1“ und „PDU2“) kommunizieren über einen gemeinsamen abstrakten Datentransportdienst („Transport“) mit entsprechenden Partnerinstanzen. Dazu wird beim Verschicken die jeweilige von den Protokollautomaten übertragene PDU bei der Übergabe an diesen Diensterbringer als *any-type*-SDU abstrahiert. Diese abstrakte SDU wird von dem Datentransportdienst unverändert übertragen und muss dann schließlich der *korrekten* Protokollinstanz übergeben werden, wo sie dann per Typkonvertierung in den Ausgangstyp zurückgewandelt wird. Die korrekte Zuordnung ist dabei es-

sentiell, da eine Fehlzuordnung nicht nur potenziell zu einer Störung der Kommunikation zwischen den jeweiligen Partnerprotokollinstanzen führen könnte, sondern auch auf Grund der Typverschiedenheit der beiden PDU-Typen unzulässig wäre.

Betrachten wir zur Beantwortung der Frage, wie eine solche Fehlzuordnung vermieden werden kann, die Behandlung dieser Problematik in verbreiteten Kommunikationsprotokollen und deren Implementierungen am Beispiel der TCP/IP-Suite.

Das *Internet Protokoll* (IPv4, siehe [RFC791]) ist der Vermittlungsschicht-Datagrammdienst der TCP/IP-Suite. Alle anderen Protokolle der TCP/IP-Suite[25] nutzen IP, um Pakete von Host zu Host zu transportieren. Ein IP-Paket (siehe Abb. 5-10) enthält dabei neben den für das Routing notwendigen Informationen und der zu übertragenden SDU (Feld „`Data`“) u. a. noch ein 8-Bit Feld namens „`Protocol`“, in welchem festgehalten wird, welches Protokoll die im Paket enthaltene SDU erzeugt hat (siehe [RFC1700]).

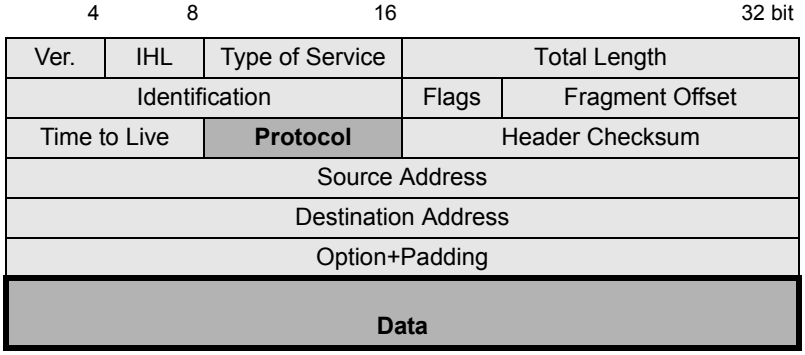


Abbildung 5-10: IPv4 PDU-Struktur

So wird z. B. bei der Erzeugung eines IP-Pakets, das als SDU eine TCP-PDU enthält (siehe [RFC793]), diese SDU in das `Data`-Feld des IP-Pakets gelegt und als Protokoll-ID der Wert 6 eingetragen (siehe Abb. 5-11). Nachdem dieses IP-Paket dann im Zielhost angekommen ist, wird anhand der enthaltenen Protokoll-ID die weitere Interpretation der enthaltenen SDU durchgeführt, indem diese an die TCP-Protokollmaschine weitergereicht wird, wo der Inhalt der SDU auf Grund der so gewonnenen Kenntnis der inneren Struktur als TCP-PDU erfolgen kann. Analog verläuft die Behandlung eines (zum TCP-Paket strukturverschiedenen) UDP-Pakets (siehe [RFC768]), wobei hier als Protokoll-ID der Wert 17 eingesetzt wird.

Diese Vorgehensweise ist ähnlich in vielen Protokollen anzutreffen, welche heterogene Datentypen als abstrakte (also zunächst Struktur-unbekannte) SDU-Objekte über einen gemeinsamen Datentransportkanal (im obigen Beispiel IP) transportieren: Neben den zunächst noch unstrukturierten Nutzdaten werden in einem Teil der PDU mit *apriori bekannter Struktur* auch Zusatzinformationen übertragen, wie das Datenobjekt zu interpretieren ist. In unserem Beispiel erfolgt die Interpretation dieser Zusatzinformation (im Feld `Protocol` des IP-Headers), indem diese zur Auswahl derjenigen Protokollmaschine eingesetzt wird, die die im IP-Paket enthaltene SDU (Feld `Data` des IP-Pakets) weiterverarbeitet. Bei fehlerfreier[26] Übertragung des IP-Pakets ist

25. mit Ausnahme von ARP und RARP

26. Bei verfälschten IP-Paketen ist durch eine Änderung des Feldes `Protocol` natürlich auch eine Fehlzuordnung der enthaltenen SDU zu einem inkompatiblen Protokollautomaten möglich. Diese Problematik ist jedoch analog zu einer unspezifischen Verfälschung des SDU-Inhalts zu betrachten, die im Allgemeinen ebenfalls keine sinnvollen Interpretationen der SDU-Struktur mehr zulässt. Entsprechend haben beide Fehlerklassen analoge Auswirkungen und werden hier nicht weiter betrachtet.

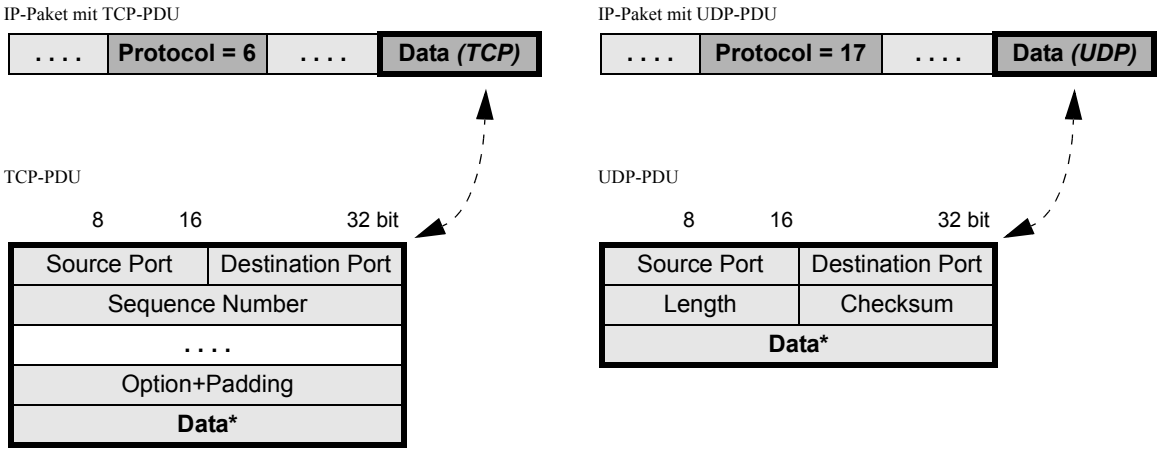


Abbildung 5-11: TCP- und UDP-PDUs eingebettet in IPv4-PDU

so sichergestellt, dass jederzeit eine korrekte Interpretation der Struktur der übertragenen Daten erfolgen kann, die Übertragung also *typsicher* erfolgt. Wir werden die Problematik der Dekomposition von PDUs nochmals in Abschnitt 5.3.2 aufgreifen.

Zur Demonstration der typsicheren Spezifikation eines Systems zur Übertragung verschiedener Typen über einen abstrakten Datentransportdienst entwickeln wir nun eine fragmentarisch dargestellte Estelle-Spezifikation unter Einsatz der Containertyp-Erweiterung.

Beispiel 5.38: Kanal- bzw. Protokoll-Multiplexer

Die anhand der TCP/IP-Suite vorgestellte Methode der typsicheren Übertragung unterschiedlicher SDU-Typen von verschiedenen Protokollmaschinen über einen gemeinsamen Datentransportkanal kann man als *reversibles Multiplexen* des Datentransportkanals betrachten. In Abb. 5-12 ist diese Funktionalität durch die Protokollmaschine `Mux` (Multiplex) modelliert.

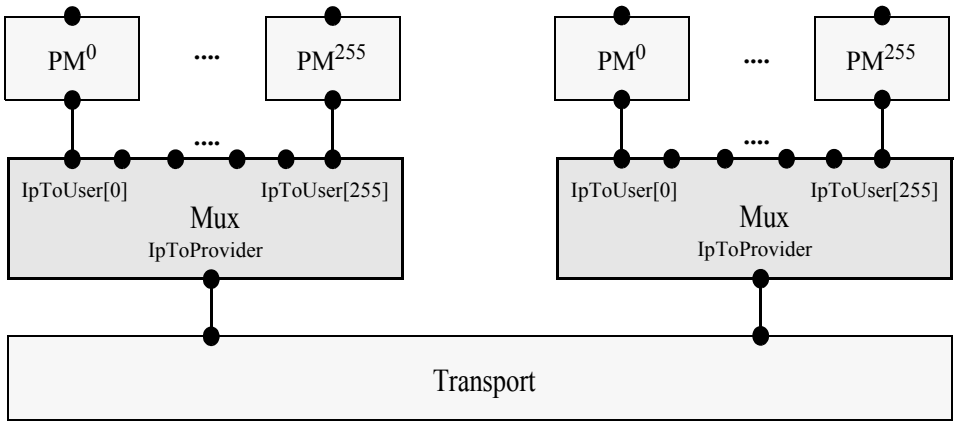


Abbildung 5-12: Anwendungsszenario eines Kanal- und Protokollmultiplexers

Der Multiplexer bietet an seiner oberen Schnittstelle ein Array von Interaktionspunkten, über die jeweils ein abstrakter Datenübertragungsdienst angeboten wird. Dieser Dienst sei der folgende: Trifft ein Sendeauftrag an einem der Interaktionspunkte im Interaktionspunkt-Array (beliebiger Index $n \in \{0, \ldots, 255\}$) einer Multiplexer-Instanz ein, so soll die dort übergebene SDU irgendwann unverändert am Interaktionspunkt mit dem gleichen Index (n) im Interaktionspunkt-Array der entsprechenden Partner-Instanz des Multiplexers als Empfangsereignis

übertragen werden. So werden über den zu Grunde liegenden Datenkanal des unteren Diensterbringers 256 getrennte Kanäle übertragen. Da diese jeweils einen abstrakten Datentransportdienst realisieren, können ohne weiteres unterschiedliche SDU-Typen über die verschiedenen Kanäle übermittelt werden, um z. B. die bis zu 256 verschiedenen durch das (8 Bit lange) `Protocol`-Feld der IP-PDU unterscheidbaren Protokolle isoliert voneinander über IP zu transportieren und schließlich den korrekten Protokollmaschinen und damit auch PDU-Typen zuzuordnen. So könnte der Index des jeweiligen Interaktionspunkts im Interaktionspunkt-Array die Bedeutung einer Protokoll-ID einnehmen, die TCP-Protokollmaschine also am Interaktionspunkt mit dem Index 6 und die UDP-Protokollmaschine am Interaktionspunkt 17 angekoppelt werden.

An der unteren Schnittstelle setzt der Multiplexer auf einem abstrakten Datenübertragungsdienst auf, der beliebige SDU-Typen unverändert transportiert. Einer naheliegenden Spezifikation des Multiplexers in Estelle mit der Containertyp-Erweiterung könnte also folgende (auszugsweise angegebene) Schnittstellendefinition zu Grunde liegen:

```
CHANNEL ChAbstractDataTransport(User,Provider);
     BY User:      D_Send(Data: ANY TYPE);
     BY Provider:  D_Recv(Data: ANY TYPE);

IP IpToUser: ARRAY [0..255] OF ChAbstractDataTransport(Provider);
IP IpToProvider: ChAbstractDataTransport(User);
```

Besonders bemerkenswert an dieser Schnittstellendefinition ist die Tatsache, dass die Protokollmaschine dank der Containertyp-Erweiterung nach „oben“(d.h. zu ihren Dienstnutzern über `IpToUser`) die selbe Kanaldefinition nutzt und damit *syntaktisch den selben Dienst*[27] bietet, wie er auch von der darunter liegenden Dienstschicht geboten wird, der darunter liegende Dienst also gewissermaßen *virtualisiert* wird. Dies ermöglicht z. B. auch die vertikale Kaskadierung mehrerer Instanzen dieses Protokolls, z. B. um eine noch größere Zahl von gemultiplexten Verbindungen zu erhalten (z. B. 2^{16} Kanäle bei zwei Multiplexer-Stufen à 2^{8} Kanälen). Für den Dienstnutzer eines einzelnen Kanals bleibt das Vorhandensein des Multiplexers (oder auch einer Kaskade von ihnen) syntaktisch wie auch semantisch transparent.

Das nächste Spezifikationsfragment zeigt die interne Definition des Multiplexer-Moduls mit der o. g. externen Schnittstelle. Die erste Transition nimmt Pakete vom Dienstnutzer entgegen und packt diese in eine PDU des Multiplexer-Moduls (`T_MPX_PDU`) ein. Die zweite Transition behandelt den Rücktransport zum Dienstnutzer und interpretiert den Struktur-bekannten Teil der eigenen PDU, um eine korrekte Zuordnung der Struktur-unbekannten SDU des jeweiligen Dienstnutzers zu diesem zu erreichen, indem sie die SDU unverändert an den entsprechenden Interaktionspunkt (`IpToUser[x.Protocol]`) weiterreicht.

```
TYPE T_MPX_PDU = RECORD
                   Protocol: 0..255;
                   Data: ANY TYPE;
                   END;

TRANS (* Multiplex User Send Requests *)
     ANY i: 0..255 DO
          WHEN IpToUser[i].D_Send(Data: Td)
               VAR x: T_MPX_PDU;
```

27. wenn auch in mehrfacher Instanziierung durch das Interaktionspunktarray

```
                    BEGIN
                    x.Protocol := i;
                    x.Data := Data;
                    OUTPUT IpToProvider.D_Send(x);
                                              (* any-type-Konvertierung *)
                    END;

TRANS (* Demultiplex Packets from Net *)
     WHEN IpToProvider.D_Recv(Data: Td)
          VAR x: T_MPX_PDU;
          BEGIN
          x := Data;                    (* any-type-Rückkonvertierung *)
          OUTPUT IpToUser[x.Protocol].D_Recv(x.Data);
          END;
```

Ebenfalls bemerkenswert ist hier, dass der PDU-Typ durch die Containertyp-Erweiterung als interne und damit *private* Definition des Protokollmaschinen-Moduls verfasst werden konnte, nach außen also weder als Teil der externen Schnittstelle noch anderweitig sichtbar wird.

(Ende von Beispiel 5.38)

Offensichtlich gelingt durch die gezeigte Vorgehensweise auch bei der Rückkonvertierung heterogener SDU-Typen eine eindeutige und damit typsichere Zuordnung, indem zusammen mit den abstrakten SDUs immer auch Informationen übertragen werden, die für die Typzuordnung notwendig sind (z. B. das Feld `Protocol` in den vorangegangenen Beispielen). Dadurch ist eine Rückkonvertierung des *any-type*s in den korrekten Ausgangstyp auf der Basis von *apriori bekannten Informationen* möglich. Wir werden uns im nächsten Abschnitt mit einer Verallgemeinerung und Dynamisierung dieser Idee befassen.

5.3.2 Sukzessive Datenkomposition und -Dekomposition

In den vorangegangenen Beispielen wurde die Containertyp-Erweiterung dazu eingesetzt, ein Datenobjekt beliebiger Struktur und Größe als *any-type*-Objekt zu abstrahieren, um es dann als Interaktions-Parameter oder Komponente einer PDU weiterzutransportieren und schließlich wieder in den Ausgangstyp zurückzukonvertieren. Dabei realisiert die Einbettung eines *any-type*-Objekts in eine ansonsten Struktur-bekannte PDU gerade den Vorgang des „Framings“, wie es z. B. auf Bytekodierungsebene in Abb. 5-4 auf Seite 186 dargestellt ist.

Die bisherigen Beispiele hatten dabei gemeinsam, dass die PDU, in die das *any-type*-Objekt eingebettet wurde, ansonsten eine feste Struktur hatte (siehe z. B. IPv4-Paketformat in Abb. 5-10). Eine solche Konstellation mit *statischer PDU-Aggregation* ist gerade in älteren Datenübertragungsprotokollen häufig vorzufinden und bietet den Vorteil, dass das Framing sowohl in der formalen Spezifikation als auch in der Implementierung besonders einfach dargestellt werden kann. Ein Nachteil ist jedoch die eingeschränkte Flexibilität dieser Vorgehensweise. So müssen *Seitenband-Daten* („sideband traffic“), also Daten, die in irgendeiner Form „parallel“ zum eigentlichen Strom der oben genannten Datenobjekte übertragen werden, entweder einen fest reservierten Platz in der PDU finden oder als *any-type*-Objekte in separaten PDU-Instanzen transportiert werden. Gerade die erste Lösung ist auf Grund der festen Reservierung eines Datenbereichs der PDU immer dann besonders ineffektiv, wenn die Größe der Seitenband-Daten pro Pa-

ket stark schwankt oder gar nur in einem Teil der Pakete ein solches Datum zu übertragen ist, da bei einer statischen Lösung immer der maximal zu übertragende Datenbereich in jeder PDU reserviert werden muss.

Moderne Datenübertragungsprotokolle bieten dagegen häufig eine flexible Unterstützung von solchen Seitenband-Daten sowohl für protokollinterne Aspekte (wie Routing-, Fehler- und Verkehrs-Informationen oder allgemeine Quality-of-Service-Parameter) als auch für die Bereitstellung eines Dienstes zum Transport von Seitenband-Daten z. B. für Multimedia-Applikationen. Dazu genügt es meist nicht mehr, eine Struktur-unbekannte SDU in eine ansonsten fest strukturierte PDU einzubetten, sondern die PDU setzt sich aus einer Folge varianter Felder zusammen, die häufig als *Segmente* bezeichnet werden. Als Beispiele betrachten wir dazu die Protokolle IPv6 und XTP 4.0.

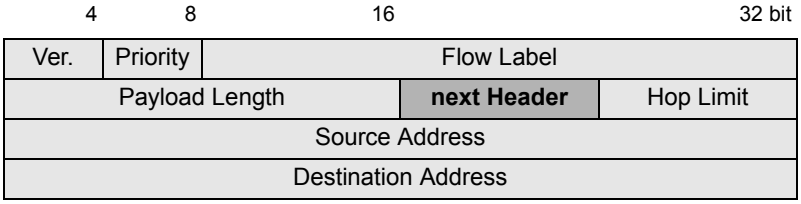


Abbildung 5-13: IPv6 PDU-Basis-Header

Das *Internet Protokoll Version 6* (IPv6, [RFC1883], [RFC2460]) bietet als Nachfolger des oben bereits beschriebenen IPv4 die Möglichkeit, zwischen dem IPv6-Paket-Basis-Header (siehe Abb. 5-13) und der im Paket transportierten eigentlichen Nutzlast (z. B. ein TCP-Paket mit entsprechendem Header- und Nutzdatenanteil) eine Folge optionaler Segmente einzubetten:

- Hop-by-Hop Options header
- Destination Options header
- Routing header
- Fragment header
- Authentication header
- Encapsulating Security Payload header
- Destination Options header

Jeder dieser Header enthält – wie auch der IPv6-Paketheader selbst – ein Feld namens „`next Header`“, in dem der Typ des jeweils folgenden Headers einkodiert ist (siehe Abb. 5-14). Dadurch kann die Protokollmaschine, die das IPv6-Paket dekodiert, ausgehend von dem immer am Beginn eines IPv6-Paket stehenden IPv6-Paketheader *schrittweise* den Typ und damit die Struktur des nächsten Headers ermitteln um so wiederum den Beginn und den Typ des darauf folgenden Headers zu bestimmen.

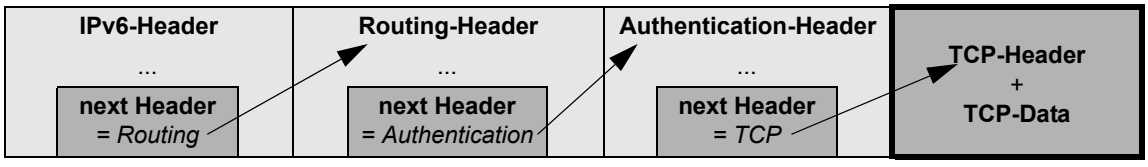


Abbildung 5-14: TCP-PDU und verschiedene Segmente der IPv6-PDU

Diese *sukzessive Typdekomposition* ist charakteristisch für Protokolle mit varianten PDU-Strukturen: Ausgehend vom Struktur-bekannten Teil des Pakets wird der Typ und damit die Struktur einer Komponente im (noch) Struktur-unbekannten Teil bestimmt und so schrittweise die Struktur des gesamten Pakets ermittelt. Eine ähnliche Konstellation finden wir im Aufbau

eines XTP 4.0 Pakets [XTP95] wieder, das aufbauend auf einem fest formatierten XTP-Header verschiedene Varianten von Strukturen enthalten kann, die zum Teil optional (z. B. „`Data Segment`“ in einem „`FIRST`“-Paket) oder sogar mit variabler Wiederholung (z. B. „`Spans`“ in einem „`ECNTL`“-Paket) auftreten können. Auch hier kann die Struktur des Pakets ausgehend vom Paketheader schrittweise interpretiert werden.

Offensichtlich ist die Behandlung derartig strukturierter Pakete von einer wesentlich höheren Dynamik geprägt, als dies in den einfachen Framing- und Unframing-Szenarien der früheren Beispiele der Fall war. Insbesondere ist die Dynamik der daraus resultierenden Typaggregation mit den Mitteln von Standard-Estelle nicht angemessen darstellbar (siehe Abschnitt 5.1). Zur Lösung dieser Problemstellung entwickeln wir nun ein *Spezifikationsmuster* auf der Basis der Containertyp-Erweiterung, mit der beliebig strukturierte Datenobjekte dynamisch komponiert und später unverändert dekomponiert werden können.

Die Grundidee der *dynamischen Datentypkomposition* besteht darin, ein *any-type*-Objekt schrittweise durch Hinzufügen von Daten zu erweitern, um so zur Laufzeit eine beliebige Aggregation von Datentypen zu erreichen. So kann (wie in Abb. 5-15 dargestellt) ausgehend von einem (Segment-) Header durch Hinzufügen von weiteren Headern schließlich die gewünschte Aggregationsstruktur erzeugt werden. Wie wir oben bereits gesehen haben, ist es dabei zum Zwecke einer späteren schrittweisen Dekomposition wichtig, dass die Struktur eines Segments jeweils apriori bekannt oder durch die jeweils vorangehenden Datenobjekte determiniert ist.[28] Dies wird z. B. beim IPv6-Paket, wie wir oben gesehen haben, durch den zuvorderst angeordneten, Struktur-bekannten IPv6-Header bzw. das in jedem Teil-Header enthaltene Feld „`next Header`“ erreicht.

Eine Beschreibung dieser dynamischen Datentypkomposition auf der Basis der vorgestellten Containertyp-Erweiterung ist folgendermaßen möglich: Zunächst wird einer *any-type*-Variablen das letzte Segment zugewiesen (`Hdr7` in Abb. 5-15). Anschließend wird schrittweise vor diesem Datenobjekt jeweils ein weiteres Segment *eingefügt*, wobei der Typ des vorherigen zuvorderst angeordneten Segments in dem neuen Segment vermerkt wird (z. B. im Feld „`next`“ von `Hdr3` bzw. `Hdr1`). So kann sukzessiv ein Datenobjekt gebildet werden, das aus der gewünschten, dynamisch festgelegten Abfolge von Datenstrukturen besteht, wobei das vorderste (also zuletzt eingefügte) Segment (`Hdr1`) einen apriori bekannten Typ hat (`Hdr1` in Abb. 5-15). Diese dynamisch erzeugte Datenstruktur soll dann später bei der Datendekomposition (s. u.) wieder schrittweise in die Ausgangstypen zerlegt werden.[29]

---

28. Eine Verallgemeinerung dieser Annahme besteht darin, auf der Basis einer strukturellen Gemeinsamkeit zwischen den verschiedenen Headern die Identifikation des Typs (und damit der Struktur) eines Headers auf der Basis von Komponenten eben dieses Headers durchzuführen. So könnten z. B. alle Header *am selben Offset* (apriori-Kenntnis) ein Feld beinhalten, das den vollständigen Typ des Headers identifiziert. Wenn die Struktur dieses Headers bis hin zu diesem Feld vorab bekannt ist (also alle Header mit der selben Struktur anfangen), kann man dies durch eine Aufteilung des Headers in zwei Teile (den Struktur-bekannten ersten Teil und den zweiten Teil, dessen Struktur durch die Felder des ersten Teils determiniert wird) auf das oben beschriebene Szenario zurückführen. Der allgemeine Fall, dass Datenfelder mit beliebigen Offsets in dem nicht weiter Struktur-bekannten Header evaluiert werden müssen, liegt jenseits der Zielsetzung der hier vorgestellten Spezifikationsmethode und erfordert in seiner Allgemeinheit den Einsatz von primitiven Funktionen zur Ermittlung des Typs bzw. zur Typkonvertierung.

29. Wir werden später im Zusammenhang mit der Datendekomposition näher darauf eingehen, warum die hier vorgestellte Komposition durch *Einfügen* von Daten vor das *any-type*-Objekt erfolgt, anstatt durch Anfügen von Daten hinter diesem (d.h. der Komposition beginnend mit der Konvertierung des ersten Headers).

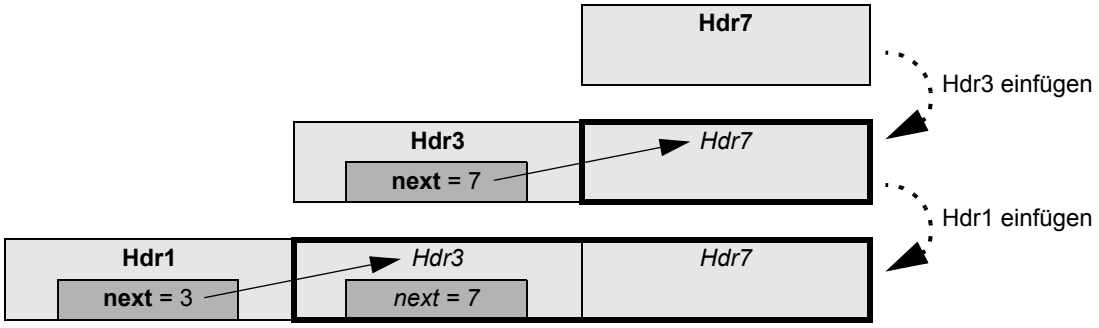


Abbildung 5-15: Sukzessive Komposition eines Paketes aus drei Segmentheadern

Es stellt sich jedoch die Frage, wie dieses *Ein-* oder *Anfügen* eines Datenobjekts an den Inhalt einer *any-type*-Variablen möglich ist, obwohl neben der Wertzuweisung keine modifizierenden Operationen auf Instanzen des Typs *any-type* existieren (siehe Abschnitt 5.2.2). Die Wertzuweisung selbst ist jedoch nicht ohne weiteres zum Ein- oder Anfügen geeignet, da durch sie der vorherige Inhalt der Variablen überschrieben wird.

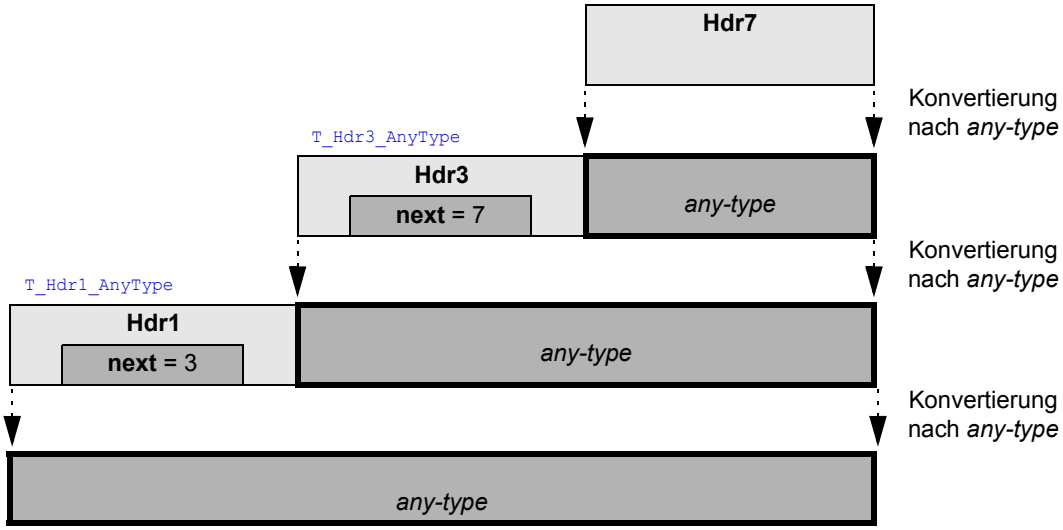


Abbildung 5-16: Sukzessive Komposition durch *any-type*-Aggregation

Die Lösung besteht darin, nicht wirklich den *any-type*-Wert durch Ein- oder Anfügen eines Datenobjekts zu *modifizieren*, sondern durch Konkatenation beider Objekte einen *neuen any-type-Wert* zu kreieren.[30] Dies geschieht, indem ein neuer Strukturtyp definiert wird, der aus einer *any-type*-Komponente und einer zweiten Komponente vom gewünschten anzufügenden Typ besteht. Weist man nun der ersten Komponente den Wert des einzufügenden Datenobjekts und der zweiten Komponenten den bisherigen *any-type*-Wert zu und konvertiert anschließend den gesamten Strukturtyp nach *any-type*, so erreicht man die geforderte Typkonkatenation (siehe Abb. 5-16). Wir werden im Folgenden dieses Prinzip anhand eines Beispiels erläutern.

30. analog zur Vorgehensweise bei der Erweiterung einer Liste in der Programmiersprache LISP [May95]

Beispiel 5.39: Dynamische Typkomposition

In diesem Beispiel soll die in Abb. 5-15 dargestellte Typstruktur dynamisch aufgebaut werden. Dazu definieren wir zunächst die Segment-Header `Hdr1`, `Hdr3` und `Hdr7`, die neben entsprechenden Nutzdaten auch jeweils (bis auf `Hdr7`) das Strukturfeld `next` beinhalten, in dem der Typ des nächsten Headers durch einen Zahlenwert angegeben wird (`3` entspricht `Hdr3`, und `7` entspricht `Hdr7`). Der Strukturtyp `Hdr7` selbst enthält kein `next`-Feld, da dieses Segment immer das letzte in der Paketstruktur bildet.[31]

Beispiel 5.39-a: Typdefinition der Paketheader

```
TYPE HdrId = 0..7;                (* identifiziert Headertypen *)

     Hdr1 =    RECORD             (* erstes Segment jedes Pakets *)
                    (* ... *)
               next: HdrId;
               END;

     Hdr3 =    RECORD
                    (* ... *)
               next: HdrId;
               END;

     Hdr7 =    RECORD                   (* letztes Segment jedes Pakets *)
                    (* ... *)
               END;
```

(Ende von Beispiel 5.39-a)

Als Nächstes definieren wir zur besseren Übersicht eine Funktion, die das Aneinanderfügen eines gegebenen Typs und eines *any-type*-Wertes am Beispiel von `Hdr3` realisiert. Diese Funktion `Insert_Hdr3` liefert als Ergebnistyp bereits den erzeugten Strukturwert, abstrahiert als *any-type*-Wert.[32]

Beispiel 5.39-b: Hilfsfunktion zum Verbinden eines `Hdr3`-Wertes und eines any-type-Wertes

```
TYPE T_Hdr3_AnyType =  RECORD
                            h:   Hdr3;
                            at:  ANY TYPE;
                       END;

FUNCTION Insert_Hdr3(h: Hdr3; at: ANY TYPE): ANY TYPE;
     (* liefert neues any-type-Objekt als Konkatenation von h und at *)
     VAR  x :  T_Hdr3_AnyType;
     BEGIN
          x.h  := h;
```

31. Dies entspricht der Struktur eines IPv6-Pakets (siehe Abb. 5-14 auf Seite 209), bei der ein TCP-Segment immer das Ende des Paketes indiziert und daher kein `next`-Feld enthält.

32. Die Definition des Strukturtyps `T_Hdr3_AnyType` wird später zur Datentyp-Dekomposition nochmals benötigt und wurde deshalb nicht in die Funktion integriert.

```
        x.at := at;
        Insert_Hdr3 := x;                    (* any-type-Konvertierung *)
    END;
```

(Ende von Beispiel 5.39-b)

Aufbauend auf der Funktion `Insert_Hdr3` und der analog definierten Funktion `Insert_Hdr1` kann nun die dynamische Komposition der in Abb. 5-15 gezeigten Typstruktur erfolgen:

Beispiel 5.39-c: Durchführung einer dynamischen Typkomposition

```
VAR  h1:  Hdr1;
     h3:  Hdr3;
     h7:  Hdr7;
     at_a, at_b, at_c: ANY TYPE;

BEGIN
     (* ... initialisiere sonstige Felder von h1, h3, h7 ... *)
     at_a := h7;                                  (* --> at_a = „Hdr7" *)
     h3.next := 7;
     at_b := Insert_Hdr3(h3, at_a);(* --> at_b = „Hdr3(next=7),Hdr7" *)
     h1.next := 3;
     at_c := Insert_Hdr1(h1, at_b);
                     (* --> at_c = „Hdr1(next=3),Hdr3(next=7),Hdr7" *)
     (* ... verschicke at_c ... *)
END;
```

(Ende von Beispiel 5.39-c)

Wesentlich an diesem Beispiel ist, dass im obigen Codefragment der Wert der *any-type*-Variablen `at_a` ... `at_c` jeweils unverändert bleibt, nachdem ihnen einmal ein Wert zugewiesen wurde. Da die Werte dieser Variablen in dem Beispiel jedoch nach ihrer Weitergabe an die Funktionen `Insert_Hdr3` bzw. `Insert_Hdr1` nicht weiter benötigt werden, genügt auch eine einzige Variable vom Typ *any-type*, auf der nach folgendem Muster operiert wird:

Beispiel 5.39-d: Vereinfachte Durchführung einer dynamischen Typkomposition

```
VAR  at: ANY TYPE;

BEGIN
     (* ... initialisiere sonstige Felder von h1, h3, h7 ... *)
     at      := h7;
     h3.next := 7;
     at      := Insert_Hdr3(h3, at);
     h1.next := 3;
     at      := Insert_Hdr1(h1, at);
     (* ... verschicke at ... *)
END;
```

(Ende von Beispiel 5.39-d)

Wesentlich bei dieser Vorgehensweise ist, dass in allen Fällen lediglich *Wertzuweisungen* auf Instanzen des Typs *any-type* erfolgt sind.

Die anschließende Dekomposition der erzeugten Datenobjekte beim Empfänger muss gemäß der Spezifikation der Containertyp-Erweiterung in Abschnitt 5.2.2 dadurch erfolgen, dass die jeweiligen Konvertierungen eines Typs in den *any-type* durch Rückkonvertierung in den jeweiligen Ausgangstyp invertiert wird. Aufgrund der dynamischen Paketstruktur muss diese Rückkonvertierung nun natürlich ebenfalls dynamisch gesteuert erfolgen.

Zunächst definieren wir zur besseren Übersichtlichkeit wiederum Zugriffsfunktionen zur Dekomposition eines Datentypverbundes aus einem *any-type*-Objekt:

Beispiel 5.39-e: Hilfsfunktion zur Extraktion eines `Hdr3`-Wertes aus einem *any-type*-Wert

```
FUNCTION Extract_Hdr3(at: ANY TYPE, VAR h: Hdr3): ANY TYPE;
     (* extrahiert Hdr3-Objekt aus at und legt es in h ab. *)
     (* Rückgabewert ist der verbleibende at-Rest.         *)
     (* at muss zuvor ein T_Hdr3_AnyType-Objekt enthalten! *)
     VAR  x =  T_Hdr3_AnyType;
     BEGIN
          x := at;                    (* any-type-Rückkonvertierung   *)
          h  := x.h;                  (* Hdr3-Objekt in h zurückgeben *)
          Extract_Hdr3 := x.at;       (* Rückgabewert: Rest aus at    *)
     END;
```

(Ende von Beispiel 5.39-e)

Nachdem solche Zugriffsfunktionen für alle Headertypen definiert wurden, kann die Dekomposition durch folgendes Codefragment erfolgen, wobei die Variable `at` zunächst das empfangene Paket enthalten soll. Dabei wird ausgehend von der Kenntnis des festen Typs des ersten Headers (Hdr1) schrittweise das gesamte Paket dekomponiert (siehe Abb. 5-17).

Beispiel 5.39-f: Durchführung einer dynamischen Typdekomposition

```
VAR  at: ANY TYPE;
     h1:  Hdr1;
     h3:  Hdr3;
     (* ... *)

BEGIN
     (* Paket liegt in Variable at! *)
     at := Extract_Hdr1(h1, at);(* apriori : Paket beginnt mit Hdr1 *)
     next := h1.next;
     (* ... sonstige Felder in h1 weiterverarbeiten ... *)
     REPEAT
          CASE next OF
               (* ... *)
               3:   BEGIN     (* jetziger Wert von at beginnt mit Hdr3 *)
                    at := Extract_Hdr3(at, h3);
                    (* ... sonstige Felder in h3 weiterverarbeiten ... *)
                    next := h3.next;
                    END;
               (* ... *)
               7:   BEGIN     (* jetziger Wert von at besteht aus Hdr7 *)
                    at := h7;
                    (* ... sonstige Felder in h7 weiterverarbeiten ... *)
                    next := 0;     (* Schleife beenden *)
                    END;
```

```
            END;  (* CASE *)
        UNTIL next = 0;
END;
```

(Ende von Beispiel 5.39-f)

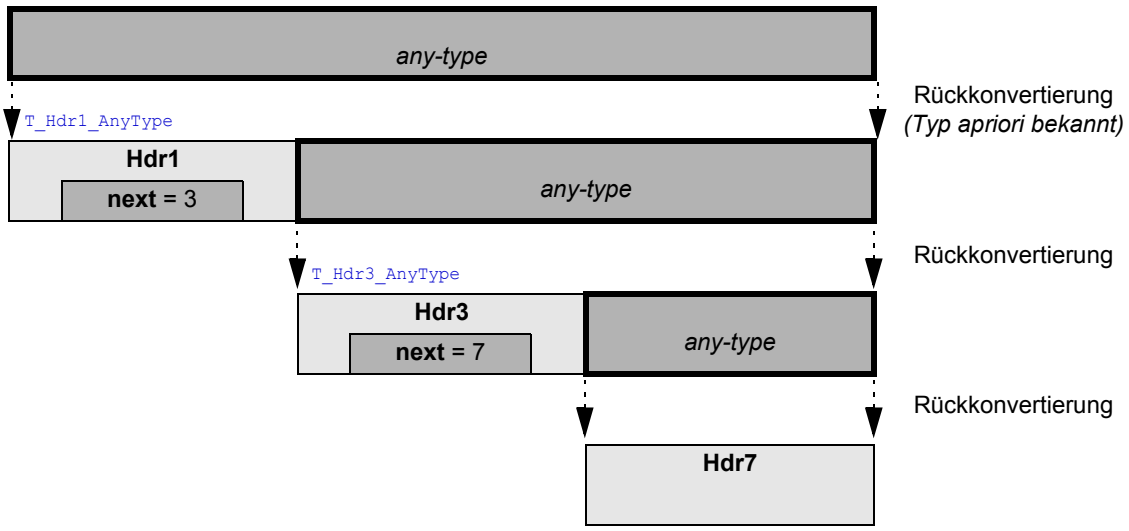


Abbildung 5-17: Sukzessive Dekomposition einer dynamischen Typstruktur

An dieser Stelle wird auch offensichtlich, warum bei der oben gezeigten Komposition des Paketes die einzelnen Segmente vorne eingefügt, die Struktur also gewissermaßen „von hinten nach vorne“ aufgebaut wurde: Bei der Dekomposition kann die verschachtelte Typstruktur nur durch schrittweise Rückkonvertierung der *any-type*-Objekte in diejenigen Typen erfolgen, aus denen sie erzeugt wurden. Entsprechend muss der Typ, der bei der Dekomposition als erster extrahiert werden soll, bei der Komposition als letzter integriert worden sein. Dies ist in den hier betrachteten Fällen jeweils der vorderste Header. Entsprechend bedingt die hier geforderte sukzessive Dekomposition „von vorne nach hinten“ eine entsprechende Komposition „von hinten nach vorne“.

(Ende von Beispiel 5.39)

5.3.3 Fundierung der strukturellen Typrekursion

Durch das vorgestellte Spezifikationsmuster auf Basis der *Containertyp*-Erweiterung ergibt sich eine *rekursive Typstruktur*, die, wie gezeigt, zur Repräsentation dynamisch strukturierter Datentypen (im vorgestellten Fall von PDU-Strukturen) eingesetzt werden kann. Ein wesentlicher Aspekt bei solchen rekursiven Strukturen ist die *Fundierung der Rekursion*. Wir werden diese nun durch eine technisch verhältnismäßig geringfügige Ergänzung konzeptionell abschließen.

Im vorangegangenen Abschnitt erfolgte die Fundierung bei der Behandlung des letzten Segmentes eines Pakets (`Hdr7`) asymmetrisch, indem dieses als einziges Segment nicht durch Einbettung in einen künstlichen Aggregationstyp (wie z. B. `T_Hdr3_AnyType` beim Segmentheader `Hdr3`) mit einem *any-type* konkateniert wurde. Die Ursache dafür ist, dass in diesem Bei-

spiel das Segment `Hdr7` immer auch das Ende der Kette von Segmenten in einem Paket darstellt. Entsprechend ergibt sich auch keine Notwendigkeit, eine solche Möglichkeit zum Anfügen eines weiteren Segmentes bereitzustellen.

Eine solche Lösung ist hinreichend für Protokolle, bei denen das jeweils letzte Segment eines dynamisch komponierten Pakets anhand seiner Typkennung als solches erkennbar ist und entsprechend auch keine Verweise auf die Typen folgender Segmente enthält. Ein Beispiel dafür ist IPv6, bei dem in einem Paket im Anschluss an das eigentliche Nutzlastsegment (z. B. ein TCP-Paket, siehe Abb. 5-14 auf Seite 209) keine weiteren Segmente mehr folgen.

Eine Verallgemeinerung der Vorgehensweise ergibt sich entsprechend durch die Aufhebung dieser Beschränkung. Angewandt auf das obige Beispiel könnte dies bedeuten, dass das konstruierte Paket bereits mit `Hdr3` endet, wobei diese Tatsache im Paket durch den Wert 0 für das Feld `next` in `Hdr3` angezeigt wird. Ein solches Paket könnte durch die im obigen Beispiel entwickelte Dekompositionsroutine (siehe Estelle-Fragment auf Seite 214) problemlos abgehandelt werden, da dort als Signal für das Ende des Pakets neben dem Auftreten eines `Hdr7`-Segmentes auch ein Wert von 0 in einem der `next`-Felder eines anderen Headers genügt.

Bei der Komposition eines solchen Pakets ergibt sich jedoch ein Problem: Da auf Grund der nunmehr (bis auf den verpflichtenden Paketheader `Hdr1`) vollständig flexibilisierten Segmentfolge das Ende des Pakets erst *nach* der Dekomposition des letzten Segments erkannt wird, ist auch keine apriori Sonderbehandlung dieses letzten Segments mehr möglich. Entsprechend muss der nunmehr als letztes Segment des Pakets angenommene `Hdr3` zusammen mit einer nachgestellten *any-type*-Komponente in seinen Aggregationstyp `T_Hdr3_AnyType` eingebettet werden. Da `Hdr3` das letzte Segment des Pakets darstellen soll, enthält jedoch diese *any-type*-Komponente gar keine Daten, sondern ist *leer*. Es ergibt sich die in Abb. 5-18 dargestellte Paketstruktur.

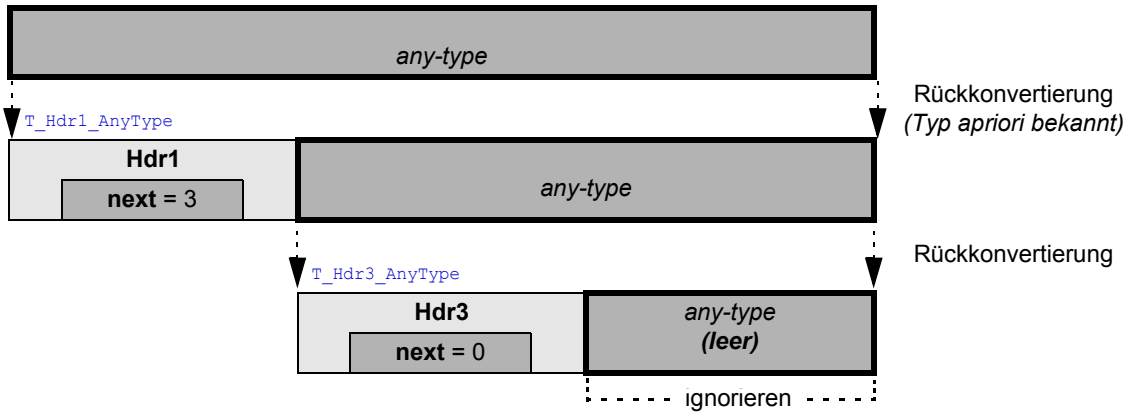


Abbildung 5-18: Sukzessive Dekomposition mit leerem Restsegment

Es stellt sich jedoch die Frage, wie ein solches *leeres Datenobjekt* zur Fundierung der strukturellen Typrekursion bei der Komposition mit den bisher vorgestellten Mitteln überhaupt erzeugt werden kann, da dazu ja eine Wertzuweisung auf die *any-type*-Komponente `at` des Aggregationstyps `T_Hdr3_AnyType` notwendig ist. Zu diesem Zweck ergänzen wir die Definition des Containertyps um folgende Konvention:

> *Instanzen eines `RECORDs` ohne Komponenten ergeben bei der Konvertierung in den Typ any-type ein leeres Datenobjekt.*

Entsprechend liefert die folgende Funktion den gesuchten *any-type*-Wert:

Beispiel 5.40: Funktion `EmptyAnyType` zur Erzeugung eines leeren Datenobjekts

```
FUNCTION EmptyAnyType: ANY TYPE;
     TYPE EmptyType = RECORD END;          (* Record ohne Komponenten *)
     VAR  x :  EmptyType;
     BEGIN
          EmptyAnyType := x;      (* Zuweisung des leeren Datenobjekts *)
     END;
```

(Ende von Beispiel 5.40)

Auf konzeptioneller Ebene ist die Bedeutung des Begriffs „*leeres Datenobjekt*“ nahe liegend: Das leere Datenobjekt dient zum Abbruch der oben zur dynamischen Datentyp-Komposition eingesetzten *Strukturtyp-Rekursion*, ohne wirklich ein Datenobjekt mit nicht-leerem Wertebereich einfügen zu müssen. Welche Auswirkungen dies auf Implementierungsebene hat, werden wir später in Abschnitt 5.4 nochmals näher beleuchten. Bis auf weiteres genügt die Vorstellung, dass das leere Datenobjekt bei der Kodierung eines Pakets keinen Platz einnimmt. Folglich kann mit einem solchen Datenobjekt auch keine Information übertragen werden, und dementsprechend ist bei der Dekomposition eines Pakets auch keine Rückkonvertierung des *any-type*-Wertes notwendig.

Durch die Einführung des leeren Datenobjekts vereinfacht sich auch die symmetrische Darstellung der Komposition eines Datenpaketes, indem die schrittweise Erweiterung des *any-type*-Wertes bei der Komposition durch die Initialisierung mit dem leeren Datenobjekt jeweils Segment für Segment unabhängig von den vorangegangenen Segmenten erfolgen kann. Dies zeigt sich besonders bei der Spezifikation einer dynamischen Komposition eines Datenpaketes, bei der zur Laufzeit für jedes Segment festgelegt wird, ob er Teil des Paketes wird oder nicht. Allein das zuletzt eingefügte Segment (der Paketheader `Hdr1`) ist obligatorisch.

Im folgenden Spezifikationsfragment wird z. B. das Vorhandensein des Segmentes vom Typ `Hdr3` davon abhängig gemacht, welchen Wert die boolesche Variable `add_h3` hat. Dabei spielt es keine Rolle, ob `Hdr3` das letzte Segment des Paketes ist (beim Einfügen also die Variable `anytype` immer noch das von der Funktion `EmptyAnyType` zurück gelieferte leere Datenobjekt enthält), oder zwischenzeitlich weitere Segmente eingefügt wurden.

Beispiel 5.41: Fundierung der Typrekursion durch das leere Datenobjekt

```
VAR  anytype: ANY TYPE;
     h1:      Hdr1;
     h3:      Hdr3;
     prev:    INTEGER;
     add_hdr3:BOOLEAN;

BEGIN
     (* ... initialisiere add_hdr und sonst. Felder von h1, h3, ... *)
     anytype  := EmptyAnyType;
     prev     := 0;
     (* ... *)
     if (add_h3) {
          h3.next  := prev;
          anytype  := Insert_Hdr3(h3, anytype);
          prev     := 3;
     }
     (* ... *)
```

```
    h1.next   := prev;
    anytype   := Insert_Hdr1(h1, anytype);
    prev      := 1;
    (* ... verschicke anytype ... *)
END;
```

(Ende von Beispiel 5.41)

Im folgenden Abschnitt werden wir die Grundidee bei der Einführung der Funktion `EmptyAnyType` verallgemeinern und so die Handhabung der *Containertyp*-Erweiterung verbessern.

5.3.4 Kompaktdarstellung der *any-type*-Komposition und -Dekomposition

In den vorangegangenen Abschnitten haben wir gezeigt, wie verschiedene Datenobjekte als Komponenten eines RECORDs aggregiert und dann in den *any-type* konvertiert bzw. später zurückkonvertiert wurden. Dabei spielte der aggregierende RECORD-Typ selbst nur eine untergeordnete Rolle als zusammenfassende Klammer der einzelnen Komponenten. Dies zeigt sich u. a. daran, dass der eigentliche Aggregationsvorgang in den obigen Beispielen in einer Funktion (`Insert_Hdr3`, siehe Beispiel 5.39-b auf Seite 212) eingekapselt wurde.

Zur notationalen Vereinfachung des Aggregationsvorgangs führen wir daher eine syntaktische Erweiterung ein, mit der auf die explizite Definition des o. g. RECORD-Typs gänzlich verzichtet werden kann. Wir motivieren dies an folgendem Beispiel:

Beispiel 5.42: Konventionelle *any-type*-Komposition

Zum Zwecke des Framings eines Datums sdu vom Typ `T_SDU` mit einem `header` und einem `trailer` (Typ `T_HEADER` bzw. `T_TRAILER`) wurde nach der bisherigen Vorgehensweise folgender Typ `T_PDU` definiert:

```
TYPE T_PDU =  RECORD
                  header:  T_HEADER;
                  sdu:     T_SDU;
                  trailer: T_TRAILER;
              END;
```

Die Erzeugung eines *any-type*-Wertes mit der Aggregation der o. g. Variablen erfolgte dann z. B. durch folgende Funktion:

```
FUNCTION Compose(hdr: T_HEADER, sdu: T_SDU, trl: T_TRAILER): ANY TYPE;
    (* liefert neues any-type-Objekt als Konkatenation der Parameter *)
    VAR x : T_PDU;
    BEGIN
        x.header  := hdr;
        x.sdu     := sdu;
        x.trailer := trl;
        Compose   := x;                    (* any-type-Konvertierung *)
    END;
```

Die Komposition der Werte von `header`, `sdu` und `trailer` erfolgte dann durch Aufruf von `Compose`, wobei der Ergebnistyp bereits als *any-type* vom enthaltenen Strukturtyp `T_SDU` abstrahiert:

```
VAR  at: ANY TYPE;
BEGIN
     at := Compose(header, sdu, trailer);
     (* ... verwende at *)
END;
```

(Ende von Beispiel 5.42)

Da bei einfachen (d.h. nicht-varianten) RECORD-Typen die Komposition der Komponenten offensichtlich ausschließlich von der Reihenfolge der Kompontenten-Typen abhängt, kann auf die explizite Angabe des Typs T_PDU und der Kompositionsfunktion Compose verzichtet werden. Dazu führen wir folgende vereinfachende Notation durch Erweiterung[33] des nicht-Terminals EXPRESSION ein:

Definition 5.6: Syntax der Kompaktdarstellung der *any-type*-Komposition

```
EXPRESSION = - | "<" ACTUAL-PARAMETER-LIST ">"
```

Constraint: The parameters in the **ACTUAL-PARAMETER-LIST** shall be not-pointer-containing.

(Ende von Definition 5.6)

Sie beschreibt die Komposition von Werten in einen (implizit definierten) Strukturtypen mit Komponenten[34] der Typen der jeweiligen aktuellen Parameter in der angegebenen Reihenfolge und die anschließende Konvertierung des Strukturtyp-Wertes in den *any-type*.

So kann der in Beispiel 5.39 dargestellte Konvertierungsvorgang in der Kompaktdarstellung ohne Notwendigkeit der Definition von Hilfs-Struktur-Typen oder -Funktionen folgendermaßen dargestellt werden:

Beispiel 5.43: Kompaktdarstellung der *any-type*-Komposition

```
VAR  at:      ANY TYPE;
     header:  T_HEADER;
     sdu:     T_SDU;
     trailer: T_TRAILER
BEGIN
     (* ... belege header, sdu, trailer *)
     at := < header, sdu, trailer >;
     (* ... verwende at *)
END;
```

(Ende von Beispiel 5.43)

33. Die Schreibweise mit „- | “ entspricht der in Abschnitt 8.1 des Estelle-Standards [ISO97] eingeführten Notation zur Bezugnahme auf die rechte Seite der bisherigen (hier Standard-Estelle-) Definition.

34. Die Namen der Komponenten sind dabei (bis auf ihre paarweise Verschiedenheit) unerheblich (s. u.).

Naheliegenderweise kann auch die Rückkonvertierung eines *any-type*s in seine enthaltenen Komponenten in einer analogen Kurznotation erfolgen. Dazu erweitern wir die Definition des nicht-Terminals VARIABLE-ACCESS und ergänzen die Hilfsdefinition:

Definition 5.7: Syntax der Kompaktdarstellung der *any-type*-Dekomposition

```
ASSIGNMENT-STATEMENT = - | "<" VARIABLE-ACCESS-LIST ">" ":=" EXPRESSION.

VARIABLE-ACCESS-LIST = VARIABLE-ACCESS, { "," VARIABLE-ACCESS }
```

Constraint: The **EXPRESSION** in the definition of **ASSIGNMENT-STATEMENT** shall have the type **any-type** and it shall contain an structured type created by an expression as defined in def. 5.6 which (i) had the same number of parameters and (ii) the **VARIABLE-ACCESS** and **ACTUAL-PARAMETER** with same index have identical types.

(Ende von Definition 5.7)

Mit Hilfe dieser neuen Form eines ASSIGNMENT-STATEMENTs kann nun die Umkehrung der o. g. Komposition in kompakter Form dargestellt werden. Dabei werden die Komponenten des rückzukonvertierenden RECORDs in ihrer vorgegebenen Reihenfolge den jeweiligen Einträgen der VARIABLE-ACCESS-LIST zugewiesen.

So kann die in Beispiel 5.43 dargestellte Kompaktdarstellung der *any-type*-Komposition durch folgende Anweisung umgekehrt werden:

Beispiel 5.44: Kompaktdarstellung der *any-type*-Dekomposition

```
VAR  at:      ANY TYPE;
     header:  T_HEADER;
     sdu:     T_SDU;
     trailer: T_TRAILER
BEGIN
     (* ... belege at *)
     < header, sdu, trailer > := at ;
     (* ... verwende header, sdu, trailer *)
END;
```

(Ende von Beispiel 5.44)

Als syntaktisch auflösbare Erweiterung kann die vorgestellte Kompaktdarstellung für die *any-type*-Komposition und -Dekomposition semantisch auf die früher vorgestellte Containertyp-Erweiterung zurückgeführt werden. Die korrekte Anwendbarkeit der Containertyp-Erweiterung basiert jedoch auf der identischen Definition der Typen (hier: RECORD-Typen) bei der Komposition und der Dekomposition. Bei der vorgestellten Erweiterung haben wir von den konkreten Strukturtypen jedoch gerade abstrahiert und uns stattdessen allein auf die Sequenz der Komponententypen bezogen. Zur Fundierung der Kompaktschreibweise gehen wir deshalb davon aus, dass es zu jeder in einer konkreten Spezifikation in der Kompaktdarstellung auftretenden Sequenz von Komponententypen systemweit einen eindeutig festgelegten Strukturtyp gibt, der aus Komponenten mit genau dieser Sequenz besteht und den entsprechenden Kompaktdarstellungen für die *any-type*-Komposition und -Dekomposition syntaktisch zu Grunde liegt.[35]

Die vorgestellte Kompaktdarstellung stellt eine erhebliche Vereinfachung der Darstellung gerade von dynamisch aggregierten Datentypen (siehe Abschnitt 5.3.2) dar. Insbesondere kann als Spezialfall das in Abschnitt 5.3.3 eingeführte leere Datenobjekt offensichtlich direkt durch den

Ausdruck[36] „<>“ erzeugt werden. Somit kann ein Paket, das gemäß der Vorgaben von Beispiel 5.39 auf Seite 212 verschachtelt aus den Headern `h1` und `h3` besteht, durch folgenden Statementblock erzeugt werden:

Beispiel 5.45-a: Kompaktdarstellung der *any-type*-Komposition mit leerem Datenobjekt

```
VAR  anytype: ANY TYPE;
     h1:      Hdr1;
     h3:      Hdr3;
     prev:    INTEGER;
     add_hdr3:BOOLEAN;

BEGIN
     (* ... initialisiere h1, h3, ... *)
     anytype  := < >;
     prev     := 0;
     (* ... *)
     if (add_h3) {
          h3.next  := prev;
          anytype  := < h3, anytype >;
          prev     := 3;
     }
     (* ... *)
     h1.next  := prev;
     anytype  := < h1, anytype >;
     prev     := 1;
     (* ... verschicke anytype ... *)
END;
```

(Ende von Beispiel 5.45-a)

Ohne Dynamik in der Paketstruktur kann das gleiche Paket dann noch wesentlich kompakter erzeugt werden:

Beispiel 5.45-b: Kompaktdarstellung der *any-type*-Komposition mit leerem Datenobjekt

```
VAR  anytype: ANY TYPE;
     h1:      Hdr1;
     h3:      Hdr3;

BEGIN
     (* ... initialisiere h1, h3, ... *)
     h3.next  := 0;
```

35. Durch ein geeignetes Namensschema ist zu jeder gegebenen Sequenz von Komponententypen ein solcher Typ leicht eindeutig automatisch ableitbar. Da dieser Typ nicht notwendigerweise expliziter Bestandteil der Spezifikation sein muss, wird in den Constraints von Def. 5.7 die Rückkonvertierung von *any-type*-Objekten mit Hilfe der vorgestellten Kurzschreibweise vorläufig auch explizit auf solche *any-type*-Objekte beschränkt, die auch mit Hilfe dieser Erweiterung erzeugt wurden. Eine Beseitigung dieser Beschränkung ist jedoch z. B. mit einer geeigneten Typ- und Namenskonvention möglich.

36. Um eine syntaktische Unterscheidung vom Symbol der mathematischen ungleich-Relation „<>“ zu gewährleisten ist das leere Datenobjekt mit einem trennenden „whitespace“ (z. B. einem Leerzeichen) anzugeben.

```
      h1.next   := 3;
      anytype   := < h1, < h3, < > > >;
      (* ... verschicke anytype ... *)
END;
```

(Ende von Beispiel 5.45-b)

Wir wollen an dieser Stelle den Einsatz der vorgestellten syntaktischen Kompaktdarstellung nicht weiter vertiefen und wenden uns stattdessen im nächsten Abschnitt der Fragestellung der grundlegenden Implementierbarkeit der Containertyp-Erweiterung selbst zu. Dabei werden wir u. a. nochmals auf die Anwendung dieser Erweiterung zur formalen Spezifikation von komplexeren Datenkompositions- und Dekompositionsszenarien zurückkommen.

5.4. Implementierung der Containertyp-Erweiterung

Die Containertyp-Erweiterung ergänzt das Typmodell von Estelle um die Möglichkeit zur abstrakten Beschreibung dynamisch aggregierter Datentypen. Dies erfordert gegenüber dem einfachen, statisch aggregierten Typsystem von Standard-Estelle eine grundlegend andere Implementierungsstruktur. Dazu geben wir im folgenden Abschnitt ein erstes Referenz-Implementierungsmodell für die Containertyp-Erweiterung auf der Basis des Typ-Implementierungsmodells von XEC an (Abschnitt 5.4.1). Um die dabei auftretenden implementierungstechnischen Probleme zu vermeiden, führen wir anschließend ein verbessertes Implementierungsmodell ein und beleuchten dessen praktische Umsetzung in der aktuellen Version von XEC (Abschnitt 5.4.2). Schließlich beschäftigen wir uns noch mit der Frage, welche Auswirkungen die zur Übertragung eines *any-type*-Datenobjekts über einen „realen“ Datenübertragungsdienst notwendige Serialisierung auf das Implementierungsmodell und die Typsicherheit der Containertyp-Erweiterung hat (Abschnitt 5.4.3).

5.4.1 Referenz-Implementierungsmodell der Containertyp-Erweiterung für XEC

In diesem Abschnitt gehen wir der Frage nach, wie die oben beschriebene Containertyp-Erweiterung auf einem konventionellen Mikroprozessorsystem implementiert werden kann. Eine der zentralen Fragen ist dabei, wie dieser neue Estelle-Typ in Kombination mit den bereits bekannten Estelle-Typen als Zustandsraumkomponente im Hauptspeicher und bei der Serialisierung zum Zwecke des Datentransports als Bytesequenz dargestellt werden kann. In beiden Fällen müssen die gemäß der Estelle-Semantik zunächst nur eigenschaftsorientiert beschriebenen Typen in eine binäre, d. h. im Hauptspeicher darstellbare *Kodierung* überführt werden.[37] Wir beginnen zunächst mit einem Überblick über die Kodierung der Standard-Estelle-Typen durch XEC, wobei wir die für Interaktionsparameter (und später auch die Containertyp-Konvertierungen) nicht relevanten *pointer containing* Typen zunächst ausschließen.

Wie wir bereits in Abschnitt 3.3.3 gesehen haben, modelliert der Estelle-Implementierungsgenerator XEC das Estelle-Typsystem sehr strukturnah als C- bzw. C++-Typen. Aufbauend auf den Estelle-Basistypen (z. B. `INTEGER`, `REAL`, `BOOLEAN`), die in der Standardkodierung von XEC in analoge[38] C++-Basistypen (z. B. `int`, `float`, `bool`) abgebildet werden, werden strukturierte Typen (`RECORD`, `ARRAY`, `SET`) ebenfalls durch die analogen C++-Strukturen (`struct`/`class`, C-Array) nachgebildet, wobei diese dann zusammen mit geeigneten Zugriffsfunktionen in (Template-) Klassen eingebettet sind. Die derart erzeugten C++-Typen sind durchgängig *PODTs* („*Plain Old Data Types*“, siehe [ISO98]), d. h. sie enthalten keine durch C++-Mechanismen induzierten zusätzlichen Komponenten wie z. B. *VMT*-Pointer (*Virtual Method Table*). Das Konzept der PODTs ermöglichen es, in C++ die aus C gewohnten Definitionen von binär

37. Man beachte, dass wir uns an dieser Stelle zum ersten Mal im Zusammenhang mit der Containertyp-Erweiterung mit einer konkreten Kodierung beschäftigen, da diese Fragestellung ausschließlich auf Implementierungsebene von Belang ist.

38. Die Standardkodierung von XEC liefert hier keine vollständig der Semantik entsprechende Implementierung für nicht endlich darstellbare Estelle-Typen. So bildet z. B. der C++-Typ „`int`“ nur innerhalb seines (plattformspezifisch) begrenzten Wertebereichs eine adäquate Modellierung für den wertemäßig unbegrenzten Estelle-Typ `INTEGER`..

repräsentierten Datentypen fester Struktur zu übernehmen: Ein solches Objekt kann (wenn es keine Pointer enthält) durch seine geschlossene binäre Repräsentation im Speicher vollständig kodiert und später daraus unverändert rekonstruiert werden.

Somit stellt die interne C++-Repräsentation der („*not pointer-containing*“) Standard-Estelle-Datentypen im Hauptspeicher bereits eine vollständige binäre Kodierung dieser Daten dar und kann somit auch zur Darstellung dieser Daten in Form einer Byte-Sequenz zum Zwecke des Transportes über einen „*realen*“ Datentransportdienst dienen („Transfer-Encoding“). Beim Empfang kann diese Byte-Sequenz dann direkt durch binäre Zuweisung auf den Speicherplatz eines Objekts (vom Ausgangstyp) diesem zugewiesen und somit eine inhaltsgleiche Kopie des Ausgangsobjekts erstellt werden.[39]

Konzeptionell basiert diese Form der Kodierung eines strukturierten Datentyps in ein zusammenhängendes Binärobjekt auf der Tatsache, dass sich die statische Typaggregationshierarchie von Standard-Estelle durch das oben beschriebene Modell in eine ebenfalls statische Aggregationshierarchie in C++ abbilden lässt. Eine Ursache dafür ist, dass jede Aggregationskomponente in ihrer Kodierung einen festen Platzbedarf hat und somit an einem festen Offset in der linearen Hauptspeicherrepräsentation des Gesamtobjekts positioniert werden kann. Entsprechend kann die Aggregation selbst als Konkatenation der Komponenten im linearen Hauptspeicher realisiert werden.[40]

Mit der Einführung der Containertyp-Erweiterung wird dieses Aggregationsmodell um den neuen Datentyp *any-type* erweitert, der jeden als Interaktionsparameter übertragbaren Estelle-Typen aufnehmen kann. Unabhängig von der Kodierung bei der Konvertierung in den bzw. aus dem *any-type* ergibt sich somit, dass für den Platzbedarf dieses Datentyps keine Obergrenze angegeben werden kann.[41] Entsprechend kann auch keine endliche Repräsentation für den *any-type* angegeben werden, die für die oben beschriebene statische Aggregation geeignet wäre.

Somit muss zur Implementierung des *any-type*-Datentyps aufbauend auf dem bisherigen Typkodierungsmodell eine Form gewählt werden, bei der ein *any-type*-Objekt (z. B. als Aggregationskomponente) lediglich einen Verweis auf das enthaltene Datenobjekt enthält und somit sich eine nicht notwendigerweise zusammenhängende Darstellung des Gesamtobjekts im Speicher ergibt. Eine zunächst nahe liegend erscheinende Realisierung ist die Darstellung des *any-type*-Objekts als C-Typ „`void *`“, also ein Zeiger auf einen unspezifizierten Typ. Wir betrachten dazu folgendes Estelle-Fragment:

```
TYPE T = RECORD
           header:  INTEGER;
           payload: ANY TYPE;
           trailer: INTEGER;
         END;
```

39. Voraussetzung für die reversible Kodierung eines Datenobjekts in dieser Form ist natürlich, dass die sendende und die empfangende Plattform die gleichen impliziten Kodierungsregeln (Byteorder, Länge von Basistypen, Alignment von Strukturkomponenten etc.) zu Grunde legen.

40. Eine gesonderte Betrachtung erfordern die von XEC transient erzeugten dynamischen SETs, sowie variante RECORDs (siehe Abschnitt 3.3.3).

41. Man kann leicht zeigen, dass z. B. die Menge aller Werte des Typs `ARRAY [1..D] OF 0..255` für $D \geq 1$ mit keiner Abbildung reversibel in einer Bytesequenz mit *weniger* als D Bytes kodiert werden kann.

```
VAR  i, j:     INTEGER;
     a:        T;

BEGIN
     i := 10;
     a.header := 1;
     a.payload:= i;                          (* any-type-Konvertierung *)
     a.trailer:= 2;
     (* ... *)
     i := 20;
     j := a.payload;                    (* any-type-Rückkonvertierung *)
                                   (* j sollte hier den Wert 10 haben *)
END;
```

Die Darstellung von *any-type* als „`void *`“ ergibt folgende erste Implementierung in C:

```
struct T {
     int       header;
     void*     payload;
     int       trailer;
};

int       i, j;
struct T a;

/* ... */
{
     i = 10;
     a.header = 1;
     a.payload= (void*) &i;                  /* any-type-Konvertierung */
     a.trailer= 2;
     /* ... */
     i = 20;
     j = * (int*) a.payload;            /* any-type-Rückkonvertierung */
}
```

Die Ausführung dieses Codes ergibt die in Abb. 5-19 dargestellte Situation.

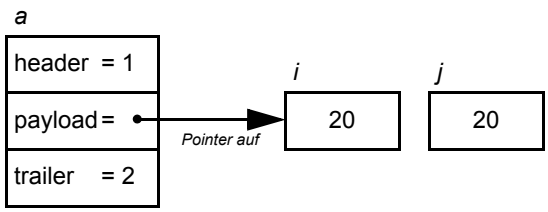


Abbildung 5-19: Situation nach Copy-By-Reference-Implementierung

Offensichtlich verletzt diese Realisierung die Copysemantik bei der Zuweisung an den *any-type*, da die nachfolgende Zuweisung auf die Variable i auch den Wert von a verändert. Entsprechend muss eine semantikkonforme Realisierung statt der einfachen Zuweisung der Adresse von `i` („`&i`“) auf die Strukturkomponente `payload` eine vollständige Kopie von `i` erstellen. Auf Grund der dynamischen Datenstruktur muss der Speicherplatz für diese Kopie ebenfalls dynamisch belegt werden:

```
a.payload = memcpy( malloc(sizeof(i)), &i, sizeof(i) );
```

Oder etwas kompakter mit analogem Ergebnis in C++:

```
a.payload = (void*) new int(i);
```

Die Ausführung dieses Codes ergibt die in Abb. 5-20 dargestellte Situation.

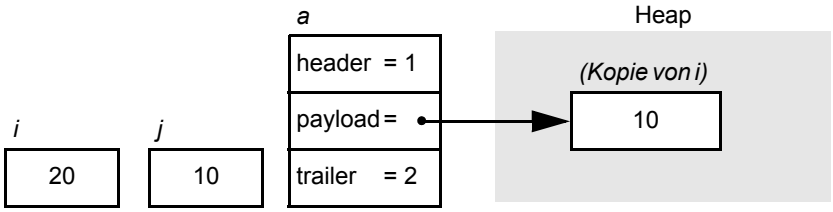


Abbildung 5-20: Situation nach Copy-Implementierung

Dieser Ansatz bietet bereits eine semantisch vollständige Implementierung der Containertyp-Erweiterung. Insbesondere wird hier deutlich, warum in der Definition der Containertyp-Erweiterung bei der Konvertierung von einem *any-type*-Datenobjekt zurück in einen (anderen) Zieltyp nicht nur gefordert wurde, dass der im *any-type*-Datenobjekt enthaltene Typ[42] *zuweisungskompatibel* zum Zieltyp ist, sondern mit diesen Zieltyp sogar *identisch* sein muss: Bei der Rückkonvertierung (im obigen Beispiel die Zuweisung zur Variablen `j`) wird neben der Kenntnis des Typs des Zuweisungsziels keine weiteren Typinformationen des im *any-type*-Datenobjekt enthaltenen Datums benötigt, da bei einer *zulässigen* Rückkonvertierung beide Typen identisch sein müssen (siehe Abschnitt 5.2.2). Im oben gezeigten Falle der Darstellung von *any-type* als `void*` liegen auf Implementierungsebene tatsächlich auch keine weiteren Typinformationen zum enthaltenen Datenobjekt vor, sodass der Zieltyp der Rückkonvertierung die einzige Grundlage für die Interpretation des *any-type*-Datums bildet.[43] Kodiert man – wie im obigen Beispiel gezeigt – die *any-type*-Werte einfach als binäres Abbild der Speicherrepräsentation des jeweiligen Ausgangstyps, so genügt bei der Rückkonvertierung in den selben Typ ebenfalls eine binäre Zuweisung der enthaltenen Speicherrepräsentation:

```
memcpy( &i, a.payload, sizeof(i) );
```

bzw.

```
i = *(int*) a.payload;
```

oder etwas expliziter in C++:

```
i = *static_cast<int*>(a.payload);
```

42. d. h. der Typ, der ursprünglich nach *any-type* konvertiert wurde

43. Einige C/C++-Plattformen bieten die Möglichkeit, die allozierte Größe eines Heap-Objekts nachträglich zu ermitteln. Auf Grund der fehlenden Eindeutigkeit der Zuordnung von Objekt-Größen zu den Objekt-Typen kann diese Information jedoch bestenfalls zur Erkennung unzulässiger Rückkonvertierungen genutzt werden.

Nachdem wir nun eine einfache Implementierung für die Konvertierung von Standard-Estelle-Typen in den *any-type* und zurück in den Ausgangstyp gefunden haben, beschäftigen wir uns als Nächstes mit der Frage, wie sich die Konvertierung in den *any-type* auf Typen auswirken, die selbst einen *any-type* als Komponente enthalten. Dazu betrachten wir aufbauend auf den vorangegangenen Beispielen folgendes ergänzendes Estelle-Fragment:

```
VAR b: T;                                    (* a hat auch den Typ T *)

BEGIN
     b.header  := 3;
     b.payload  := a;                        (* any-type-Konvertierung *)
     b.trailer := 4;
END;
```

Gemäß der oben gezeigten Implementierungsmethode wird bei der Ausführung der Anweisung „`b.payload := a`“ eine binäre Kopie des Inhalts von `a` auf dem Heap erzeugt, und es ergibt sich somit die in Abb. 5-21 gezeigte Situation.

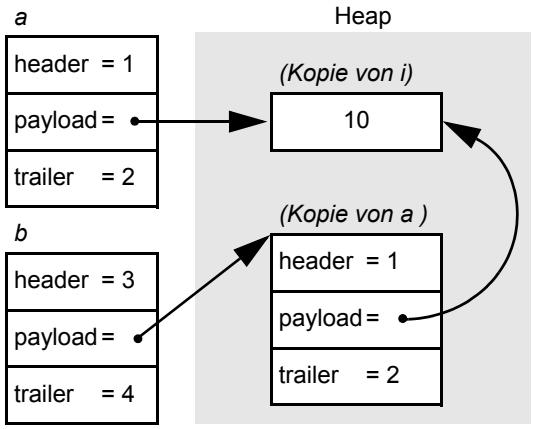


Abbildung 5-21: Kopieren verschachtelter any-type-Werte

An diesem Beispiel wird die Problematik der oben vorgestellten Implementierungsmethode bei der verschachtelten Zuweisung von *any-type*-enthaltenden Datenstrukturen zu *any-type*-Variablen augenscheinlich, da auf diese Art unkontrolliert[44] Mehrfachreferenzen auf Heap-Objekte entstehen. Dies macht die synchrone Freigabe der Heap-Objekte bei der Freigabe der Referenzen problematisch: Wird z. B. `a.payload` ein neuer Wert zugewiesen, so darf das ursprünglich referenzierte Heap-Objekt („*Kopie von i*“) nicht freigegeben werden, da noch Referenzen auf dieses Objekt existieren (siehe Abb. 5-21). Dies ist umso problematischer, als die tatsächliche Typstruktur des von `b.payload` referenzierten Heap-Objekts („*Kopie von a*“) in diesem Zustand gar nicht mehr bekannt ist: Erst bei einer möglichen Rückkonvertierung in den Ausgangstyp steht der Typ des in der *any-type*-Komponente von b enthaltenen Objekts wieder zur Verfügung. Somit kann bei dieser Implementierungsmethode auch nur bedingt adhoc ermittelt werden, wie viele Referenzen noch auf ein solches Heap-Objekt existieren (z. B. für eine Garbage-Collection). Ein analoges Problem ergibt sich, wenn derartige verschachtelte Datenstrukturen wieder freigegeben werden, indem der referenzierenden *any-type*-Variablen ein neuer Wert zugewiesen wird. So würde beim in Abb. 5-21 dargestellte Szenario ohne die Existenz der Referenz durch `a.payload` eine neue Wertzuweisung auf die Komponente `b.payload` man-

44. Wir werden später in Abschnitt 6.4.6 auf die Kontrolle von Mehrfachreferenzen zurückkommen.

gels der Kenntnis der Struktur des von `b.payload` referenzierten Heap-Objekts („*Kopie von a*“) bestenfalls dieses direkt referenzierte Objekt freigegeben, das indirekt referenzierte Objekt („*Kopie von i*“) jedoch nicht.

Die Problematik lässt sich zurückführen auf die Implementierung der Zuweisung eines *any-type*-Wertes auf eine *any-type*-Variable, wie sie im obigen Beispiel bei der Kopie des Inhalts der Variablen `a` (zum Zwecke der Zuweisung auf `b.payload`) erfolgte. Wir sind oben davon ausgegangen, dass auf Grund der binären Kopie des Inhalt von `a` auch der Wert der `void*`-Variablen mitkopiert würde und somit eine Mehrfachreferenz auf das von `a.payload` referenzierte Objekt entstünde. Diese Methode ist hinsichtlich der geforderten Copy-Semantik zulässig, da die Containertyp-Erweiterung (mit Ausnahme der Zuweisung eines neuen Wertes) konzeptionell keine Manipulation des *Inhalts* einer *any-type*-Variablen unterstützt und somit auch keine Verletzung der Copy-Semantik durch Mehrfachreferenzen entstehen kann. Die Zuweisung eines neuen Wertes auf eine *any-type*-Variable hingegen bewirkt lediglich die Ersetzung einer Referenz auf ein Heap-Objekt durch einen anderen Wert, indem die `void*`-Komponente, welche die *any-type*-Variable implementiert, einen neuen Wert erhält. Unter der Voraussetzung, dass ein solches Heap-Objekt frühestens[45] dann freigegeben und somit modifiziert wird, wenn keine Referenzen mehr auf selbiges bestehen, erfüllt diese Implementierung die Copy-Semantik.

Eine nahe liegende alternative Implementierung der Kopie einer *any-type*-Variablen könnte nun darin bestehen, statt einer einfachen Kopie des `void*`-Wertes auch das referenzierte Objekt zu kopieren, sodass insgesamt keine Mehrfachreferenzen entstünden. Das Problem besteht nun darin, dass bei einer Kaskade von *any-type*-Referenzen (z. B. bei `b.payload` in Abb. 5-21) auch alle rekursiv enthaltenen *any-type*-Referenzen kopiert werden müssten. Dem steht in der obigen Implementierungsmethode jedoch wiederum die Unkenntnis des tatsächlichen Typs des Inhalts einer *any-type*-Variablen (d. h. auf Implementierungsebene das durch diesen `void*` referenzierte Heap-Objekt) entgegen.

Wir werden im nächsten Abschnitt jedoch eine C++-Typstruktur entwickeln, die eine vollständig typerhaltende interne Darstellung der Werte einer *any-type*-Variablen liefert und somit ein vollständiges (d. h. rekursives und typerhaltendes) Kopieren eines *any-type*-Wertes ermöglicht.

5.4.2 Implementierung der Containertyp-Erweiterung im XEC-Toolkit

Die Implementierung der Containertyp-Erweiterung auf der Basis des XEC-Toolkits[46] betrifft

(i) den Compiler-Front-End PET,

(ii) den Implementierungsgenerator XEC und

(iii) die Laufzeitbibliothek XECRT.

---

45. Neben einer Garbage-Collection, die unreferenzierte Heap-Objekte ggf. verzögert freigibt, wäre es prinzipiell auch denkbar (wenn auch bei lange laufenden Implementierungen mit fortgesetzter Allokation neuer *any-type*-Objekte nicht praktikabel), solche Objekte überhaupt nicht freizugeben.

46. Die im Zusammenhang mit der Containertyp-Erweiterung eingeführten Modifikationen der Quellen aller Komponenten werden im C++-Quelltext durch das Präprozessor-Symbol `ANYTYPE_EXT` aktiviert und sind somit leicht als solche zu identifizieren.

Die Erweiterung des Compiler-Frontends PET gliedert sich im Wesentlichen in eine Ergänzung des Parsers zur Behandlung des durch die Containertyp-Erweiterung eingeführten neuen Type-Denoters „`ANY TYPE`“ (siehe Abschnitt 5.2.2), die Einführung einer Klasse `AnyType` in das Klassensystem zur Repräsentation des neuen Estelle-Typs im Syntaxbaum einer Spezifikation und die Erweiterung der Kompatibilitätsregeln für die Konvertierung zwischen den bisherigen Estelle-Typen und dem *any-type*.

Herzstück der Modifikationen bezüglich der Typkompatibilitätsregeln ist die Erweiterung der Funktion `assignCompatibleTypes`, die zu zwei übergebenen Typen anzeigt, ob der zweite Typ dem ersten zugewiesen werden kann. Im folgenden C++-Fragment von PET (Datei `type.cc`) ist diese Funktion vollständig angegeben. Die beiden im Zusammenhang mit der Containertyp-Erweiterung hinzugefügten Bedingungen, unter denen zwei Typen zuweisungskompatibel sind, erlauben die Zuweisung eines „not pointer containing“ Typs auf einen *any-type* bzw. die Zuweisung in umgekehrter Richtung.[47] Es ist dabei zu beachten, dass gerade bei der Rückkonvertierung die zusätzlichen Anforderungen an die Gleichheit des im *any-type* enthaltenen Typs mit dem Typ des Zuweisungsziels (siehe Def. 5.4 auf Seite 199) hier noch nicht geprüft wird, da dies im Allgemeinen erst später zur Laufzeit möglich ist (s. u.).

```
int assignCompatibleTypes( PDef pd1, PDef pd2) {
    return (
        compatibleTypes( pd1, pd2)
        || (typeOf(pd1)==(PDef)&PredefReal)
                && compatibleTypes(pd2, (PDef)&PredefInt)
#ifdef ANYTYPE_EXT
        || ((typeOf(pd1)==(PDef)&PredefAnyType)
                && !pointerContaining(pd2))
        || ((typeOf(pd2)==(PDef)&PredefAnyType)
                && !pointerContaining(pd1))
#endif
    );
}
```

Die Anpassung des Implementierungsgenerators XEC zur Umsetzung der Containertyp-Erweiterung gestaltete sich dagegen minimalistisch, da lediglich beim Auftreten des Type-Denoters „`ANY TYPE`“ in dem von PET gelieferten Syntaxbaum der Spezifikation (in Funktion `CDeclDef::dump_type`) der entsprechende Klassenname der XEC-Laufzeit-Bibliothek „`AnyType`“[48] ausgegeben werden muss:

```
CDeclDef::dump_type(ostream& os, PTypeDenoter p, /* ... */) {
    switch( p->defCat()) {
        /* ... */
        case INTEGERTYPE:
            return os << "TYPE_integer";
```

47. Der Fall der Zuweisung eines *any-type*-Wertes auf einen *any-type* wird bereits durch die Funktion `compatibleTypes` positiv abgehandelt, da diese u. a. identische Typen immer als kompatibel identifiziert.

48. Nicht zu verwechseln mit der gleichnamigen Klasse „`AnyType`“ aus der PET-Klassen-Bibliothek, die auf Quelltextebene in keinem direkten Zusammenhang mit der XEC-Laufzeit-Bibliothek steht: Die PET-Klassen-Bibliothek wird zum Zeitpunkt der Übersetzung von PET und XEC selbst genutzt, während die XEC-Laufzeit-Bibliothek nur zum Zeitpunkt der Übersetzung des von XEC generierten Codes zum Einsatz kommt.

```
            /* ... */
#ifdef ANYTYPE_EXT
            case ANYTYPE:
                return os << "AnyType";
#endif
            /* ... */
        } /* end switch */
}
```

Somit erfolgt seitens des Codegenerators keinerlei Sonderbehandlung des *any-type*s und der in Abschnitt 5.2.2 spezifizierten Konvertierungen zwischen anderen Typen und dem *any-type*. Aus Sicht des Codegenerators ist der *any-type* nichts anderes als ein atomarer, vordefinierter Estelle-Typ wie z. B. auch `INTEGER`. Die Behandlung der Typkonvertierungen erfolgt vollständig durch ein geeignetes C++-Klassensystem in der Laufzeitbibliothek XECRT.

Diese Erweiterung der Laufzeitbibliothek besteht ausschließlich in der Einführung der Klasse `AnyType`, die – wie wir oben gesehen haben – vom Code Generator XEC als Typ für alle Instanzen eines *any-type*s eingesetzt wird. Gemäß der in Abschnitt 5.2.2 geforderten Eigenschaften des *any-type*s muss diese Klasse folgende Operationen unterstützen:

- die Zuweisung eines beliebigen anderen Typs
- die Konvertierung des Inhalts in einen beliebigen anderen Typ (wobei dieser Inhalt aus dem Zieltyp entstandenen ist)
- das unveränderte Kopieren des Inhalts auf eine andere *any-type*-Variable

Wir werden sehen, dass gerade die ersten beiden Operationen in C++ durch den Einsatz von Templates sehr elegant implementiert werden können, indem entsprechende Methoden dieser Klasse derart als Funktions-Templates modelliert werden, dass bei der Zuweisung eines fremden Typs bzw. der Konvertierung in einen fremden Typ dieser fremde Typ selbstständig vom C++-Compiler korrekt zugeordnet wird und so schließlich eine typsichere Behandlung dieser Konvertierungen erfolgen kann.

Der folgende Ausschnitt aus der Klassendefinition von `AnyType` zeigt die Signaturen der dafür verantwortlichen Methoden, die wir nun etwas näher betrachten wollen:

```
class AnyType {
    /* ... */
  public:
    // --- Konvertierungs-Methoden
    const AnyType& operator=(const AnyType& at);                    // 1
    template <class TT> const AnyType& operator=(const TT& tt);     // 2
    template <class TT> operator const TT();                        // 3
    // --- Konstruktoren
    AnyType();                                                      // 4
    AnyType(const AnyType& at);                                     // 5
    template <class TT> AnyType(const TT& tt);                      // 6
};
```

Die sechs Methoden gruppieren sich in drei Konvertierungs-Methoden und drei Konstruktor-Methoden, wobei insgesamt drei Methoden Templates mit jeweils einem Typparameter „`TT`“ sind. Es ist zu beachten, dass die Klasse `AnyType` selbst jedoch kein Klassen-Template ist, sondern eine reguläre Klasse, die lediglich Typ-parametrierbare Methoden enthält. Entsprechend können ohne weitere Parametrierung Instanzen der Klasse `AnyType` erzeugt werden, jedoch benötigen zu ihrem Aufruf einige der Methoden Typ-Parameter. Diese Typ-Parameter müssen

jedoch in keinem Fall explizit angegeben werden, sondern werden vom C++-Compiler automatisch aus dem Kontext ermittelt. Technisch besonders bemerkenswert sind dabei das Type-Cast-Operator-Template (3) und das Konstruktor-Template (6).

Betrachten wir jedoch zunächst die regulären (d. h. nicht Template) Methoden. Die Konstruktor-Methode (4) dient zur Erzeugung einer nicht explizit initialisierten `AnyType`-Instanz. Methode (1) dient dazu, den Inhalt des `AnyType`-Objekts zu verwerfen und ihm den Wert eines anderen `AnyType`-Objekts zuzuweisen. Die Semantik dieses Vorgangs ist nahe liegend, mit der konkreten Implementierung werden wir uns später noch beschäftigen. Die Konstruktor-Methode (5) dient als Copy-Konstruktor für diese Klasse und wird z. B. benötigt, um ein `AnyType`-Objekt „*by-value*“ als aktuellen Funktionsparameter zu übergeben („`AnyType z(y);`“ in Beispiel 5.46). Sie implementiert die sequentielle Ausführung der beiden Schritte Initialisierung gemäß Konstruktor-Methode (4) und einer anschließenden Zuweisung des übergebenen Wertes auf das eigene Objekt gemäß Methode (1) (z. B. durch „`*this = at;`“).

Bevor wir zu den Template-Methoden kommen, demonstrieren wir den Einsatz aller Methoden am folgenden Beispiel, das die wichtigsten Fälle abdeckt, die im Zusammenhang mit der Containertyp-Erweiterung in von XEC generiertem C++-Code auftreten. In den jeweiligen Kommentaren sind die Nummern der dort implizit oder explizit aufgerufenen Methoden angegeben.

Beispiel 5.46: Implizite und explizite Methodenaufrufe der C++-Klasse „`AnyType`“

```
int i = 1, j = 2;
class C { /* ... */ } c;

void f(int);
void g(AnyType);

AnyType x;    // (4) leere Initialisierung
AnyType y(3); // (6) Initialisierung durch beliebigen Typ
AnyType z(y); // (5) Initialisierung als Kopie

/* ... Statements: */

// * --> AnyType
x = i;    // (2) Zuweisung eines beliebigen Typs (primitiv, aus Variable)
x = 3;    // (2) Zuweisung eines beliebigen Typs (primitiv, aus Wert)
x = c;    // (2) Zuweisung eines beliebigen Typs (Klasseninstanz)
g(i);     // (6) Zuweisung eines beliebigen Typs
          //     (primitiv, Funktionsparameter)
g(4);     // (6) Zuweisung eines beliebigen Typs
          //     (primitiv, Funktionsparameter)

// AnyType --> *
j = x;    // (3) Konvertierung auf beliebigen Typ
          //     (primitiv, auf Variable)
f(x);     // (3) Konvertierung auf beliebigen Typ (primitiv, auf Wert)

// AnyType --> AnyType
y = x;    // (1) unveränderte Kopie eines AnyType-Inhalts
g(x);     // (5) unveränderte Kopie eines AnyType-Inhalts
```

(Ende von Beispiel 5.46)

Wie man an dem Beispiel sieht, erfolgt die Interaktion zwischen den Instanzen der Klasse `AnyType` und anderen Typen ebenso nahtlos wie bereits auf der Ebene von Estelle (mit der Containertyp-Erweiterung). Syntaktisch wird dies auf der C++-Ebene durch die bereits genannten Methoden-Templates der Klasse `AnyType` erreicht. Die Methode (2) dient dabei dazu, eine Referenz auf eine Instanz eines beliebigen *fremden* Typs TT an das `AnyType` zu übergeben, um dessen Wert darin abzulegen. Diese Methode wird bei direkten Zuweisungen von Instanzen anderer Typen („`x = i`“ und „`x = c`“ im obigen Beispiel) eingesetzt. Bei der Zuweisung von primitiven Werten wie z. B. in „`x = 3`“ wird vom C++-Compiler automatisch ein temporäres Objekt mit diesem Wert erzeugt, dessen Referenz dann an die Methode weitergegeben werden kann.

Bei genauer Betrachtung stellt man an dieser Stelle fest, dass bei der Zuweisung eines `AnyType`-Objekts auf ein anderes grundsätzlich zwei passende Methoden (1 und 2) existieren:

```
const AnyType& operator=(const AnyType& at);                  // 1
template <class TT> const AnyType& operator=(const TT& tt);   // 2
```

Da C++ in solchen Fällen der nicht-Template-Methode (1) den Vorrang gibt (siehe Abschnitt 14.8.3 und 13.3.3 in [ISO98]), löst sich dieser Konflikt ganz im Sinne der Semantik der zu Grunde liegenden Containertyp-Erweiterung auf: Bei der Zuweisung von *any-type* auf *any-type* ist keine Konvertierung erforderlich und entsprechend muss die Zielvariable nach der Zuweisung eine Kopie des Wertes der Quellvariablen enthalten.

Analog zur regulären Konstruktor-Methode (5) dient das Konstruktor-Methoden-Template (6) als Copy-Konstruktor, wobei hier eine Referenz auf eine Instanz eines beliebigen fremden Typs `TT` übergeben wird, um dessen Wert in dem gerade erzeugten `AnyType`-Objekt abzulegen. Das Konstruktor-Template wird z. B. benötigt, um einen fremden Typ als aktuellen Parameter an eine Funktion mit formalem Parameter-Typ `AnyType` „*by-value*“ zu übergeben („`g(i)`“ im obigen Beispiel). Sie implementiert die sequentiellen Ausführungen der Initialisierung gemäß Konstruktor-Methode (4) und einer anschließenden Zuweisung der übergebenen fremden Typinstanz auf das eigene Objekt gemäß Methode (2) (z. B. durch „`*this = tt;`“). Und wie bei Methode (2) wird auch hier bei der Übergabe von primitiven Werten wie z. B. in „`g(4)`“ vom C++-Compiler automatisch ein temporäres Objekt mit diesem Wert erzeugt, dessen Referenz dann an die Methode weitergegeben werden kann.

Syntaktisch wie auch semantisch besonders interessant ist das Type-Cast-Operator-Template (3), das als einzige Methode die Umwandlung eines in einem `AnyType`-Objekt ablegten Wertes in einen fremden Typ (den Ausgangstyp) ermöglicht:

```
template <class TT> operator const TT();                      // 3
```

Type-Cast-Operatoren dienen in C++ normalerweise dazu, die implizite Umwandlung einer Klasse in einen konkreten Zieltyp durchzuführen. Ein typisches Beispiel wäre die folgende Klasse `Float` zur Abstraktion des C/C++-Gleitkommatyps `float`:

Beispiel 5.47: Typsicherheit von Type-Cast-Operatoren in C++

```
class Float {
    float value;
    static const float EPSILON;
    /* ... */
  public:
    int as_int();
```

```
    operator int() {return as_int();}
    operator float() {return value;}
    operator bool() {return value < EPSILON;}
};
```

Möchte man den in einer Instanz von `Float` abgelegten Gleitkommawert als Ganzzahlwert auslesen, so kann man neben der dazu vorgesehenen explizit aufzurufenden Methode `Float::as_int()` auch den Type-Cast-Operator mit dem Zieltyp `int` implizit aufrufen:

```
Float x;
int i;
float r;
void f(int i);

i = x;        // ruft implizit "operator int()" auf
f(x);         // ruft implizit "operator int()" auf
r = x;        // ruft implizit "operator float()" auf
if (x) ...    // ruft implizit "operator bool()" auf
```

(Ende von Beispiel 5.47)

Interessant an diesen Type-Cast-Operatoren ist, dass sie kontextsensitiv vom Compiler genutzt werden: Sofern der von der Umgebung benötigte Zieltyp eindeutig einem der Ergebnistypen der Type-Cast-Operatoren zuzuordnen ist, wird der entsprechende Operator implizit aufgerufen.

Das Type-Cast-Operator-Template (3) der Klasse `AnyType` stellt dabei eine *extreme Verallgemeinerung* dieses Mechanismus dar, indem es als Template-Parameter gerade diesen Zieltyp der Konvertierung erhält und somit die Konvertierung in andere Typen effektiv allquantifiziert. Dadurch kann dieser Type-Cast-Operator immer angewendet werden, solange der Zieltyp eindeutig feststeht. Dies ist (wie in den vorangegangenen Beispielen) bei direkten Zuweisungen und bei Parameterübergaben der Fall.

Da die Containertyp-Erweiterung bei der Rückkonvertierung vom *any-type* in einen fremden Typ ausschließlich Situationen erlaubt, in denen der Zieltyp eindeutig feststeht, ist das Type-Cast-Operator-Template (3) bei allen Rückkonvertierungen vom C++-Compiler automatisch implizit anwendbar.

Aufbauend auf dieser äußeren Schnittstelle der C++-Klasse `AnyType` mit ihren Typ-adaptiven Methoden-Templates zur Zuweisung eines beliebigen fremden Typs auf ein `AnyType`-Objekt bzw. der Rückkonvertierung des Inhalts eines `AnyType`-Objekts in einen vom Anwendungskontext festgelegten fremden Typ geben wir nun die in der aktuellen Laufzeitbibliothek XECRT eingesetzte Implementierung dieser Klasse an. Diese realisiert eine vollständig automatisch implementierbare, typsichere und effiziente Implementierung der Containertyp-Erweiterung.

Dazu definiert die Klasse `AnyType` die beiden privaten Klassen `AnyType::Abstract_Container` und `AnyType::Container`. Wie im UML-Klassendiagramm in Abb. 5-22 dargestellt, ist die Klasse `AnyType::Container` ein Klassen-Template, das die reguläre Klasse `AnyType::Abstract_Container` als (abstrakte) Basisklasse besitzt.[49]

49. Aufgrund der Eindeutigkeit der Klassennamen verzichten wir im Folgenden im Fließtext auf die vollständige Qualifizierung der Klassennamen.

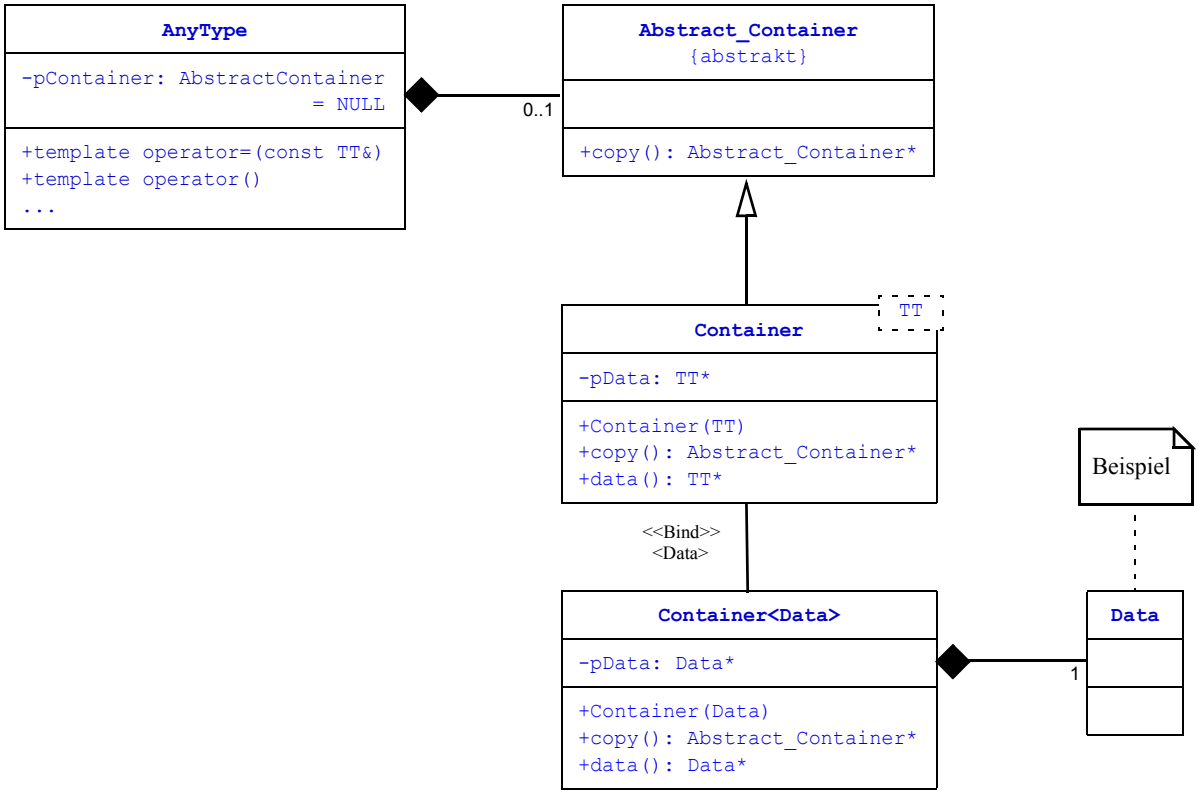


Abbildung 5-22: UML-Diagramm der Struktur der XECRT-Klasse „AnyType“

```
class AnyType {
  private:
     class Abstract_Container {
          /* ... */
     };
     template <class TT> class Container : public Abstract_Container {
          /* ... */
     };
     Abstract_Container* pContainer;
  public:
     /* ... Zugriffsmethoden (s. o.) ... */
};
```

Die Klasse AnyType selbst enthält neben den weiter oben genannten Zugriffsmethoden im Wesentlichen nur noch die Instanzvariable pContainer vom Typ Zeiger auf die Klasse Abstract_Container.[50] Bei einem AnyType-Objekt, das kein Datenobjekt enthält (z. B. weil ihm bisher explizit kein Wert zugewiesen wurde), hat dieser Pointer den Wert NULL.

Wird nun einem solchen AnyType-Objekt ein Wert eines beliebigen fremden Typs zugewiesen, so wird das dabei aufgerufene Methoden-Template (2) (mit dem fremden Typ des zugewiesenen Wertes als Typparameter TT) instanziiert. Eine Implementierung dieser Methode könnte folgendermaßen aussehen:

50. Die Klasse AnyType selbst ist somit ein *PODT* („*Plain Old Data Type*“, siehe [ISO98]).

```
template <class TT> void AnyType::operator=(const TT& tt) {
    delete pContainer;
    pContainer = new Container<TT>(tt);
}
```

Nach der Freigabe des alten Inhalts des `AnyType`-Objekts wird eine neue Instanz des Klassen-Templates `Container<TT>` erzeugt, wobei als Typ-Parameter explizit der zugewiesene fremde Typ angegeben wird. Der aufgerufene Konstruktor der Klasse `Container<TT>` erstellt dabei unter Kenntnis des an `TT` gebundenen Typs eine vollständige Kopie des zugewiesenen Wertes `tt`. Dies schließt auch die rekursive Kopie ggf. enthaltener weiterer `AnyType`-Objekte ein. Die Adresse dieser Kopie wird dabei in der privaten Instanzvariablen `Container::pData` (die vom Typ „`TT*`“ ist) abgelegt. Da für jeden beliebigen Typ `TT` die Klasse `Container<TT>` von der Basisklasse `Abstract_Container` abgeleitet ist, ist die Zuweisung eines solchen Zeigers auf den „`Abstract_Container* pContainer`“ eine zulässige Typabstraktion.

Beim Kopieren eines `AnyType`-Objekts auf ein anderes wird gemäß der Semantik der Containertyp-Erweiterung auch eine vollständige Kopie des enthaltenen Wertes erzeugt.[51] Diese Kopie soll vollständig typsicher erfolgen, damit die ggf. in dem zu übertragenden Wert fremden Typs enthaltenen `AnyType`-Objekte ebenfalls rekursiv kopiert werden. Um dies ohne explizite Angabe[52] des im `AnyType`-Objekts enthaltenen Typs realisieren zu können, enthält die abstrakte Basisklasse `Abstract_Container` eine (reine) virtuelle Methode namens `copy`, die im abgeleiteten Klassen-Template `Container<TT>` redefiniert wird:

```
template <class TT> Abstract_Container* Container<TT>::copy() const {
    return new Container<TT>(*pData);
}
```

Durch diese Konstellation kann ohne jede weitere Kenntnis des im `AnyType`-Objekts enthaltenen Typs durch Aufruf der Funktion `AnyType::pContainer->copy()` eine vollständige und typsichere Kopie dieses `AnyType`-Objekts erzeugt werden. Analog dazu arbeitet auch die Freigabe des Inhalts eines `AnyType`-Objekts, indem die Klasse `Abstract_Container` einen virtuellen Destruktor enthält, der durch einen Typ-spezifischen Konstruktor in dem Klassen-Template `Container<TT>` redefiniert wird. Dadurch kann auch die *Freigabe* des Inhalts eines `AnyType`-Objekts[53] typsicher und damit (ggf. durch rekursives Kopieren) vollständig erfolgen.

Der interessanteste Fall ergibt sich bei der Rückkonvertierung des Inhalts eines `AnyType`-Objekts in einen fremden Typ. Durch die implizite Typparametrierung des entsprechenden Type-Cast-Operator-Templates (3) der Klasse `AnyType` ist der Typ des Zuweisungsziels bekannt, und es ergibt sich folgende erste Implementierung:

```
template <class TT> AnyType::operator const TT() {
    return static_cast< Container<TT>* >(pContainer)->data();
}
```

---

51. Wir werden uns später im Zusammenhang mit effizienten Datenübertragungstechniken noch einmal mit diesem Punkt beschäftigen (siehe Abschnitt 6.4.6).

52. Aus dem Kontext ergibt sich bei einer *any-type* zu *any-type*-Zuweisung kein Hinweis auf die enthaltenen Typen.

53. dies erfolgt ohne explizite Angabe des enthaltenen Typs

Hier wird unter der Annahme, dass der Typ des im `AnyType`-Objekt enthaltenen Wertes mit dem von der Umgebung vorgegebenen Ziel-Typ der Rückkonvertierung (gebunden an `TT`) übereinstimmt, zunächst der Zeiger `pContainer` (Typ Zeiger auf die Basis-Klasse `Abstract_Container`) mit Hilfe des `static_cast`-Operators in einen Zeiger auf die abgeleitete Klasse `Container<TT>` umgewandelt und dann der Wert des enthaltenen Datums mit Hilfe der Funktion `Container<TT>::data()` als Kopie zurückgegeben. Da diese letzte Kopie typsicher erfolgt, können auch hier ggf. in diesem Wert enthaltene weitere `AnyType`-Objekte korrekt rekursiv kopiert werden.

Diese erste Implementierung des Type-Cast-Operator-Templates (3) erfüllt vollständig die geforderte Semantik der Containertyp-Erweiterung. Sie kann jedoch relativ leicht erweitert werden, sodass mögliche Verletzungen der Forderung nach Typgleichheit des Inhalts des *any-type*-Objekts und des Zuweisungsziels der Rückkonvertierung (`TT`) erkannt werden (siehe Def. 5.4 auf Seite 199). Eine solche Verletzung kann auf zwei Ursachen beruhen:

(i) der in dem *any-type*-Objekt enthaltene Wert ist durch Zuweisung eines Wertes von einem anderen Typ als dem Zieltyp entstanden oder

(ii) das `AnyType`-Objekt enthält gar keinen Wert, z. B. wenn noch gar kein Wert zugewiesen wurde („*preinitial state*“, siehe Abschnitt 9.4.5.2 von [ISO97]).

Der Fall *(ii)* kann leicht dadurch erkannt werden, dass der Zeiger `AnyType::pContainer` den Wert `NULL` hat.

Der Fall *(i)* einer Rückkonvertierung eines *any-type*-Wertes in einen falschen Zieltyp lässt sich mit Hilfe des C++-Operators `dynamic_cast` erkennen: Da die Basisklasse `Abstract_Container` virtuelle Methoden enthält, kann der `dynamic_cast`-Operator[54] zur Laufzeit ermitteln, ob der als Parameter übergebene Zeiger auf die Basisklasse tatsächlich auf eine Instanz der explizit als Typparameter (in spitzen Klammern) angegebenen abgeleiteten Klasse[55] zeigt. Wenn dies der Fall ist, also die Rückkonvertierung vom *any-type* in den konkreten Zieltyp zulässig ist, so liefert der dynamic_cast-Operator den entsprechenden (typkonvertierten) Zeigerwert. Bei einem unzulässigen Konvertierungsversuch wird dagegen der `NULL`-Pointer zurückgegeben. Somit ergibt sich eine *vollständig (d. h. statisch wie auch dynamisch) typsichere Realisierung* folgendermaßen:[56]

```
template <class TT> AnyType::operator const TT() {
    if (pContainer == NULL)
        error();      // Fehler: Objekt enthält keinen Wert
    Container<TT>* p = dynamic_cast< Container<TT>* >(pContainer);
    if (p == NULL)
        error();      // Fehler: Typen verschieden
    return p->data();
};
```

54. Die Prüfung der Zulässigkeit der Konvertierung, wie auch die Konvertierung selbst erfolgen in den meisten C++-Implementierungen anhand der in polymorphen Objekten enthaltenen VMT-Verweise.

55. Tatsächlich werden natürlich Zeiger-Typen auf die jeweiligen Klassen angegeben.

56. Es wird davon ausgegangen, dass ein Aufruf der Funktion `error()` nicht in den lokalen Aufrufkontext zurückkehrt (z. B. durch sofortige Programm-Termination mittels `halt()` oder Auslösen einer Exception mit dem `throw`-Statement), da es ansonsten bei der Ausführung des `return`-Statements zu einer (unzulässigen) NULL-Pointer-Dereferenzierung kommt.

Damit ist die Präsentation der wesentlichen Mechanismen der Containertyp-Erweiterung von XEC abgeschlossen. Wesentlich bei dieser Implementierung ist, dass sie *vollständig typsicher* arbeitet und insbesondere auch zur Laufzeit die Einhaltung der geforderten Typgleichheit bei der Rückkonvertierung von *any-type* in einen regulären Estelle-Typ *validieren* kann. Dies wird erreicht, indem der logische Baum von rekursiv ineinander verschachtelten *any-type*-Objekten und ihren jeweiligen Inhalten in der Implementierung durch Einsatz eines Template-Klassensystems vollständig und typsicher erhalten bleibt. Dies bedeutet, dass in gewissem Maße *Zusatzinformationen* über die Typstruktur des Inhalts einer *any-type*-Variablen mit in dieser abgelegt wurde. Wir werden im nächsten Abschnitt sehen, wie sich der Umgang mit dem *any-type* gestaltet, wenn solche Zusatzinformationen nicht mehr zur Verfügung stehen.

5.4.3 Kodierung und Serialisierung von *any-type*-Objekten

In den vorangegangenen Abschnitten haben wir verschiedene Implementierungsansätze für die Containertyp-Erweiterung gesehen, die frei von Implementierungsdetails (wie z. B. der Angabe einer konkreten Kodierung) eine vollautomatische Implementierung des spezifizierten Systems erlauben. Ziel war es dabei, bereits auf einer hohen Abstraktionsebene der formalen Beschreibung automatisch Implementierungen ableiten zu können. So kann z. B. ein abstrakter Datentransportdienst mit Hilfe der Containertyp-Erweiterung formal spezifiziert und anschließend vollautomatisch implementiert werden, ohne dass explizite Angaben zur Kodierung oder den Paketformaten hinzugefügt werden müssen.

Weiterhin haben wir gesehen, wie auf der Basis der Containertyp-Erweiterung bereits auf sehr abstrakter Ebene eine Spezifikation von dynamisch aufgebauten Paketstrukturen möglich war, durch die die funktionalen Aspekte von Protokollen wie z. B. TCP oder IPv6 durchaus angemessen repräsentiert werden können (siehe Abschnitt 5.3). Auch hier wurden die Details der Kodierung der übertragenen Daten der (ebenfalls vollautomatisch ableitbaren) Implementierung überlassen.

Wenn wir uns bei der Frage nach der automatischen Implementierbarkeit einen weiteren Schritt der Präzision manueller Implementierungen annähern, so ergibt sich natürlich auch die Frage, wie eine konkret vorgegebene Kodierung erreicht werden kann, die möglicherweise Bestandteil der Protokollspezifikation ist. So enthalten die meisten Protokolle präzise Kodierungsregeln, die neben den grundlegenden Strukturen der Pakete[57] auch Details wie die Anzahl der Bytes zur Darstellung eines Wertes (*Range*), die Ausrichtung der einzelnen Felder (*Alignment*)[58] und die Reihenfolge der Bytes bei der Ablage von Zahlenwerten (*Byte-Order*) umfassen. Es ist offensichtlich, dass diese Regeln explizit formuliert werden müssen, wenn eine automatisch generierte Implementierung sogar auf Binärebene kompatibel mit einem Protokoll bzw. manuellen Protokollimplementierungen sein soll.

57. also Abfolge und grundlegender Typ der einzelnen Felder

58. Als „*Alignment*“ bezeichnet man die Positionierung von bestimmten Datentypen oder Feldern auf Adressen (oder Paket-Offsets) mit bestimmten Teilbarkeitsanforderungen. So müssen z. B. in der SUN-SPARC-Architektur 4-Byte-Integer auf durch 4 teilbare Adressen zu liegen kommen, damit auf diese zugegriffen werden kann. Unterschiedliche Alignment-Anforderungen erfordern häufig das Einfügen von ungenutzten sog. Padding-Feldern in Datenstrukturen oder -Paketen.

Konzeptionell ist diese Aufgabenstellung vergleichbar mit *Encoding-Rules*, wie sie z. B. für *ASN.1* [ISO88a] existieren bzw. man sie dafür definieren kann. Auch dort wird eine formal beschriebene Datenstruktur in eine konkrete binäre Darstellung übersetzt. Die *Basic-Encoding-Rules* (BER, [ISO88b]) von ASN.1 sind dabei jedoch nicht geeignet, um eine *beliebige* binäre Darstellung (z. B. für einen TCP-Paketheader) zu erreichen, sondern sie definieren letztlich ein *eigenes Binärformat*. Somit kann ASN.1 als Basis zur *Implementierung des Containertyperweiterung* eingesetzt werden, wenn keine spezifischen Anforderungen an die Binärrepräsentation der kodierten Pakete bestehen bzw. die gewünschten Encoding-Rules zur Verfügung stehen. Wir kommen in Abschnitt 5.5 nochmals auf diesen Aspekt zurück.

Um jedoch eine frei vorgegebene binäre Darstellung zu erreichen, müssen auch allgemein entsprechende Encoding-Rules definiert und implementiert werden. Solche spezifischen Encoding-Rules ermöglichen es dann auch, auf explizite Zusatzinformationen zur Kodierung von dynamischen[59] Datenstrukturen zu verzichten und auf die in einem solchen dynamisch strukturierten Datenpaket zwingend (implizit oder explizit) enthaltenen Zusatzinformationen zurückzugreifen. So haben wir z. B. in Abschnitt 5.3.2 anhand des IPv6-Paketformats gesehen, wie die in den Paketen bereits enthaltenen Strukturinformationen zur sukzessiven Dekomposition eines Datenpaketes genutzt werden können.

Vor diesem Hintergrund wird deutlich, dass die Containertyp-Erweiterung bei der Dekomposition von binär kodierten Daten mehr Zusatzinformationen bereitstellt, als dies bei ASN.1 normalerweise der Fall ist, da bei der Dekomposition (also schrittweisen Rückkonvertierung vom *any-type*) jeweils bereits auf Spezifikationsebene die Typinformation des enthaltenen Typs explizit angegeben werden muss, die in den Basic-Encoding-Rules vorgesehenen Strukturinformationen also meist gar nicht benötigt werden.

Alternativ können *any-type*-Daten jedoch auch ohne Umweg über ASN.1 spezifisch kodiert werden. Dazu ist für eine konkret vorgegebene binäre Kodierung von Daten die Angabe der zu Grunde liegenden Kodierungsregeln auf Implementierungsebene notwendig. Dies kann z. B. erfolgen, indem die Konvertierungsfunktionen für die Umwandlung eines zu übertragenden Typs in den *any-type* und zurück jeweils manuell angegeben (bzw. implementiert) werden. Damit können alle auf C/C++-Ebene darstellbaren Konvertierungsregeln realisiert werden, womit sich (mit einer entsprechenden Plattform-Abhängigkeit) beliebige Kodierungen erreichen lassen.

Verzichtet man aber auf die Festlegung einer *konkreten* Kodierung, so bleibt als Mittelweg zwischen der in den vorhergehenden Abschnitten dargestellten internen Kodierung der Estelle-Implementierung (die, wie wir gesehen haben, noch Strukturanteile[60] enthält), und einer vom Protokoll beliebig vorgegebenen binären Kodierung (die letztlich ja eine direkte Kommunikation mit einer Handimplementierung auf Ebene binärer Pakete erlaubt) noch der Zwischenschritt einer beliebigen *Serialisierung* der internen (strukturierten) Darstellung des *any-type*s, ähnlich wie es ASN.1 mit den Basic-Encoding-Rules bietet.

Die Zielsetzung dieser Kodierung besteht nun darin, so weit wie möglich ohne explizite Angabe einer konkreten Kodierung zwei kompatible, offene Estelle-Systeme vollautomatisch so zu implementieren, dass die Pakete über einen Dienst ausgetauscht werden können, der lediglich Byte-Folgen unverändert bidirektional überträgt. In diesem Szenario bleibt die konkrete Kodierung der Daten vollständig der Implementierung überlassen, solange sie als sequentielle Folge von Bytes darstellbar ist.

---

59. z. B. Rekursion, optionale Felder oder Arrays variabler Länge

60. in Form der dargestellten Baumstruktur

Diese *Serialisierung* der automatisch generierten Implementierung des *any-type*s bewirkt, dass die Dienstschnittstellen in einer hierarchisch strukturierten Diensthierarchie in der Implementierung wieder auf die in der ursprünglichen Motivation (siehe Abschnitt 5.1) dargestellten abstrakten Datentransportdienste zur Übertragung von Byte-Sequenzen abgebildet werden können (siehe Abb. 5-4 auf Seite 186).

Eine solche Serialisierung kann auf zwei Arten erfolgen: Entweder man überträgt in dieser Byte-Sequenz ausschließlich die Kodierung der jeweiligen Inhalte (ohne Kenntnis der Bedeutung dieser Inhalte), oder man fügt der Byte-Sequenz noch Zusatzinformationen über die interne Verschachtelungsstruktur der *any-type*-Komponenten hinzu. Im letzteren Fall kann natürlich in jedem Fall eine Kodierung gefunden werden, die vollständig reversibel ist (dies entspricht letztlich der Vorgehensweise der Basic-Encoding-Rules bei ASN.1). Die Übertragung der Daten *ohne Hinzufügen von Zusatzinformationen* stellt dagegen eine größere Herausforderung dar und ist nur in bestimmten Fällen vollautomatisch implementierbar. Zu diesen Fällen gehören einfache Framing-Situationen, also Pakete, die neben einer festen Struktur höchstens eine *any-type*-Komponente enthalten (siehe Abb. 5-23). Bezogen auf diese *any-type*-Komponente (N-SDU) besteht das Paket somit aus einem Header fester Größe (*n* Bytes), einem Trailer fester Größe (*m* Bytes) und dazwischen der *any-type*-Komponente variabler Größe. Es ist offensichtlich, wie ein solches Paket (mit *x* Bytes Größe) ohne weitere Zusatzinformationen in seine drei Komponenten zerlegt werden kann, indem die Größe und Position des *any-type*-Anteils variabler Größe in der Serialisierung anhand der Werte *n*, *m* und *x* ermittelt werden (siehe Abb. 5-23).

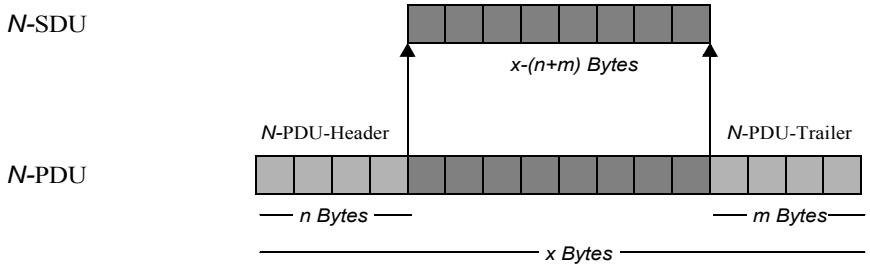


Abbildung 5-23: Automatische Paketdekomposition bei einfachen Framing-Szenarien

In komplexeren Szenarien, in denen zwei oder mehr *any-type*-Komponenten in einem Paket enthalten sind, sind dagegen explizit zugefügte oder implizit aus dem Paket gewonnene Zusatzinformationen notwendig, um die Dekomposition durchführen zu können (siehe Abb. 5-24).

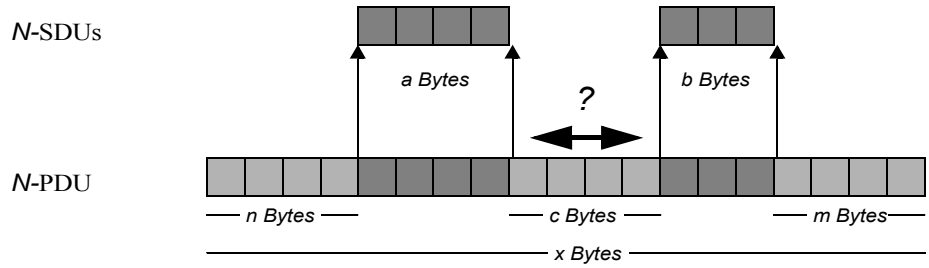


Abbildung 5-24: Paketdekomposition bei komplexen Framing-Szenarien

Die Gewinnung und Auswertung solcher Zusatzinformationen kann dabei entweder auf Implementierungsebene oder (zumindest teilweise) auf Spezifikationsebene erfolgen. So können auf Spezifikationsebene verfügbare Kontextinformationen (z. B. aus der Interpretation von Paket-

headern gewonnene Größenangaben von *any-type*-Fragmenten) zur Parametrierung solcher Extraktionsvorgänge genutzt werden. Die Extraktion selbst könnte dann über eine PRIMITIVE Funktion mit folgender Schnittstelle erfolgen:

```
FUNCTION extract(at: ANY TYPE, idx: INTEGER, len: INTEGER): ANY TYPE;
    PRIMITIVE;
```

Diese Funktion liefert aus dem *any-type*-Objekt at ein enthaltenes Teilobjekt beginnend an Index idx mit der Länge len. So können die beiden in dem in Abb. 5-24 dargestellten Paket enthaltenen *any-type*-Objekte durch folgende Funktionsaufrufe extrahiert werden:

```
SDU1 := extract(PDU, n, a);
SDU2 := extract(PDU, n+a+c, b);
```

Voraussetzung dazu ist natürlich die Kenntnis der Größen und Offsets dieser Teilobjekte innerhalb des zu dekodierenden Datenobjekts. Diese Größen können entweder über pakettypspezifische primitive Funktionen kontextfrei oder aus (z. B. explizit übertragenen) Kontextinformationen auf Spezifikationsebene ermittelt werden. Im letzteren Fall ist beim Erzeuger des Paketes der Einsatz einer PRIMITIVE-Funktion zur Ermittlung der Größe eines *any-type*-Objekts erforderlich:

```
FUNCTION sizeof(at: ANY TYPE): INTEGER;
    PRIMITIVE;
```

Wir wollen an dieser Stelle die Möglichkeiten zur Kodierung und Dekodierung serialisierter Darstellungen von *any-type*-Objekten nicht weiter vertiefen, da diese Aspekte prinzipiell außerhalb der Zielsetzungen formaler Beschreibungstechniken wie Estelle liegen. Es sollte lediglich demonstriert werden, dass neben der Verwendung automatisch implementierbarer Kodierungen durch das Hinzufügen von Zusatzinformationen (wie z. B. die Basic-Encoding-Rules bei ASN.1) und einer vollständig auf Implementierungsebene abgewickelten Kodierung (z. B. durch primitive Encode- und Decode-Funktionen für die einzelnen PDU-Typen) durchaus wesentliche Teilaspekte der Kodierung und Dekodierung auch auf Spezifikationsebene gesteuert werden können.

5.5. Übertragbarkeit auf andere FDTs

Zuletzt betrachten wir noch die Frage nach der *Übertragbarkeit* der vorgestellten Erweiterung auf andere formale Beschreibungstechniken. Grundsätzlich ist die Grundidee der Containertypen und der vorgestellten Konvertierungsregeln unabhängig von einer konkreten FDT und sollte insofern uneingeschränkt übertragbar sein. In einigen FDTs existieren jedoch bereits alternative Möglichkeiten zur Typabstraktion, die teilweise die der Einführung der Containertyp-Erweiterung zu Grunde liegenden Problemstellungen bereits lösen. Als Beispiel betrachten wir hier zunächst die Integration von ASN.1 in SDL-92.

In SDL (SDL-92, [ITU94]) bestehen viele der in diesem Kapitel anhand von Estelle motivierten Probleme bzgl. der Typabstraktion bei Protokollschnittstellen in ähnlicher Weise, und tatsächlich scheint eine Portierung der vorgestellten Containertyp-Erweiterung nach SDL möglich und sinnvoll.

Aufgrund der Integration von ASN.1 [ISO88a] in SDL steht hier jedoch eine Alternative zur Typabstraktion und Serialisierung von Protokolldaten bereits zur Verfügung. Mit dieser können letztlich insbesondere mit den strukturierten Typen *ANY*, bzw. *Open Type* [ISO00] ähnliche Typabstraktionen realisiert werden.

Hierbei ist jedoch zu bedenken, dass der Einsatz von ASN.1 im Vergleich zur Containertyp-Erweiterung eine deutlich höhere *Detaillierung* der Zugriffs- und Konvertierungsoperationen erfordert und somit ein *geringeres Abstraktionsniveau* bietet. ASN.1 kommt aus konzeptioneller Sicht jedoch als mögliche Implementierung der Containertyp-Erweiterung in Frage (siehe auch Abschnitt 5.4.3). Dazu bietet sich der Zwischenschritt einer Umwandlung von einer abstrakten Spezifikation auf Basis der Containertyp-Erweiterung in eine Implementierungsspezifikation auf Basis von ASN.1 an.

In SDL-2000 [ITU00] sind mit der Einführung der Objektorientierung von Datentypen zusätzlich möglicherweise auch Objekthierarchien zur Typabstraktion nutzbar. Insgesamt ist zu einer endgültigen Bewertung der einzelnen Konzepte jedoch eine genauere vergleichende Untersuchung der Effektivität der verschiedenen Abstraktionsmittel erforderlich.

5.6. Zusammenfassung

Gegenstand des voranstehenden Kapitel 5 war die Fragestellung, wie Datenübertragungsdienste *anwendungsneutral*, *wiederverwendbar* und *problemorientiert* mittels formaler Beschreibungstechniken spezifiziert werden können. Dazu haben wir zunächst am Beispiel von Estelle die bestehenden Möglichkeiten zur Spezifikation solcher Datenübertragungsdienste untersucht und dabei festgestellt, dass die Typsicherheit formaler Beschreibungstechniken der gestellten Aufgabe auf verschiedenen Ebenen entgegensteht.

So entwickelt sich bei einem hierarchisch strukturierten Kommunikationssystem bereits bei einer fest vorgegebenen Menge von Dienstnutzern mit heterogenen PDUs auf der obersten Ebene eine komplexe Dienstschnittstelle, die durch eine komplexe *horizontale Typstruktur* (siehe Abschnitt 5.1.4) geprägt ist. Orthogonal dazu bewirkt die Abbildung der in Datenübertragungsprotokollen praktisch allgegenwärtigen Framing-Operationen auf die statisch definierte Aggregation von SDU-Typen in PDU-Strukturtypen eine zu den Basisdiensten hin zunehmend verschachtelte *vertikale Typstruktur* an den Dienstschnittstellen, die insbesondere gegenläufig zur Komplexität der geleisteten Dienste verläuft (siehe auch Abb. 5-7 auf Seite 192). Diese beiden Effekte verstärken sich gegenseitig und führen letztlich zu Dienstschnittstellen mit exponentieller Typkomplexität.

Entsteht bei geschlossenen Spezifikationen mit einer fest vorgegebenen Menge von Protokollkomponenten und Dienstnutzern auf diese Weise schnell eine statische Typstruktur erheblicher Komplexität, so zeigt sich bei der Spezifikation von Diensterbringern oder Protokollkomponenten als *offene Systeme* (siehe *Open-Estelle* in Kapitel 7), dass mit den in Estelle verfügbaren Ausdrucksmitteln eine anwendungsneutrale und somit wiederverwendbare Beschreibung solcher Komponenten schlichtweg nicht möglich ist.

Zur Lösung dieses Problems haben wir als eines der wesentlichen Ergebnisse von Kapitel 5 die syntaktische und semantische Estelle-Erweiterung „*Containertyp*“ eingeführt (siehe Abschnitt 5.2.2), die neben dem neuen Datentyp „*any-type*“ insbesondere auf speziellen impliziten *Konvertierungsregeln* zwischen diesem Typ und anderen Typen beruht. Die Grundidee dieser Konvertierungsregeln ist dabei, dass jeder (nicht „pointer-containing“) Typ zuweisungskompatibel zum *any-type* ist, die Rückkonvertierung eines so entstandenen *any-type*-Wertes in einen anderen Typen jedoch ausschließlich dann zulässig ist, wenn der *any-type*-Wert durch Konvertierung aus genau diesem Zieltyp entstanden ist. Die in diesem Sinne zulässigen Konvertierungen sind offensichtlich *typerhaltend* und damit unabhängig von einer konkreten Implementierung oder Typkodierung (d.h. im mathematischen Sinne) *semantisch eindeutig.*

Durch den Einsatz des Containertyps als SDU-Typ der Dienstschnittstelle ist gerade im Bereich hierarchisch strukturierter Datenübertragungsdienste eine Trennung zwischen den für Dienstnutzer und Diensterbringer gleichermaßen relevanten Aspekten der *gemeinsamen Schnittstelle* (die Menge der zulässigen Interaktionen) und den nur für die jeweiligen Dienstnutzer relevanten Aspekte (Struktur und Inhalt der durch den Dienst zu übertragenen SDU) möglich. Die Anwendbarkeit dieser Abstraktion beruht auf dem Umstand, dass Datenübertragungsdienste meist die zu übertragende SDU lediglich *uninterpretiert* und *unverändert* von einem Dienstzugangspunkt (SAP) zu einem anderen transportieren. Entsprechend kann während des Transports von Inhalt und Struktur der SDU vollständig abstrahiert werden. Erst beim Empfänger-Dienstnutzer (in hierarchisch strukturierten Kommunikationssystemen ist dies die Partnerprotokollinstanz des Sender-Dienstnutzers) wird mit Hilfe der Kenntnis der Struktur der übertragenen SDU durch Rückkonvertierung in den Ausgangstyp der Zugriff auf den Inhalt der SDU ermöglicht. Diese Vorgehensweise entspricht zunächst vollständig den Gegebenheiten bei realen Datenü-

bertragungsdiensten, die als SDU eine Struktur-unbekannte Byte-Sequenz übertragen. Erst beim Empfänger kann unter Kenntnis der vom Sender erwarteten Paketstruktur diese Byte-Sequenz strukturiert und interpretiert werden.

Im Gegensatz zu realen Datenübertragungsdiensten *abstrahiert* jedoch die Containertyp-Erweiterung *vollständig von der konkreten Transportkodierung* und von den damit verbundenen Interoperabilitäts-Fragestellungen (Basistypkodierung, Byteorder, Alignment, etc.). Stattdessen wird die Existenz geeigneter Kodierungen auf die Implementierungsebene verlagert (z. B. ASN.1). Dies entspricht der Zielsetzung der FDTs, auf Spezifikationsebene eine mathematisch-abstrakte und implementierungsunabhängige Systembeschreibung zu erzielen und steigert somit das Abstraktionsniveau der Spezifikation.

Ein bemerkenswerter Effekt der *any-type*-Abstraktion der SDU- und PDU-Typen an den Dienstschnittstellen von hierarchisch strukturierten Kommunikationssystemen ist zudem, dass der konkrete SDU-Typ ein *internes Geheimnis des jeweiligen Dienstnutzers* bleiben kann. So können insbesondere die PDU-Typen von Protokollmaschinen als interne Typdefinitionen des entsprechenden Estelle-Moduls spezifiziert werden. Dies verbessert die Einhaltung des *Geheimnisprinzips* zwischen den verschiedenen Protokollkomponenten ganz wesentlich, da die nach außen exportierten Definitionen und insbesondere die resultierenden oberen und unteren Dienstschnittstellen jeder Protokollkomponente auf die für die jeweiligen Kommunikationspartner relevanten Aspekte reduziert werden können. Eine Folge dieser Abstraktion ist die Beobachtung, dass in vielen Fällen durch die beschriebene *any-type*-Abstraktion der SDU- und PDU-Typen die Dienstschnittstellen der verschiedenen vertikalen Schichten zueinander sehr *ähnlich* – wenn nicht sogar *identisch* – werden.

Die Effektivität der eingeführten Erweiterung bei der anwendungsneutralen Spezifikation von Datenübertragungsdiensten haben wir anhand verschiedener Beispiele aus dem Bereich praktisch eingesetzter Protokolle (u. a. IPv4 und IPv6) demonstriert. Dabei haben wir auch das Konzept der *sukzessiven Datenkomposition und -dekomposition* (siehe Abschnitt 5.3.2) auf der Basis der Containertyp-Erweiterung realisiert. Dadurch kann die Komposition und Dekomposition dynamisch strukturierter Pakete, wie man sie z. B. in IPv6 oder XTP vorfindet, vollständig auf der Ebene der formalen Beschreibungstechnik spezifiziert werden. Insbesondere konnte gezeigt werden, dass der für die Typsicherheit relevante korrekte Einsatz der Rückkonvertierung von *any-type*-Werten nicht nur in einfachen Datenübertragungsszenarien, sondern auch in komplexen *dynamischen* und sogar *rekursiven Typstrukturen* möglich ist.

Ein wichtiger Faktor für den praktischen Einsatz der Containertyp-Erweiterung ist auch ihre *automatische Implementierbarkeit* (siehe Abschnitt 5.4). Dazu haben wir aufbauend auf einer einfachen Referenz-Implementierungs-Methode die zur Implementierung in XEC eingesetzte Klassenhierarchie vorgestellt, die durch geschickten Einsatz der in C++ verfügbaren Mittel zur impliziten Typparametrierung von Methoden-Templates eine *vollständig typsichere Implementierung der Containertyp-Erweiterung* nahezu ausschließlich auf Ebene der XEC-Laufzeitbibliotheken realisiert. Diese Implementierung kann auch auf Grund der Typsicherheit auf Implementierungsebene unzulässige Rückkonvertierungen von *any-type*-Werten zur Laufzeit eindeutig erkennen.

6. Effiziente Datenübertragung

Die bereits in Kapitel 4 diskutierten Optimierungsansätze konzentrierten sich auf die Fragestellung, wie der *Kontrollfluss* formal beschriebener Protokolle automatisch effizient implementiert werden kann. Dabei haben wir bisher die Fragestellung der *effizienten Datenübertragung* explizit ausgeschlossen, indem wir in unseren Betrachtungen und Experimenten verhältnismäßig kleine Datenobjekte (typischerweise nur wenige Bytes) als Nutzdatentypen eingesetzt haben. Dadurch wurde der Aufwand für die Übertragung dieser Nutzdaten so weit reduziert, dass er keinen nennenswerten Einfluss auf die Gesamtperformance des Systems hatte. Stattdessen bestimmte letztlich nur die (mehr oder weniger) effiziente Auswahl und Ausführung von Modulinstanzen und den enthaltenen Transitionen des formal beschriebenen endlichen Automaten die Leistung des Systems.

In diesem Abschnitt beschäftigen wir uns nun mit der Fragestellung, welchen Einfluss der *Aufwand zur Übertragung von Nutzdaten* auf die Systemperformance hat, wenn die Menge der zu übertragenden Daten und somit auch der zu ihrer Handhabung erforderliche zeitliche Aufwand in die Größenordnungen von praktisch eingesetzten Datenübertragungsprotokollen vorstößt. Wir werden dabei sehen, dass die naive Implementierung der auf einer Copysemantik beruhenden Datenübertragungsmechanismen von formal beschriebenen Protokollen häufig bereits bei Nutzdatengrößen von einigen Kilobyte dazu führt, dass der zeitliche Aufwand für die Handhabung der Daten bei weitem den Aufwand für die Abwicklung des zu Grunde liegenden Kontrollflusses überschreitet (siehe auch [ThGo97b]).

Dieses Ungleichgewicht wird natürlich durch die Optimierung des Kontrollflusses noch zusätzlich verschärft. Umgekehrt relativiert der Overhead bei der Übertragung großer Datenmengen letztlich aber auch den möglichen Gewinn aus der Optimierung des Kontrollflusses erheblich, sobald nur noch ein Bruchteil der gesamten Ausführungszeit auf letzteren verwendet wird.

Als Grundlage für unsere späteren Optimierungsversuche betrachten wir in Abschnitt 6.1 zunächst, welche *Daten-Operationen* überhaupt *in formalen Spezifikationen von Kommunikationsprotokollen* typischerweise vorzufinden sind und inwieweit diese Einfluss auf die Performance möglicher Implementierungen haben. Dazu untersuchen wir dann in Abschnitt 6.2 eine semantiknahe Referenzimplementierung der Daten-Operationen formaler Beschreibungstechniken, die schließlich in die Entwicklung eines künstlichen Datenübertragungsbenchmarks mündet, mit dem die Effizienz dieser und weiterer Implementierungsmethoden experimentell ermittelt oder abgeschätzt werden kann.

Als Basis für spätere Optimierungen der Datenübertragung in automatisch generierten Implementierungen betrachten wir dann in Abschnitt 6.3 in *manuell erstellten Implementierungen* vorzufindende Optimierungsmethoden. Wir werden dabei sehen, dass manuell erstellte Implementierungen dabei globale Koordinationsstrategien realisieren, die für eine automatische Implementierung der auf der Copy-Semantik beruhenden Datenübertragung in formal beschriebenen Protokollen nur bedingt geeignet sind. Deshalb untersuchen wir in Abschnitt 6.4 nicht nur

die Optimierung der Datenübertragung in automatisch generierten Implementierungen *beliebiger* Standard-Estelle-Implementierungen, sondern wir betrachten auch den möglichen Gewinn für die automatische Implementierung durch die Einhaltung spezifischer *Spezifikationsstile* oder sogar *Erweiterungen der FDT Estelle*. Gerade im letzteren Fall steht neben der reinen Performance der gewonnenen Implementierungen auch das dabei erreichbare Abstraktionsniveau der zu Grunde liegenden formalen Protokollspezifikation im Mittelpunkt unserer Betrachtungen.

6.1. Daten-Kopieroperationen auf Spezifikationsebene

Automatenbasierte formale Beschreibungstechniken wie z. B. Estelle oder SDL strukturieren das zu beschreibende System typischerweise in eine Menge von abgeschlossenen, unabhängig voneinander agierenden *Komponenten*, die in erster Linie durch die Übertragung asynchroner *Nachrichten* (Interaktionen) miteinander kommunizieren.[1]

Zusammen mit solchen Nachrichten können Daten (als *Parameter*) übertragen werden, deren Werte beim Versenden festgelegt werden und dem Empfänger später zum Zeitpunkt der Entgegennahme bereitstehen. Bei diesem Vorgang wird semantisch im Augenblick des Verschickens der Nachricht der Inhalt der Nachrichtenparameter endgültig auf den Wert der aktuellen Parameter festgelegt, sodass dieser Wert unverändert bis zum Empfänger übertragen wird, der Sender nach dem Verschicken also keinen direkten Einfluss mehr auf die Werte der Nachrichtenparameter nehmen kann.[2]

Man bezeichnet diese Semantik als *Copy-Semantik*, da mit dem Verschicken der Interaktion eine Kopie des Wertes der aktuellen Parameter erzeugt wird. So hat im folgenden Estelle-Spezifikations-Fragment die Zuweisung eines neuen Wertes (2) auf die Variable `i` nach dem Verschicken der Interaktion keinen Einfluss mehr auf den als Interaktions-Parameter an den Empfänger übertragenen Wert (1).

Beispiel 6.48: Copy-Semantik der Parameterübergabe an Interaktionen

```
TRANS
     VAR i: INTEGER;
     BEGIN
     i := 1;
     OUTPUT IpToReceiver.Message(i);
     i := 2;
     (* ... *)
     END;
```

(Ende von Beispiel 6.48)

Dieser Vorgang ist vergleichbar mit der Übergabe von Funktions- bzw. Prozedur-Parametern „*by-value*“ oder der Zuweisung des Inhalts einer Variablen auf eine andere. In all diesen Fällen wird aus semantischer Sicht jeweils eine vollständige Kopie des übergebenen Datums erzeugt, die anschließend unabhängig von Manipulationen des Quell-Datums bleibt. Wir bezeichnen dies im Folgenden daher als *semantische Kopieroperation*.[3]

---

1. Mit der Kommunikation über gemeinsame Zustandsraumkomponenten (*shared Variables* bzw. *shared Memory*) beschäftigen wir uns in Abschnitt 6.4.

2. Der Versender kann bestenfalls noch *indirekt* den Empfang der Nachricht beeinflussen, z. B. in Estelle durch Auftrennen einer Verbindung oder Termination der empfangenden (Kind-) Modulinstanz. Eine Manipulation der einmal übergebenen Werte einer Interaktion ist jedoch im Nachhinein nicht mehr möglich.

3. Der Ausdruck „*semantische Kopieroperation*“ impliziert dabei nur indirekt (eben „semantisch“) eine spezifische Implementierungsmethode. Wir werden im Folgenden untersuchen, inwieweit solche Operationen unter Vermeidung tatsächlicher Kopieroperationen implementiert werden können.

Um später die Auswirkungen semantischer Kopieroperationen auf die Effizienz von abgeleiteten Implementierungen abschätzen zu können, untersuchen wir zunächst, in welchen Szenarien und in welchem Umfang diese typischerweise in formalen Spezifikationen von Datenübertragungsprotokollen auftreten.

6.1.1 Semantische Kopieroperationen in Protokollmaschinen

Wie wir bereits in Kapitel 5 gesehen haben, werden Kommunikationssysteme meist in *Diensthierarchien* strukturiert. Dabei wird auf jeder Ebene ein *Dienst* von einem *Diensterbringer* aufbauend auf den Diensten darunter liegender Schichten erbracht. In Kommunikationssystemen ist dieser Diensterbringer meist eine Protokollmaschinen-Instanz, die in Zusammenarbeit mit einer adäquaten Protokollmaschinen-Partnerinstanz an den oberen Dienstschnittstellen jeweils einen spezifischen Dienst bereitstellt. Somit ergibt sich die in Abb. 6-1 dargestellte Hierarchie von Diensten und sie erbringenden Protokollautomaten.

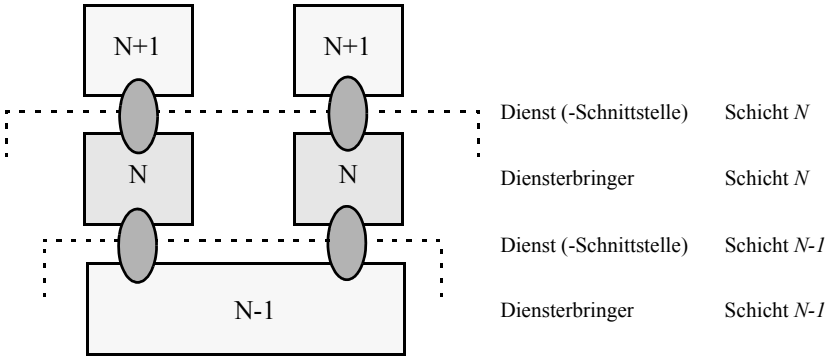


Abbildung 6-1: Diensthierarchien zur Strukturierung von Kommunikationssystemen

Eine nahe liegende Abbildung dieser Organisation in die Strukturierungsmittel formaler Beschreibungstechniken besteht darin, die einzelnen Protokollmaschinen als jeweils abgeschlossene Objekte zu modellieren, die mit ihrer Umgebung über eine (bezüglich der zu lösenden Aufgabe) möglichst minimale äußere Schnittstelle kommunizieren. Diese Schnittstelle wird dabei naheliegenderweise gerade durch die oberen und unteren Dienstschnittstellen definiert (siehe auch Abschnitt 5.1). Eine zur *problemorientierten* Spezifikation angemessene Modellierung einer solchen Protokollmaschine in Estelle besteht also darin, die Protokollmaschine selbst als ein *Modul* darzustellen, das zu den oberen bzw. unteren Dienstschnittstellen über *externe Interaktionspunkte* kommunizieren kann.

Bei komplexeren Protokollmaschinen ist es zur weiteren Strukturierung oft sinnvoll, durch die Einführung von (lokalen) Kindmodulen auch interne Funktionalitäten strukturell abzugrenzen, um so deren Interaktionen explizit auf ihre gemeinsame Modulschnittstelle zu beschränken. Dieser Strukturierungsansatz wurde z. B. bei der Erstellung der formalen XTP-Spezifikation eingesetzt (siehe Abschnitt 2.3.4). So strukturiert sich eine XTP-Protokollmaschinen-Modulinstanz intern neben einem Multiplexer-Modul (Ⓧ in Abb. 6-2) noch in jeweils eine dynamisch erzeugte Modulinstanz („Assoziation“, siehe Ⓐ in Abb. 6-2) pro gerade aktiver Verbindung, wobei jede Assoziation nochmals in vier Kindmodulinstanzen unterstrukturiert ist, die ebenfalls jeweils voneinander getrennte Funktionalitäten des Protokolls in Bezug auf die jeweilige Assoziation abwickeln.

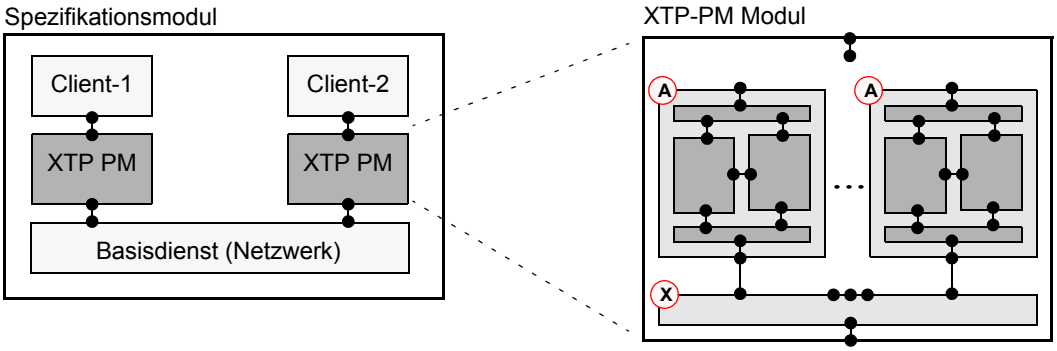


Abbildung 6-2: Die Modulinstanz- und Verbindungsstruktur der Estelle-Spez. von XTP

Die Motivation für eine derart komplexe Unterstrukturierung ist zum einen eine angemessene Modellierung der dynamischen Aspekte der Protokollabwicklung (hier: dynamische Erzeugung und Termination von Assoziations-Modulinstanzen), zum anderen aber auch der Versuch einer Nachbildung der im XTP-Standard [XTP95] vorgegebenen Strukturierung der Protokollmaschine (hier: Unterstruktur der Assoziations-Module (A)). Dies senkt die Komplexität der Spezifikation, erhöht die Verfolgbarkeit im Entwicklungsprozess und steigert somit Wartbarkeit und Zuverlässigkeit der Spezifikation. Vor diesem Hintergrund ist die vorgefundene Struktur gerade für eine *problemorientierte* Spezifikation[4] des Protokolls angemessen.

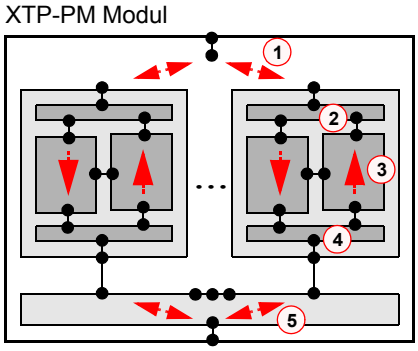


Abbildung 6-3: Semantische Datenkopieroperationen in der XTP-PM (Estelle-Spez.)

Durch diese Unterstrukturierung des XTP-Protokollmaschinenmoduls wird beim Versenden bzw. Empfangen eines Datenpaketes der Inhalt des Datenpaketes in beiden Richtungen jeweils mindestens fünfmal als Interaktions-Parameter empfangen und anschließend weiter verschickt (siehe (1) bis (5) in Abb. 6-3). Ergänzt man nun diese Protokollmaschine zu dem in Abb. 6-2 (links) dargestellten kompletten Kommunikations-Szenario mit zwei XTP-Dienstnutzer-Modulen (Client-1 und Client-2), den beiden XTP-Protokollmaschinen-Partnermodulinstanzen (XTP PM) und einer Modulinstanz zur Realisierung eines Basisdienstes, über den die beiden Partnermodulinstanzen kommunizieren können, so ergeben sich für eine Ende-zu-Ende-Kommunikation noch weitere Sendeoperationen:

4. Die zu Grunde liegende Fassung der Estelle-Spezifikation von XTP wurde ausschließlich als problemorientierte Spezifikation erstellt, ohne auf Implementierungsaspekte Rücksicht zu nehmen (siehe auch Abschnitt 2.3.4.2).

- Eine minimale Realisierung der Dienstnutzer-Module erfordert zur Durchführung einer realistischen Kommunikation über den XTP-Dienst zumindest das Versenden einer Sendeauftrags-Interaktion mit dem zu verschickenden Datum (z. B. aus einer lokalen Variablen) als Parameter und den Empfang einer analogen Empfangs-Interaktion bei der Partnerinstanz des Dienstnutzers, die als Parameter das ursprünglich verschickte Datum enthält.
- Eine minimale Realisierung des Basisdienstes besteht darin, die als Basisdienst-SDU-Parameter einer Interaktion von den XTP-Protokollmaschinen übergebenen Datenpakete durch Feuern einer Transition zu empfangen und anschließend (wiederum als Interaktionsparameter) an die jeweils andere XTP-Protokollmaschine zu übertragen.[5]

Somit ergeben sich für eine komplette Ende-zu-Ende-Übertragung eines Nutzdatenpakets insgesamt 14 Sende- und Empfangereignisse mit den zu übertragenden Nutzdaten als Parametern. Im nächsten Abschnitt werden wir untersuchen, inwieweit innerhalb von Protokollmaschinen neben den genannten semantischen Kopieroperationen aufgrund der Übertragung von Daten als Interaktionsparameter noch weitere explizite Kopieroperationen auftreten.

6.1.2 Explizite semantische Kopieroperationen in Protokollmaschinen

Die explizite Spezifikation von Kopieroperationen findet man in Protokollmaschinen im Wesentlichen in zwei typischen Situationen: dem *Framing* bzw. *Unframing* von Paketen und bei der *Zwischenspeicherung* von Nutzdaten über die Transitionsausführung hinweg.

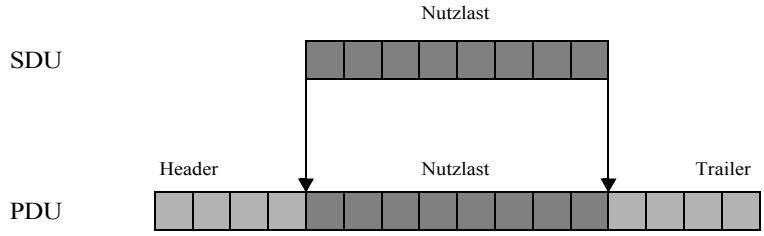


Abbildung 6-4: Framing einer SDU in eine PDU durch Umkopieren der SDU

Als *Framing* bezeichnen wir dabei den Vorgang der Einbettung einer vom Dienstnutzer übergebenen SDU (als Nutzlast) in eine PDU zum Zwecke des Weitertransports des entstehenden Paketes durch untergeordnete Diensterbringer (siehe auch Abschnitt 5.1). Dabei entsteht die PDU (siehe Abb. 6-4) typischerweise als Konkatenation

(i) eines Paketheaders,

(ii) der zu transportierenden Nutzlast und

(iii) eines Pakettrailers.

In Estelle erfolgt dieser Einbettungs-Prozess – sofern die resultierende PDU als ein geschlossenes Datenobjekt modelliert werden soll – durch Anlage einer Hilfsvariablen vom PDU-Typ, welcher als strukturierter Typ den SDU-Typ als Komponente enthält (siehe Beispiel 6.49-a). Dieser SDU-Komponente der Hilfsvariablen wird dann der einzubettende SDU-Wert explizit

5. Diese Lösung ist vom Aufwand minimal, wenn man von einer ggf. möglichen direkten Kopplung der beiden externen Interaktionspunkte des Basisdienstes (durch `ATTACH`- und `CONNECT`-Operationen auf zwei internen Hilfs-Interaktionspunkten) absieht, wodurch der Basisdienst jedoch jeder Eingriffsmöglichkeit in die Dienst-Qualität der Datenübertragung beraubt ist.

zugewiesen. Nachdem auch die übrigen Komponenten der PDU-Struktur initialisiert wurden, steht in Form dieser Hilfsvariablen der geforderte PDU-Wert in geschlossener Form bereit, um als Interaktions-Parameter weiter verschickt zu werden.

Beispiel 6.49-a: Framing von SDUs (Estelle-Fragment)

```
TYPE SDU_T = {...};
TYPE PDU_T = RECORD
               header: INTEGER;
               payload: SDU_T;
               trailer: INTEGER;
             END;
{...}

TRANS
  WHEN ipup.msg{sdu: SDU_T}
    VAR pdu: PDU_T;
    NAME frame:
      BEGIN
      pdu.header  := {...};
      pdu.payload := sdu;
      pdu.trailer := {...};
      OUTPUT ipdown.msg(pdu);
      END;
```

(Ende von Beispiel 6.49-a)

Dieser Framing-Schritt beinhaltet in Form der Zuweisung des SDU-Wertes an die entsprechende Komponente der PDU-Hilfsvariablen eine *explizite Kopieroperation* der Nutzlast. Der umgekehrte Vorgang, also das Unframing zur Extraktion eines SDU-Wertes aus einem vom untergeordneten Diensterbringer als Interaktionsparameter übergebenen PDU-Wert kann hingegen ohne explizite Kopieroperationen in Estelle spezifiziert werden, da in diesem Fall direkt auf den Inhalt der SDU-Komponente des PDU-Interaktionsparameters zugegriffen werden kann, um sie z. B. wiederum als Interaktionsparameter an den Dienstnutzer weiterzugeben:

Beispiel 6.49-b: Unframing von SDUs (Estelle-Fragment)

```
TRANS
  WHEN ipdown.msg{pdu: PDU_T}
    NAME unframe:
      BEGIN
      { ... evaluate pdu.header and pdu.trailer ... }
      OUTPUT ipup.msg(pdu.payload);
    END;
```

(Ende von Beispiel 6.49-b)

Da derartige Framing- und Unframing-Operationen typischerweise paarweise in den zusammengehörigen Partnerprotokollinstanzen eines Kommunikationssystems auftreten, erfordert die Übertragung einer Nutzlast von Ende zu Ende für jedes Framing-/Unframing-Paar im Protokollstack jeweils eine zusätzliche semantische Kopieroperation.

Die zweite häufig anzutreffende Situation, in der in formalen Protokollspezifikationen explizite semantische Kopieroperationen auftreten, ist die *Zwischenspeicherung* von (als Interaktionsparameter empfangenen) Nutzdaten über einen Zeitraum,[6] der über die Ausführung der Emp-

fangstransition selbst hinausgeht. Dies wird z. B. eingesetzt, um Interaktionsparameter nicht sofort in der Empfangstransition vollständig weiter zu verarbeiten, sondern sie zunächst in einer Variablen zwischenzuspeichern und zu einem späteren Zeitpunkt weiter zu verarbeiten bzw. weiter zu transportieren.[7]

Eine wichtige Variante dieser Situation ergibt sich in Kommunikationsprotokollen, die u. U. ein mehrfaches Verschicken der selben Nutzdaten über einen längeren Zeitraum erfordern, wie es z. B. bei der *Wiederholung* von möglicherweise bei der Übertragung verloren gegangener Pakete notwendig ist. Eine Protokollmaschine, die aufbauend auf einen in dieser Hinsicht unzuverlässigen Datenübertragungsdienst einen verlustfreien Dienst leisten soll, muss die vom Dienstnutzer übergebenen zu transportierenden Nutzdaten nach dem ersten Verschicken (als Teil eines Paketes) zusätzlich lokal zwischenspeichern, um – z. B. beim Ausbleiben einer rechtzeitigen Quittierung des Empfangs dieses Paketes – diese Nutzdaten erneut übertragen zu können.

So wird in Beispiel 6.50-a nach dem (erstmaligen) Versenden der PDU an den Diensterbringer diese PDU (und damit auch die darin enthaltenen Nutzdaten) explizit in einen Zwischenpuffer kopiert, der eine modullokale Variable ist und somit auch nach Ausführung der Transition erhalten bleibt.

Beispiel 6.50-a: Paket-Pufferung (Estelle-Fragment)

```
VAR Buffer: ARRAY [1 .. Buffer_Size] OF PDU_T;

TRANS
  WHEN ipup.msg{sdu: SDU_T}
    VAR pdu: PDU_T;
    NAME send_first:
      BEGIN
      pdu := {...};
      OUTPUT ipdown.msg(pdu);
      Buffer[{...}] := pdu;
      END;
```

(Ende von Beispiel 6.50-a)

In Beispiel 6.50-b ist eine Transition skizziert, die beim Ausbleiben der erwarteten Quittierung nach Ablauf des Delay-Timers eine wiederholte Versendung der PDU realisiert. Offensichtlich ist dazu keine weitere explizite semantische Kopieroperation erforderlich, da die beim erstmaligen Versenden zwischengespeicherten PDUs unverändert wiederholt werden und daher direkt als Interaktionsparameter dienen können. Werden dagegen die SDUs zwischengespeichert, so ist ein erneutes Framing mit der damit verbundenen expliziten Kopieroperation erforderlich.

---

6. Der Begriff *Zeit* ist hier im Sinne des sequentiellen Voranschreitens des Systemzustands zu verstehen. Die Estelle-Semantik ordnet dagegen dem Feuern einer Transition de facto keinen echten Zeitraum im Sinne unterschiedlicher Start- und Endzeitpunkte zu, sondern Zeit vergeht *asynchron* zum Voranschreiten des Systemzustands (siehe Abschnitte 5.3.5, 9.6.4 und 9.6.5 von [ISO97]).

7. Interaktionsparameter haben in Empfangstransitionen nur die Gültigkeitsdauer der entsprechenden Transitionsausführung.

Beispiel 6.50-b: Paket-Wiederholung (Estelle-Fragment)

```
TRANS
  PROVIDED({...})
    DELAY({...})
      NAME send_repeat:
        BEGIN
        OUTPUT ipdown.msg( Buffer[{...}] );
        END;
```

(Ende von Beispiel 6.50-b)

Über die beiden vorgenannten Situationen hinaus trifft man in formalen Spezifikationen von Kommunikationsprotokollen gelegentlich auch komplexere explizite Kopieroperationen an, um z. B. die Fragmentierung und Reassemblierung von Paketen oder Kodierungsaspekte von Datentypen bis hin zu kryptografischen Aspekten oder Datenkompression zu spezifizieren. Da solche Aspekte oft nicht angemessen mit den Ausdrucksmitteln formaler Beschreibungstechniken wie Estelle beschrieben werden können, beschäftigen wir uns an dieser Stelle nur am Rande mit der datenübertragungstechnischen Komplexität dieser Operationen (siehe auch Abschnitt 5.1).

Einen Grenzfall in dieser Hinsicht stellt der Paket-Pufferungsmechanismus in der Estelle-Spezifikation der XTP-Protokollmaschine dar. Als Transportschicht-Protokoll setzt XTP einen *Sendepuffer* ein, um – wie oben beschrieben – ggf. den Verlust von Paketen durch Wiederholungen zu kompensieren. Dieser Sendepuffer wird in der Estelle-Spezifikation der XTP-Protokollmaschine zusammen mit einem Empfangspuffer auch genutzt, um eine Entkopplung zwischen der Anzahl der in einer SDU übergebenen Nutzdaten-Bytes und der in einer PDU übertragenen Bytes zu ermöglichen.[8] Dies wird auf Estelle-Ebene dadurch möglich, dass die Struktur der SDU ein Byte-Array ist und so ein byteweises Umkopieren der Nutzdaten-Bytes in den ebenfalls als Byte-Array modellierten Sendepuffer sowie eine Weiterübertragung in den ebenfalls als Byte-Array modellierten Nutzdatenbereich der PDU möglich ist. Die Umkehrung dieses Vorgangs erfolgt analog beim Empfang einer PDU über den ebenfalls als Byte-Array modellierten *Empfangspuffer*.

Die Modellierung dieses Mechanismus in Estelle hat dabei eine nicht unerhebliche Komplexität. Sie gelingt jedoch überhaupt erst auf Grund der spezifischen Strukturierung des SDU-Typs der XTP-Protokollmaschine in der Estelle-Spezifikation als Byte-Array. Eine Bewertung der datenübertragungstechnischen Komplexität des Puffermanagements ist an dieser Stelle auf Grund des sich ergebenden Verwaltungsoverheads[9] des Mechanismus und der fehlenden Zuordnung zwischen der Anzahl der SDUs und PDUs nur schwer möglich. Geht man jedoch davon aus, dass die SDUs jeweils die für eine PDU zulässige Zahl von Nutzdatenbytes enthalten, so ergeben sich für den gesamten Vorgang einer Ende-zu-Ende-Übertragung über den Framing-Aufwand hinaus auf der Sende- und der Empfangsseite jeweils eine zusätzliche explizite semantische Kopieroperation.

---

8. Diese Entkopplung und der damit verbundene Overhead sind in der zum Vergleich herangezogenen XTP-Handimplementierung SandiaXTP (siehe Abschnitt 6.3.5) nicht implementiert.

9. Es ist zu beachten, dass die Byteweise Handhabung der Nutzdaten eine unmittelbare Abhängigkeit der Gesamtkomplexität der Handhabungsoperationen und der zu übertragenden Nutzdatenmenge bedingen. Somit kann – abhängig von Spezifikation und konkreter Implementierung – die Datenverwaltung (z. B. die Berechnung jedes einzelnen Array-Index pro Nutzdatenbyte) eine erheblich höhere Komplexität als die eigentliche Kopieroperation (hier ein einzelnes Byte) besitzen.

6.1.3 Semantische Kopieroperationen in komplexen Kommunikationsszenarien

Bei einer Ausdehnung der in den vorigen Abschnitten beschriebenen Kommunikationsszenarien auf eine *größere Anzahl von Dienstschichten* müssen die Nutzdaten natürlich von den Protokollmaschinen jeder einzelnen Schicht behandelt und schließlich an die (nächsthöhere oder nächstniedrigere) Schicht weitergereicht werden. Entsprechend vergrößert sich die Anzahl der semantischen Kopieroperationen (unter der Annahme vergleichbarer Kopier-Komplexität dieser Protokollautomaten) linear mit der Anzahl der Dienstschicht. So führt jede (nicht triviale) Dienstschicht bezüglich der Ende-zu-Ende-Kommunikation mindestens zwei semantische Kopieroperationen ein, wenn sie beim Empfang einer Nachricht mit Nutzdaten (SDU) von ihrer oberen Dienstschnittstelle eine entsprechende neue Nachricht (PDU) an ihre untere Dienstschnittstelle weiterreicht, bzw. ihre Partnerinstanz beim Empfang dieser PDU den Vorgang in umgekehrter Richtung durchführt. Führt die Dienstschicht zudem ein Framing durch Aggregation der SDU in die PDU durch, so ist mindestens eine weitere (explizite) Kopieroperation erforderlich. Wie wir oben am Beispiel der XTP-Protokollmaschine gesehen haben, kann jedoch auch eine wesentlich größere Anzahl von semantischen Kopieroperationen in jeder einzelnen Dienstschicht erforderlich sein. In entsprechend fein strukturierten Systemen können sich deshalb durchaus Kopieroperationen im hohen zweistelligen Bereich ergeben.

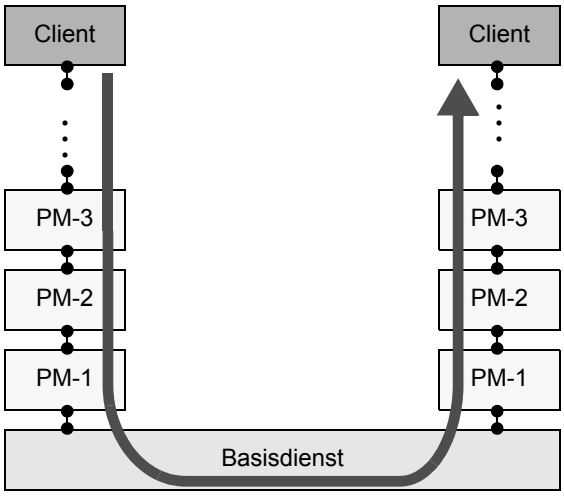


Abbildung 6-5: Einfacher Datentransportpfad bei einheitlichem Basisdienst

Dieser Effekt zeigt sich ganz besonders bei der Modellierung von Vermittlungsschicht-Diensten, da hier Pakete nicht wie in den darüber liegenden Schichten beim Ende-zu-Ende-Transport lediglich durch einen („vertikal“ angeordneten) Protokollmaschinen-Turm zu einem Basisdienst „hinunter“ geleitet werden, um dann durch einen Turm von Partner-Protokollmaschinen direkt wieder nach „oben“ weitergeleitet zu werden (*„V“-Muster*, siehe Abb. 6-5). Auf der Ebene der Vermittlungsschicht wird dagegen der Transport von Datenpaketen typischerweise über mehrere („horizontal“ angeordnete) „Hops“ modelliert (*„W“-Muster*, siehe Abb. 6-6). Die Anzahl der dafür notwendigen semantischen Kopieroperationen hängt dabei linear von der Anzahl der notwendigen Hops ab und kann entsprechend weit über die Anzahl der für die Abwicklung der höheren Protokollschichten notwendigen Kopieroperationen hinausgehen.

Eine weitere Quelle zusätzlicher semantischer Kopieroperationen ergibt sich aus der Korrektur von Paketverlusten und -Verfälschungen aufgrund von unzuverlässigen Diensten. Solche Ereignisse erfordern normalerweise zu ihrer Korrektur die *Wiederholungen* des verlorenen oder

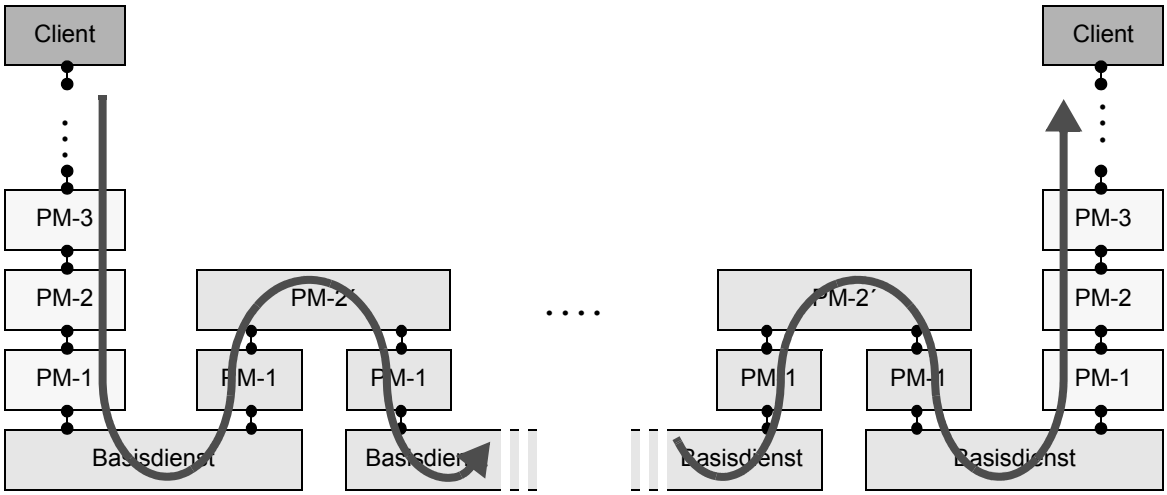


Abbildung 6-6: Datentransportpfad bei Paketvermittlung über mehrere „Hops“

verfälschten Paketes. Je nach Wiederholungs- und Quittierungsstrategie kann beim Verlust eines Paketes sogar die Wiederholung einer größeren Zahl von Paketen notwendig werden. Eine solche Paketwiederholung führt natürlich dazu, dass zur *erfolgreichen* Übertragung eines Paketes zwischen den (die Paketwiederholung initiierenden) Protokollinstanzen die zu übertragenden Nutzdaten mehrfach zum Transport an die darunter liegenden Schichten übergeben und dort weitergereicht werden. Je nach Art und Zeitpunkt der Paketverfälschungen bzw. des Paketverlustes führen auch die erfolglosen Datenübertragungen zu einer erheblichen Anzahl von semantischen Kopieroperationen (siehe Abb. 6-7) und vervielfachen somit die Anzahl der Kopieroperationen, die zur Übertragungen eines Nutzdatenpakets von Ende zu Ende erforderlich ist.

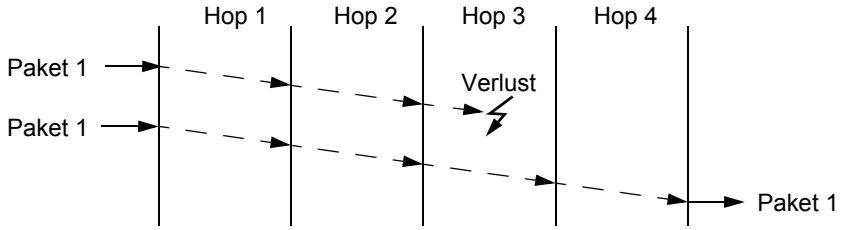


Abbildung 6-7: Mehrfach-Datentransport durch Paketverlust

In den nächsten Abschnitten beschäftigen wir uns daher mit der Frage, wie semantische Kopieroperationen manuell bzw. halb- oder vollautomatisch *effizient* implementiert werden können und welche Kosten diese Realisierungen jeweils verursachen.

6.2. Referenz-Implementierung semantischer Kopieroperationen

In diesem Abschnitt untersuchen wir eine Implementierungsmethode, die auf Grund ihrer universellen Anwendbarkeit, leichten Implementierbarkeit und Nähe zur Estelle-Semantik als *Referenz-Implementierungsmethode* gelten kann. Wir werden dabei jedoch feststellen, dass diese Vorteile mit dem Preis eines erheblichen Laufzeit-Overheads im Vergleich zu gut optimierten Handimplementierungen erkauft werden.

6.2.1 Grundlagen der Referenz-Implementierungsmethode

Die naheliegendste Form der Implementierung von Datentransportmechanismen in formal beschriebenen Protokollen ist die direkte Abbildung semantischer Kopieroperationen auf *physische Kopieroperationen*. Dazu wird bei der Übergabe eines Datums als Interaktionsparameter eine vollständige Kopie des übergebenen Wertes in einem separaten Speicherbereich angelegt.[10] Durch die Anfertigung dieser Kopie, auf die auf Spezifikationsebene zunächst kein Bezug mehr besteht, kann das übergebene Datum völlig unabhängig von den Eigenschaften der jeweiligen Spezifikation unverändert bis zum Empfänger der Interaktion übertragen werden. Insbesondere haben nachträgliche Manipulationen an einem als aktuellen Parameter übergebenen Datenpuffer (z. B. einer Variablen) keinen Einfluss mehr auf den Wert dieser Kopie (siehe Beispiel 6.48 auf Seite 247).

Danach wird der Inhalt der Kopie unverändert bis zum Empfänger der Interaktion transportiert, wo das enthaltene Datum beim Feuern der Annahmetransitionen schließlich als Inhalt einer Interaktions-Parameter-Variablen (interaction-argument-identifier, siehe Abschnitte 7.3.4 und 7.5.6 von [ISO97]) wieder auf Spezifikationsebene zugänglich wird. Da nach dem Feuern dieser Annahmetransition (bzw. dem Verwerfen der Interaktion im Falle des nicht-Empfangs[11]) die Interaktion und die damit verbundenen Daten nicht weiter benötigt werden, können auch die durch den Transportmechanismus erstellten Kopien des ursprünglich übergebenen Wertes (bzw. die zu ihrer Ablage genutzten Speicherbereiche) freigegeben werden.

Ganz offensichtlich ist der beschriebene Implementierungsmechanismus für die Übertragung von Interaktions-Parametern sehr stark an die semantische Definition dieses Vorgangs in Estelle angelehnt. Entsprechend werden an die zu implementierende Spezifikation keine spezifischen Anforderungen gestellt, um die semantikkonforme Anwendung dieses Mechanismus zu ermöglichen. Somit handelt es sich um einen universellen Implementierungsmechanismus für diesen Teilaspekt. Nicht zuletzt deshalb wird der beschriebene Mechanismus von allen bekannten automatischen Implementierungsgeneratoren (einschließlich XEC, siehe Abschnitt 6.2.2) standardmäßig eingesetzt.

Ein wesentlicher Punkt bei diesem Datentransportmechanismus ist, dass eine semantikkonforme Implementierung zunächst nur *genau eine* Kopieroperation erfordert, nämlich zur Sicherung des Wertes des aktuellen Interaktionsparameters zum Zeitpunkt des Verschickens der Interak-

10. Analog werden auch andere semantische Kopieroperationen wie z. B. die Übergabe eines Wertes an eine Funktion „by-value“ implementiert.

11. z. B. durch Senden einer Interaktion in einen nicht verbundenen Interaktionspunkt

tion. In einfachen Implementierungsszenarien ist dieser eine Kopiervorgang auch bereits völlig ausreichend: Arbeiten der Sender und der Empfänger der Interaktion im selben Adressraum, so kann diese Kopie beim Empfang der Interaktion direkt und ohne weitere Kopien als Parameter-Variable eingesetzt werden, indem alle Referenzen auf die Parameter-Variable auf den Speicherplatz der Kopie aufgelöst werden. Es ist dabei zu beachten, dass analog zu formalen „*by-value*“-Parametern in Funktionen auch die formale Parameter-Variable im Kontext der Annahmetransition (interaction-argument-identifier) eine echte Variable ist, die lediglich auf den Wert der beim Verschicken der Interaktion übergebenen aktuellen Parameter *initialisiert* ist. Dies bedeutet, dass innerhalb der Annahmetransitionen der Wert dieser Variablen beliebig verändert werden kann. Da jedoch weder der Originalwert noch der veränderte Wert nach Ausführung der Annahmetransition weiter benötigt werden (siehe oben), ist dies für den beschriebenen Implementierungsmechanismus nicht nachteilig.

In komplexeren Implementierungsszenarien arbeiten Sender und Empfänger einer Interaktion jedoch möglicherweise auch in unterschiedlichen Adressräumen. Existiert innerhalb beider Adressräume ein gemeinsamer Speicherbereich (*shared variables / shared memory*), so kann die oben beschriebene Kopie u. U. direkt in einem solchen gemeinsamen Speicherbereich angelegt werden. In diesem Fall ist auch hier die Realisierung der Übertragung von Interaktionsparametern mit genau einer Daten-Kopieroperation möglich. Existieren keine solchen gemeinsamen Speicherbereiche, so sind meist zur Übertragung wesentlich aufwändigere Mechanismen erforderlich, wie z. B. die Nutzung von Übertragungsdiensten des Betriebssystems zur Kommunikation über Netzwerke. Ein Beispiel hierfür ist die Übertragung von Datenpaketen zwischen verschiedenen Hosts über einen TCP/IP-Netzwerk. Solche Mechanismen sind im Vergleich zu einfachen Datenkopier-Operationen im Hauptspeicher weitaus aufwändiger.

6.2.2 Implementierung der Interaktionsparameterübergabe in XEC

Da die oben beschriebene Referenz-Implementierungsmethode als *universeller Implementierungsmechanismus* unabhängig von den Eigenschaften der zu implementierenden Estelle-Spezifikation immer eine semantikkonforme Implementierung der Datentransportaspekte liefert, wurde sie auch beim Implementierungsgenerator XEC eingesetzt. Wir betrachten im Folgenden, welche Datentransportoperationen bei der Übertragung eines komplexen Datums als Interaktionsparameter dabei erforderlich sind.

Wie wir bereits in Abschnitt 3.3.4 gesehen haben, generiert XEC für jede Kanaldefinition eine C++-Klasse („`CHANNEL_...`“), in der unter anderem pro über den Kanal übertragbarer Interaktion eine Methode („`IA_...`“) definiert wird, mit der das Verschicken einer solchen Interaktion möglich ist. Für Interaktionen mit einer nicht-leeren Parameterliste besitzt diese Methode eine äquivalente formale C++-Parameterliste, in der komplexe Typen per Referenz übergeben werden. Weiterhin enthält die Kanal-Klassendefinition für jede solche Interaktion mit einer nicht-leeren Parameterliste eine Strukturtypdefinition („`struct IA_..._ARG`“), die für jeden Parameter der Interaktion eine äquivalente Strukturkomponente („`PAR_...`“) enthält. Diese Struktur wird später zur Aufnahme der übergebenen Parameter-Daten während des Transports der Interaktion dienen. Es ergibt sich aus der in Beispiel 6.51-a dargestellten Kanaldefinitionen das in Beispiel 6.51-b ausschnittsweise angegebene C++-Codefragment.

Beispiel 6.51-a: Estelle-Kanaldefinition

```
TYPE userdata = ARRAY [1 .. 1000] OF 0..255;
```

```
CHANNEL ch(user1, user2);
BY user1,user2: msg(Arg: integer; data: userdata);
```

(Ende von Beispiel 6.51-a)

Beispiel 6.51-b: Fragment des aus Beispiel 6.51-a generierten C++-Codes

```
class CHANNEL_ch : public IP {
    public:
        struct IA_msg_ARG {
            TYPE_integer PAR_arg;
            TYPE_userdata PAR_data;
        };

        inline void IA_msg (
            const TYPE_integer PAR_arg,
            const TYPE_userdata& PAR_data
        );

        static const unsigned IA_msg_ID = 1;

        typedef InterAction<CHANNEL_ch, IA_msg_ID, IA_msg_ARG>
                IA_msg_TYPE;

    /* ... */
};
```

(Ende von Beispiel 6.51-b)

Wie wir bereits früher gesehen haben, werden Estelle-Interaktionspunkte von XEC direkt als Instanzen der ihnen zu Grunde liegenden Kanal-Klassendefinition realisiert. Entsprechend wird das Verschicken einer Interaktion über einen Interaktionspunkt (z. B. „`ipout`“ in Beispiel 6.52-a) einfach als Methodenaufruf des entsprechenden Interaktionspunkts-Objekts realisiert (siehe Beispiel 6.52-b).

Beispiel 6.52-a: Estelle-Transition mit `OUTPUT`-Statement

```
IP ipout: ch(user2);

TRANS
    VAR d: userdata;
    NAME send:
        BEGIN
        OUTPUT ipout.msg(123, d);
        END;
```

(Ende von Beispiel 6.52-a)

Beispiel 6.52-b: Fragment des aus Beispiel 6.52-a generierten C++-Codes

```
CHANNEL_ch IP_ipout;
/* ... */
```

```
void /* ...:: */ TRANS_send::exec() {
     TYPE_userdata VAR_data;
     parent()->IP_ipout.IA_msg(123, VAR_data);
}
```

(Ende von Beispiel 6.52-b)

In der bereits beschriebenen Methode zum Verschicken von Interaktionen („`IA_msg(...)`“, siehe Beispiel 6.52-c) wird schließlich eine neue Instanz der Template-Klasse „`InterAction<...>`“ (oben als „`IA_msg_TYPE`“ definiert) erzeugt, die auf Grund der Typparametrierung mit dem oben beschriebenen Strukturtyp zur Aufnahme der Parameter-Daten („`IA_msg_ARG`“) in der Komponente „`param`“ die übergebenen Parameterwerte per direkter Zuweisung aufnehmen kann. Da komplexe Typen (wie z. B. „`TYPE_userdata`“ im obigen Beispiel) im Gegensatz zu atomaren Typen per Referenz an die Methode übergeben wurden, bedeutet diese Zuweisung die *einzige Kopieroperation* auf komplexen Typen im Verlauf des gesamten Datentransportprozesses.

Beispiel 6.52-c: Sende-Methode für Interaktion „`msg`“ in C++

```
inline void /* ...:: */ CHANNEL_ch::IA_msg (
     const TYPE_integer PAR_arg,
     const TYPE_userdata& PAR_data
) {
     IA_msg_TYPE* pIA = new IA_msg_TYPE;
     pIA->param.PAR_arg = PAR_arg;
     pIA->param.PAR_data = PAR_data;
     pIA->send(this);
}
```

(Ende von Beispiel 6.52-c)

Schließlich wird durch Aufruf der Methode „`send(...)`“ der Weitertransport des nunmehr vollständigen Interaktionsobjekts (genauer: der Referenz auf das Objekt) in die Zielwarteschlange initiiert. Dieser Transport erfolgt dabei unter völliger Abstraktion des Interaktionstyps und der enthaltenen Parameter, indem die Template-Klasse „`InterAction<...>`“ abstrahiert als Instanz der Basisklasse „`InterActionAbstract`“ gehandhabt wird (siehe Klassendiagramm in Abb. 6-8).

Erst bei der Annahme der Interaktion durch den Empfänger erfolgt unter Ausnutzung der eindeutigen Typidentifikation[12] durch den Rückgabewert der Methode „`typeId()`“ eine Rückkonvertierung der Referenz auf das Objekt in den qualifizierten Typ. Dazu wird bei der Überprüfung der Schaltbarkeit einer Empfangstransition getestet, ob die vorderste Interaktion der referenzierten Eingabe-Queue über die Methode „`typeId()`“ die geforderte[13] Interaktions-ID liefert. Ist dies der Fall und wird die Transition schließlich ausgeführt, so kann die Referenz auf

12. Jede Interaktion innerhalb einer Kanaldefinition erhält eine (mit „1“ beginnend) fortlaufend nummerierte, eindeutige ID („`IA_..._ID`“), die als Template-Parameter für das Klassen-Template „`InterAction<...>`“ benutzt wird. Da die Estelle-Syntax und -Semantik garantiert, dass nur Interaktionspunkte aus der gleichen Kanaldefinition verbunden werden können, genügt diese ID (abgefragt über die Methode „`typeId()`“) zur eindeutigen Typ-Bestimmung einer abstrakten Interaktion beim Empfänger.

13. Empfangstransitionen können jeweils nur einen spezifischen Interaktionstyp empfangen.

die empfangene Interaktion auf ihren nunmehr bekannten Ausgangstyp zurück konvertiert werden. Dadurch werden insbesondere auch die in der Komponente „param“ abgelegten Interaktionsparameter zugänglich.

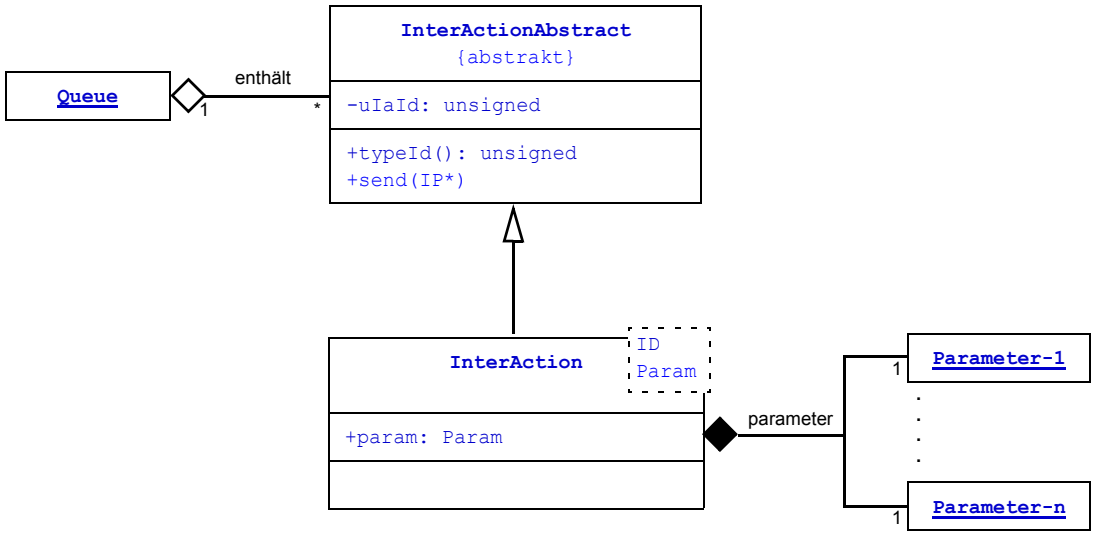


Abbildung 6-8: UML-Diagramm der Struktur der XECRT-Interaktions-Klassen

Zu diesem Zweck werden Zugriffe auf die Parameter der empfangenen Interaktion direkt auf die Komponenten der in der Interaktion enthaltenen „param“-Struktur abgebildet, indem über die Methode „param()“ des Empfangsinteraktions-Objekts eine Referenz auf diese Struktur erlangt werden kann. So kann der direkte (lesende wie auch schreibende) Zugriff einer Empfangstransition (siehe Beispiel 6.53-a) auf einen übergebenen komplexen Parameter direkt und ohne weitere Kopieroperationen in ein äquivalentes C++-Konstrukt übertragen werden (siehe Beispiel 6.53-b).

Beispiel 6.53-a: Estelle-Empfangstransition mit direktem (auch änderndem) Parameterzugriff

```
TRANS
     WHEN ipin.msg(Arg: integer; data: userdata)
     NAME receive:
          BEGIN
          IF data[13] = 199 THEN
               data[13] := 200;
          END;
```

(Ende von Beispiel 6.53-a)

Beispiel 6.53-b: Fragment des aus Beispiel 6.53-a generierten C++-Codes

```
void /* ...:: */ TRANS_receive::exec() {
     if ( (param().PAR_data)[13] == 199 ) {
          (param().PAR_data)[13] = 200;
     }
}
```

(Ende von Beispiel 6.53-b)

Dies gilt natürlich unter allen Bedingungen, also z. B. auch, wenn die empfangenen Parameter direkt wiederum als Interaktionsparameter für eine neue Interaktion benutzt werden (siehe Transition „`forwarder`“ in Beispiel 6.54-a und Beispiel 6.54-b).

Beispiel 6.54-a: Estelle-Empfangstransition mit Weiterversand der Parameterdaten

```
TRANS
     WHEN ipin.msg(Arg: integer; data: userdata)
     NAME forwarder:
          BEGIN
          OUTPUT ipout.msg(Arg, data);
          END;
```

(Ende von Beispiel 6.54-a)

Beispiel 6.54-b: Fragment des aus Beispiel 6.54-a generierten C++-Codes

```
void /* ...:: */ TRANS_forwarder::exec() {
     parent()->IP_ipout.IA_msg( param().PAR_arg, param().PAR_data );
}
```

(Ende von Beispiel 6.54-b)

Nach der Ausführung der Annahmetransition wird das (nun nicht mehr benötigte) Interaktionsobjekt zusammen mit seinen (möglicherweise durch die Transitionsausführung modifizierten) Parameterdaten durch Anwendung des C++ „`delete`“-Operators wieder freigegeben. Insgesamt wurden die komplexen Interaktionsparameter beim Versand und dem späteren Empfang der Interaktion nur einmal kopiert.

6.2.3 Explizite Kopieroperationen

Abschließend betrachten wir nun noch, welche *zusätzlichen* physischen Kopieroperationen bei der Ausführung von Transitionen durch die bereits in Abschnitt 6.1.2 angesprochenen, explizit spezifizierten semantischen Kopieroperationen in XEC bedingt werden.

Entsprechend des Ansatzes von XEC, Estelle-Operationen zunächst so weit wie möglich durch analoge C++-Operationen zu implementieren (siehe Abschnitt 3.2.2), werden explizit spezifizierte Kopieroperationen – wie auch bei allen anderen bekannten Implementierungsgeneratoren – durchgängig in entsprechende physische Kopieroperationen umgesetzt. Dies ermöglicht unabhängig von den Eigenschaften der Spezifikation eine semantikkonforme Implementierung solcher Operationen und ist damit eine universelle Implementierungsmethode.

So wird der Block der in Beispiel 6.49-a auf Seite 251 dargestellten Empfangstransition, in der das Framing einer SDU (Typ `sdu_t`) in eine PDU (Typ `pdu_t`) mit aggregierter SDU-Typ-Komponente (`payload`) spezifiziert ist, von XEC in das folgende C++-Fragment übersetzt, in dem durch die Zuweisung auf die Komponente `VAR_pdu.COMP_payload` eine vollständige Kopie des Interaktionsparameters erzeugt wird.

Beispiel 6.55: Fragment des aus Beispiel 6.49-a auf Seite 251 erzeugten C++-Codes

```
void /* ...:: */ TRANS_frame::exec() {
     TYPE_pdu_t VAR_pdu;
     VAR_pdu.COMP_header  = /* ... */;
     VAR_pdu.COMP_payload = param().PAR_data;
     VAR_pdu.COMP_trailer = /* ... */;
     parent()->IP_ipdown.IA_msg(VAR_pdu);
}
```

(Ende von Beispiel 6.55)

Zusammenfassend lässt sich also festhalten, dass die von XEC eingesetzte Referenz-Implementierung des Interaktionsparameter-Transports den Versand und späteren Empfang einer Interaktion mit insgesamt nur einer Kopieroperation der übergebenen Parameter realisiert. Weitere, explizit spezifizierte Kopieroperationen, die z. B. beim Zwischenspeichern von Paketen zum Zwecke des wiederholten Sendens oder beim Framing zur Einbettung von SDUs in PDUs erforderlich sind, führen ebenfalls zu einer physischen Kopie der enthaltenen Nutzdaten.

6.2.4 Datenübertragungsbenchmark

Zur quantitativen Bewertung der Effizienz verschiedener Implementierungsmethoden für semantische Kopieroperationen wurde der bereits in Abschnitt 2.3.2 eingeführte Datenübertragungsbenchmark entwickelt, der anhand eines vereinfachten Kommunikationsszenarios die bereits in Abschnitt 6.1 dargestellten, für Kommunikationsprotokolle typischen Datenübertragungs- und Datenbearbeitungsschritte in einer geschlossenen (Standard-) Estelle-Spezifikation anwendet.

Die Spezifikation (siehe Anhang B.1) beschreibt ein Ende-zu-Ende-Kommunikationsszenario, bei dem zwei Dienstnutzer („Usr“ Module) jeweils über einen dreistufigen Protokollstack (Module „PM_1“ bis „PM_3“) und ein vermittelndes Netzwerk (Module „Net“ und „Router“) mit einer parametrierbaren Anzahl von „Hops“ Nutzdaten (als Parameter von Sendeaufträgen) austauschen können (siehe Abb. 6-9).

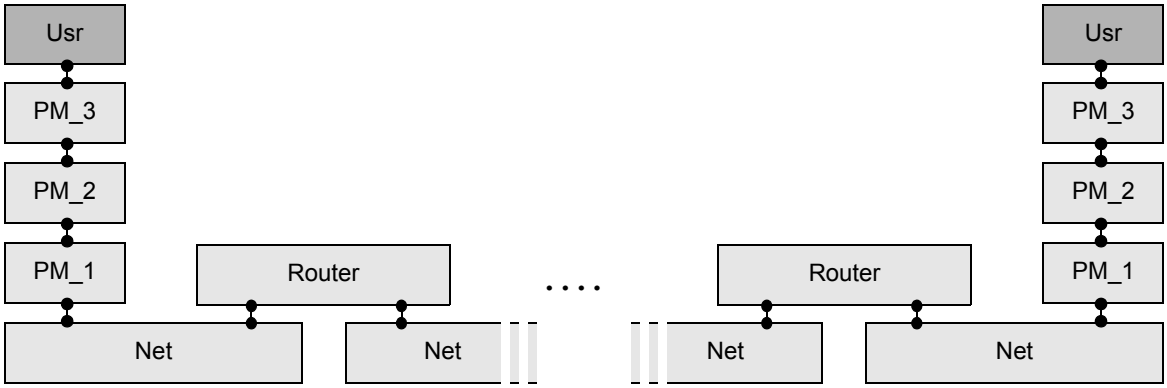


Abbildung 6-9: Modulstruktur der Datenübertragungsbenchmarkspezifikation

Die Protokollmaschinen „PM_1“ bis „PM_3“ simulieren typische Framing- und Unframing-Situationen, wie wir sie bereits in Abschnitt 6.1.2 beschrieben haben. Dazu werden beim Versenden einer Nachricht zum Netzwerk durch Feuern einer Transition („Frame“) die von den jewei-

ligen Dienstnutzern übergebenen SDUs unverändert in PDUs eingebettet, welche neben dieser Nutzlast noch Paketheader und -Trailer enthalten. Diese protokollspezifischen Zusatzinformationen wurden in der Benchmarkspezifikation als Integer-Typen angelegt und jeweils mit einem Dummy-Wert initialisiert. Umgekehrt erfolgt beim Empfang einer Nachricht vom Netzwerk durch Feuern einer Transition („`Unframe`“) das analoge Unframing und die Weitergabe der SDU an den jeweiligen Dienstnutzer. Die Inhalte der Paketheader und -Trailer bleiben unberücksichtigt.

Die Protokollmaschine „`PM_2`“ simuliert über dieses einfache Framing und Unframing hinaus zusätzlich die Pufferung gesendeter Pakete zum Zwecke einer potenziellen Paketwiederholung, wie sie z. B. in Transportschichtprotokollen erforderlich ist (siehe auch Abschnitt 6.1.2). Dazu wird in der Transition „`Frame`“ nach dem Versenden einer PDU zum Netzwerk eine Kopie der PDU in einen modullokalen Ringpuffer („`Resend_Buffer`“) abgelegt, der eine parametrierbare Anzahl von PDUs speichern kann. Aus diesem Puffer heraus wären Paketwiederholungen möglich, es wird von dieser Möglichkeit in der hier vorgestellten Fassung der Benchmarkspezifikation jedoch kein Gebrauch gemacht. Auf Empfangseite ist daher hier auch keine besondere Behandlung dieses Mechanismus erforderlich.

Die Protokollmaschine „`PM_3`“ setzt auf eine Instanz eines Netzwerk-Dienstes auf. Diese „`Net`“-Module haben jeweils zwei obere Interaktionspunkte, über die sie einen symmetrischen Datenübertragungsdienst bereitstellen. Dieser besteht darin, eingehende Nachrichten von einem Interaktionspunkt durch Feuern einer Transition unverändert an den anderen Interaktionspunkt weiter zu reichen.

Um die Vermittlung von Paketen über mehrerer Netzwerke (d. h. Instanzen von „`Net`“) nachbilden zu können, kann jeder dieser „Hops“ durch die Kopplung zweier Netzwerke durch eine „Router“-Instanz simuliert werden. Dazu besitzen diese Module zwei untere Interaktionspunkte, über die sie symmetrisch beim Empfang einer PDU an einem der Interaktionspunkte durch Feuern einer Transition den Nutzlastanteil der PDU extrahieren und eine neue PDU bilden, um sie dann an den jeweils anderen Interaktionspunkt zu verschicken. Diese Kombination aus Unframing und Framing soll den Einsatz verschiedener Paketformate auf verschiedenen Teilnetzen simulieren. Die Anzahl der „Hops“ (d. h. die Anzahl der Instanzen von Modul „Router“) kann parametriert werden, wobei ein Wert von 0 bedeutet, dass die beiden oben beschriebenen Protokollstacks direkt über die selbe Instanz von „`Net`“ kommunizieren.

Das gesamte Kommunikationsszenario ergibt die in Abb. 6-9 gezeigte Konstellation von Instanzen der vorgestellten Module zu einem Kommunikationspfad, über den zwei Dienstnutzer („`Usr`“-Module) über einen Protokollstack und ein vermittelndes Netzwerk Nutzdaten vom Typ „`T_UserData`“ austauschen können. In der Benchmarkspezifikation ist dieser Typ als Byte-Array parametrierbarer Größe (Konstante „`UserDataSize`“) angelegt. Die Dienstnutzer sind Instanzen des selben Moduls, die jeweils abwechselnd aktiv sind. Dabei sendet der gerade aktive Dienstnutzer den Inhalt einer lokalen Variablen[14] vom Typ „`T_UserData`“ über das darunter liegende Kommunikationssystem an den anderen Dienstnutzer und wird danach inaktiv. Sobald die Gegenstelle dieses Nutzdatum empfängt, kopiert sie es in eine lokale Variable und wird selbst aktiv, um wiederum eine Übertragung in Gegenrichtung zu initiieren. Bei der Initialisierung des Systems wird dabei genau einer der Dienstnutzer (gesteuert durch einen Initialisierungs-Parameter) aktiviert, sodass der Nachrichtenaustausch in einem „Ping-Pong“-

14. Es ist zu beachten, dass die vorgestellte Dienstschnittstelle, in der der Dienstnutzer einen beliebigen (lokalen) Datenbereich als Quelle und Ziel der Datenübertragung nutzt, auch der Dienstschnittstelle von SandiaXTP (siehe Abschnitt 6.3.5) entspricht.

Muster erfolgt. Zum Ausgleich statistischer Schwankungen erfolgt eine parametrierbare Anzahl von Datenübertragungen, die im Folgenden angegebenen Werte beziehen sich jedoch auf den (ggf. durchschnittlichen) Aufwand einer Ende-zu-Ende-Übertragung.

Alle Protokollabläufe wurden so weit wie möglich vereinfacht, um den Einfluss datenübertragungsfremder Aspekte der Implementierung so weit wie möglich zu reduzieren. So werden z. B. keine Paketverluste oder Timeouts zu deren Erkennung modelliert. Dadurch konnten alle Module mit jeweils nur zwei Transitionen (zur Realisierung beider Übertragungsrichtungen) realisiert werden. Eine wesentliche Randbedingung für die hier und weiter unten vorgestellten verschiedenen Implementierungen des Benchmarks war die Darstellung des zu sendenden Datums und des empfangenen Datums jeweils als Inhalt einer lokalen Variablen im Benutzermodul, die bezüglich Lebensdauer und möglicher Zugriffsrestriktionen für den jeweiligen Dienstnutzer unbeschränkt verfügbar sind. Wir werden später an geeigneter Stelle nochmals auf diesen Aspekt zurückkommen.

Im Verlauf einer Ende-zu-Ende-Übertragung ergeben sich bei H Hops die in Tabelle 6.25 angegebenen Operationen auf den zu übertragenden Nutzdaten (bzw. auf PDUs die diese Nutzdaten direkt oder indirekt enthalten).

Tabelle 6.25: Operationen auf Nutzdaten für Ende-zu-Ende-Übertragung

Modulname	Richtung	Anzahl Instanzen	Operationen pro Modulinstanz				
			Frame	Unframe	Copy	Send	Receive
Usr	send	1				1	
PM_3/1	send	2	1			1	1
PM_2	send	1	1		1	1	1
Net		H+1				1	1
Router		H	1	1		1	1
PM_3/1	receive	2		1		1	1
PM_2	receive	1		1		1	1
Usr	receive	1			1		1

Somit sind für eine komplette Ende-zu-Ende-Übertragung eines Nutzdatums jeweils $H + 3$ Framing- bzw. Unframing-Operationen, 2 Kopieroperationen und jeweils $2 \cdot H + 7$ Sende- bzw. Empfangsoperationen als Interaktionsparameter erforderlich. Da bis auf die Empfangsoperationen alle diese Operationen mit genau einer semantischen Kopieroperation verbunden sind, ergeben sich somit insgesamt $3 \cdot H + 12$ semantische Kopieroperationen, also gemäß der oben beschriebenen Referenz-Implementierungsmethode von XEC die gleiche Anzahl von physischen Kopieroperationen.

Bei den hier und im Folgenden durchgeführten Experimenten wurden folgende Parameter für den Datenübertragungsbenchmark eingesetzt:

- Anzahl Hops (H): 5
- Anzahl der Ping-Pong-Durchläufe: 10000
- Anzahl der PDUs im „`Resend_Buffer`“: 3
- Größe von „`T_UserData`“: variabel

Die Ende-zu-Ende-Übertragung mit 5 Hops beinhaltet insgesamt $(3 \cdot 5 + 12) = 27$ (semantische) Kopieroperationen aus verschiedenen Protokollmechanismen.

Die Auswertung des Benchmarks für die Referenz-Implementierungsmethode von XEC unter Variation der Größe des zu übertragenden Nutzdatentyps erfordert in der Testumgebung[15] die in Tabelle 6.26 angegebenen Zeiten für eine Ende-zu-Ende-Übertragung.

Tabelle 6.26: Ergebnisse des Datenübertragungsbenchmarks für die Referenz-Implementierungsmethode

Größe Nutzdatentyp [Byte]	**Übertragungsdauer**		
	absolut [ms] [a]	**absolut normiert**	**relativ** [µs/Byte]
10	0,0595 ±0,001	= 1,0	5,95
100	0,0700 ±0,001	1,2	0,70
1´000	0,1764 ±0,004	3,0	0,176
10´000	1,543 ±0,009	26	0,154
100´000	15,38 ±0,008	258	0,154
1´000´000	153,2 ±0,007	2575	0,153

a. Die angegeben Genauigkeiten beziehen sich auf die Zeitmessung und die statistische Streuung der gemessenen Ergebnisse (siehe Abschnitt 2.3).

Offensichtlich überwiegt für kleine Nutzdatentypen (10 und 100 Byte) der Zeitanteil für das Protokollmanagement den Datenübertragungsaufwand so weit, dass kein nennenswerter Einfluss auf die (absolute) Ausführungszeit besteht, die absolute Ausführungsdauer also fast konstant bleibt. Ab etwa 1´000 Byte kehrt sich dies um, so dass sich näherungsweise ein linearer Zusammenhang zwischen Größe des Nutzdatentyps und absoluter Übertragungsdauer ergibt, die relative Übertragungsdauer (d.h. der Quotient der beiden Werte) nahezu unverändert bei ca. 0,16 µs/Byte bleibt.

Wie unbedeutend bei größeren Paketen der Zeitaufwand für das Protokollmanagement ist, erkennt man am Vergleich der absoluten Übertragungsdauer großer Pakete mit der von kleinen Paketen (Spalte „absolut normiert“, bezogen auf Übertragungsdauer für 10 Byte Nutzdatentyp). Liegen bei 1´000 Byte die Werte noch in ähnlichen Größenordnungen (1:3), so ist der Protokollaufwand bei größeren Paketen zunehmend vernachlässigbar (bei 10´000 Byte bereits 1:26 entsprechend < 4%).

Ausgehend von diesen Referenzwerten untersuchen wir nun alternative Implementierungsmethoden, die eine effizientere Handhabung der Datenübertragungsmechanismen erlauben. Dabei werden wir die im nächsten Abschnitt vorgestellten Handimplementierungs-Techniken zur Optimierung von Datenübertragungsszenarien später in Abschnitt 6.4 auf ihren möglichen Einsatz bei der automatischen Implementierung von Protokollspezifikationen überprüfen.

15. Plattform: Sun Sparc Ultra-5, 1x 440 MHz CPU, SunOS 5.8, gcc 2.95.2, xec 1.3.0 („-m optimize“), Betrieb der generierten Implementierung ohne Auswahloptimerung

6.3. Optimierungstechniken bei Handimplementierung

Im Unterschied zu formalen Spezifikationstechniken bieten Handimplementierungen in imperativen Programmiersprachen wie C oder C++ eine wesentlich feinere Adaption an die technischen Details der jeweiligen Zielplattform und erlegen dem Programmierer sehr wenige bzw. gar keine Restriktionen bei der Modifikation des Zustandsraumes in der Implementierung auf.

Diese hohe Ausdrucksfähigkeit und Flexibilität wird erkauft mit einer bestenfalls plattformabhängigen Semantik und dem allgemeinen Mangel an konzeptionell erzwungenen Abstraktionen des durch die Implementierung beschriebenen Systems. So ist z. B. die erzwungene Trennung der Zustandsräume benachbarter Estelle-Module eine wichtige Grundlage für die Unabhängigkeit der damit spezifizierten Teilautomaten und damit auch Grundlage der Semantik des Gesamtsystems. In einer Handimplementierung ist der Programmierer dagegen nicht an solche erzwungenen Restriktionen gebunden und kann fast in beliebiger Art und Weise den Zustandsraum des Gesamtsystems zu jedem Zeitpunkt modifizieren. An Stelle des durch die verfügbaren Ausdrucksmittel bedingten Zwangs zur Abstraktion, wie wir ihn in FDTs finden, treten hier vom Implementierer selbst auferlegte Konventionen und Restriktionen.

Dem Nachteil der Handimplementierung, dass ihr selbst mit solchen strikten, vom Programmierer einzuhaltenden Restriktionen nur schwer eine (im mathematischen Sinne) präzise und abstrakte Semantik zuzuordnen ist, steht jedoch der Vorteil gegenüber, dass diese Beschränkungen (im Gegensatz zu den fest vorgegebenen Restriktionen der formalen Beschreibungstechniken) dem konkreten Problem flexibel angepasst werden können.

Bei der Implementierung der Datenübertragung von Kommunikationsprotokollen tritt der beschriebene Unterschied zwischen Handimplementierungen und aus formalen Spezifikationen automatisch erzeugten Implementierungen besonders deutlich zu Tage. Wie wir in den vorangegangenen Abschnitten gesehen haben, erfolgt in formalen Protokollspezifikationen z. B. der Austausch von Daten zwischen Protokollautomaten typischerweise als Interaktionsparameter, wobei semantisch beim Versenden der Interaktion eine Kopie der zu übertragenden Interaktionsparameter erstellt wird. Dadurch wird gewährleistet, dass in einer semantikkonformen Implementierung durch eine Manipulation des als Interaktionsparameter übergebenen Datums beim Empfänger keine Rückwirkung auf den beim Versenden übergebenen aktuellen Interaktionsparameter erfolgt, der Sender also vollständig von der weiteren Verarbeitung der Interaktion und ihrer Parameter isoliert bleibt und umgekehrt der Empfänger in gleicher Weise von nachträglichen Datenmanipulationen durch den Sender isoliert ist. Entsprechend müssen Sender und Empfänger (auf der Ebene dieses Datentransportmechanismus[16]) auch keine Annahmen über mögliche Kommunikationspartner machen oder Anforderungen an diese stellen.

In Handimplementierungen wird ein solcher Kopiervorgang des übertragenen Datums nicht durch einen (Spezifikations-) Mechanismus erzwungen, sondern nach Maßgabe des Programmierers implizit realisiert. Entsprechend können hier ggf. auch andere Lösungen eingesetzt werden, die keine Kopie des übertragenen Datums erfordern, sondern stattdessen z. B. lediglich eine *Referenz* auf das zu übertragende Datum mit der Nachricht weiter schicken, sodass Sender und Empfänger gemeinsam Zugriff auf das selbe Datum haben, ohne dass ein physischer Kopiervorgang erforderlich ist. Natürlich ist in einem solchen Fall die Einhaltung von spezifischen Zugriffsregeln notwendig, damit z. B. eine Manipulation der gemeinsam genutzten Daten durch

16. Im Gegensatz dazu wird auf der Ebene des spezifizierten Protokolls häufig eine spezifische Reaktion auf ein Paket erwartet, z. B. die Quittierung des Empfangs durch eine entsprechende Interaktion in Gegenrichtung.

den Sender (z. B. die Nutzung des Sendedatenpuffers zur Komposition des nächsten zu übertragenden Datenpakets) nicht zu unerwünschten Rückwirkungen auf die Sicht des Empfängers auf diese Daten führt. Die wesentliche Eigenschaft solcher Zugriffsregeln ist, dass sie von allen Komponenten eingehalten werden müssen, die Zugriff auf die betroffenen Daten erlangen können. Offensichtlich erfordert dies gerade in einem Kommunikationssystem u. U. die Vereinbarung und Einhaltung solcher Zugriffsregeln über alle beteiligten Komponenten des Systems hinweg. Es wird sich zeigen, dass dies ein wesentlicher Hemmschuh beim Einsatz solcher Methoden in nicht monolithisch implementierten Systemen ist.

Grundsätzlich müssen die Methoden zur Vermeidung von physischen Kopieroperationen in manuell erstellten Implementierungen von Kommunikationsprotokollen die folgenden, bereits früher (siehe Abschnitt 6.1) genannten protokolltypischen Operationen effizient handhaben:

- Übertragen von Daten an andere Protokollautomaten,
- Framing und Unframing von Paketen,
- Pufferung von gesendeten Paketen für potentielle Übertragungswiederholungen und
- Fragmentierung und Defragmentierung von Paketen

In den folgenden Abschnitten werden wir einige Implementierungstechniken für Handimplementierungen dieser Operationen betrachten.

6.3.1 Initial Placement

Mit „Initial Placement“ bezeichnet man allgemein die Technik, die zu übertragenden Daten so abzulegen, dass sie möglichst ohne weitere Kopieroperationen über mehrere Protokollschichten hinweg weiterverarbeitet werden können (siehe Abb. 6-10). Dadurch wird der Datentransport zwischen den Dienstschichten ohne Kopieroperationen der Daten realisiert, es wird statt dessen nur ein Deskriptor der Datenablage[17] (z. B. ein Pointer auf diese) zwischen den Protokollkomponenten ausgetauscht. Wenn die Datenablage den beteiligten Komponenten *implizit* bekannt ist, ist nicht einmal die Übermittlung eines expliziten Deskriptors erforderlich. Dies gilt z. B. dann, wenn sie immer über den selben Puffer erfolgt.

Initial Placement setzt natürlich voraus, dass die Ablage der Daten in einem Bereich erfolgt, der in allen zugreifenden Kontexten direkt zugänglich ist. Somit ist diese Technik zunächst auf Protokollkomponenten innerhalb des gleichen Prozesses bzw. einer Gruppe von Prozessen mit Zugriff auf einen gemeinsamem Speicherbereich („shared memory“) beschränkt. Gerade im letzteren Fall erfordert der konkurrierende Zugriff auf das selbe Speicherobjekt durch verschiedene Prozesse eine wirksame Zugriffssteuerung zur Vermeidung von unerwünschten Wechselwirkungen. Diese Zugriffsteuerung kann durch die Koordination der beteiligten Prozesse oder auch durch Hardwareunterstützung (siehe Abschnitt 6.3.6) realisiert werden.

17. im Folgenden auch als *Referenz auf die Daten* bezeichnet

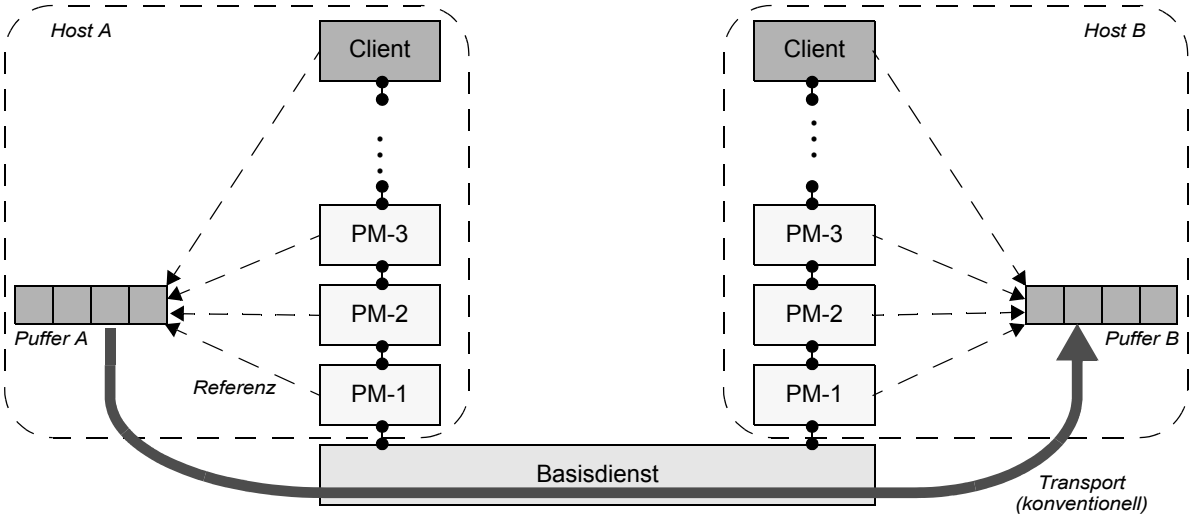


Abbildung 6-10: Initial-Placement in gemeinsamem Puffer eines Hosts (jeweils A/B)

6.3.2 Application Layer Framing

Das „*Application Layer Framing*“ (ALF, [ClTe90]) ist eine Initial-Placement-Technik, die bereits auf Applikationsebene Nutzdaten derart in einem Puffer (z. B. „Puffer A“ in Abb. 6-11) ablegt, dass später die darunter liegenden Protokollmaschinen beim Framing der (von der jeweils darüber liegenden Protokollmaschine in den Puffer abgelegten) SDU durch direktes Vorneanfügen eines Paketheaders und Hintenanfügen eines Pakettrailers *in den selben Puffer* erfolgt. Dadurch entstehen schließlich an der Schnittstelle zum untersten Netzwerkdienst in diesem Puffer die ineinander geschachtelten PDUs als zusammenhängende Bytesequenz. Beim Empfang wird in umgekehrter Reihenfolge die in einem solchen Puffer (als ebenfalls zusammenhängende Bytesequenz empfangene) PDU vom jeweiligen Protokollautomaten in Header, Trailer und SDU strukturiert, um letztere dann als *Referenz auf diesen Teil des Puffers* an den jeweiligen Dienstnutzer weiterzugeben.

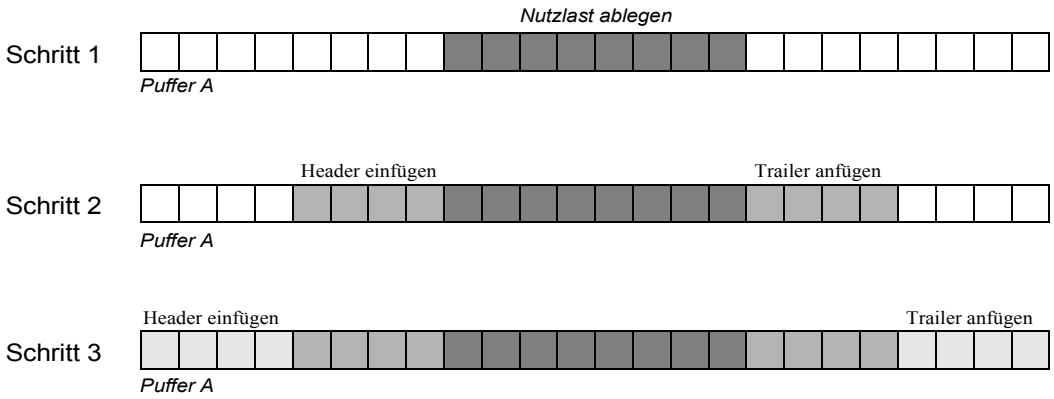


Abbildung 6-11: Zeitlicher Ablauf des Application Layer Framing

Diese Implementierungsmethode hat zwei wesentliche Vorteile:

- Jedes Datum (Nutzdaten wie auch Paketheader und -Trailer) wird beim Senden (Framing) wie auch beim Empfang (Unframing) jeweils nur *einmal* dem Puffer zugewiesen und niemals kopiert oder verschoben. Dadurch wird nicht nur der Datentransport zwischen den Dienstschichten, sondern auch das Framing ohne Kopieroperationen realisiert. Wenn die Nutzdaten vom Dienstnutzer direkt in dem Puffer erzeugt werden können (also nicht erst in anderen Speicherbereichen erzeugt und dann in den Puffer kopiert werden), so ergibt sich für die genannten Aspekte eine bzgl. Datenkopieroperationen optimale Implementierung.
- Das Ergebnis des Framingvorgangs ist bereits eine zusammenhängende Bytesequenz, die über einfache Standard-Betriebssystemschnittstellen zum Versenden von zusammenhängenden Daten mit nur einem Funktionsaufruf[18] als Sendeauftrag an einen Netzwerk-Basisdienst übergeben werden kann.

Somit verspricht dieses Verfahren für Framing, Datentransport und die Weitergabe an plattformspezifische Basisdienste eine hohe Effizienz. Diesen Vorteilen stehen aber auch Nachteile gegenüber:

- Da die initiale Positionierung der Nutzlast im Puffer durch den Dienstnutzer auch Rückwirkungen auf die maximale Größe der in den darunter liegenden Protokollschichten noch in den Puffer integrierbaren Header und Trailer hat, müssen bereits beim Dienstnutzer die Anforderungen der darunter liegenden Protokollschichten in dieser Hinsicht berücksichtigt werden. Dies wird deutlich schwieriger, wenn heterogene Basisdienste eingesetzt werden, die u. U. abhängig von den Übertragungsanforderungen der Verbindung andere Protokolle oder sogar für jedes Paket z. B. einen anderen Leitweg nutzen und dabei auch unterschiedliche Dienste mit unterschiedlichen Headern und Trailern zur Anwendung kommen (siehe auch Abschnitt 5.1.4). Umgekehrt bewirkt eine vorsorglich großzügige Auslegung dieser Bereiche eine Verschwendung der möglicherweise ungenutzt bleibenden Pufferbereiche.
- Die Pufferung von Paketen für eine potentielle Wiederholung ist nur durch Kopieren oder das Zulassen von Mehrfachreferenzen auf den Puffer möglich. Im letzteren Fall bleibt der gesamte Puffer (der ja prinzipbedingt potentiell größer als das eigentliche Paket ist) für den Zeitraum der Paketaufbewahrung belegt (siehe auch Abschnitt 6.3.6).
- Fragmentierung und Reassemblierung ist bei ALF nur durch Umkopieren möglich.[19]

Die im folgenden Abschnitt vorgestellte Technik versucht, diese Nachteile durch eine Abstraktion der dem Application Layer Framing zu Grunde liegenden Idee zu vermeiden.

6.3.3 Scatter-Gather-Vektoren

Der Einsatz von „*Scatter-Gather-Vektoren*“ ([DrPeDa94], [PaDrZw00]) ist ebenfalls eine Initial-Placement-Technik, die im Vergleich zum ALF jedoch flexibler in der Anwendung ist. Zu Grunde liegt die Idee, als Ergebnis des Framings nicht das Paket als eine zusammenhängende Bytesequenz zu erzeugen, sondern eine Menge von Paketfragmenten beschrieben durch eine Beschreibungsstrukur – hier ein *Vektor* – einzusetzen. Die einzelnen Felder des Vektors be-

18. z. B. „`send(int socket, const void* data, size_t length, int flags)`“ zum Versenden von Daten über einen Socket unter UNIX

19. speziell wenn die einzelnen Fragmente mit Headern oder Trailern versehen werden

schreiben jeweils Ort und Größe eines jeden Fragments. Das vom Vektor beschriebene Paket besteht aus der Verkettung dieser Fragmente in der angegebenen Reihenfolge. Im einfachsten Fall besteht ein solcher Vektor aus nur einem Eintrag mit Ort und Größe einer Bytesequenz.

Zum Framing eines durch einen Scatter-Gather-Vektor beschriebenen Paketes genügt es, die Header und Trailer als (ein- oder mehrelementige) Bytesequenzen in einem global zugreifbaren Speicherbereich abzulegen und diese durch jeweils einen (ein- oder mehrelementigen) Scatter-Gather-Vektor zu beschreiben, um dann die drei Vektoren in geeigneter Reihenfolge zu einem neuen Vektor zu konkatenieren (siehe Abb. 6-12). Damit ist das Framing allein durch Kopie der Deskriptorstruktur möglich, ohne die eigentlichen Daten bewegen oder manipulieren zu müssen. Das Unframing erfolgt analog zum ALF ebenfalls ohne Kopieroperationen auf den Daten, wobei hier u. U. jedoch zur Erzeugung eines Scatter-Gather-Vektors für die extrahierte und an die Dienstnutzer weiterzureichende SDU eine Spaltung[20] von Scatter-Gather-Vektor-Fragmenten erforderlich ist.[21]

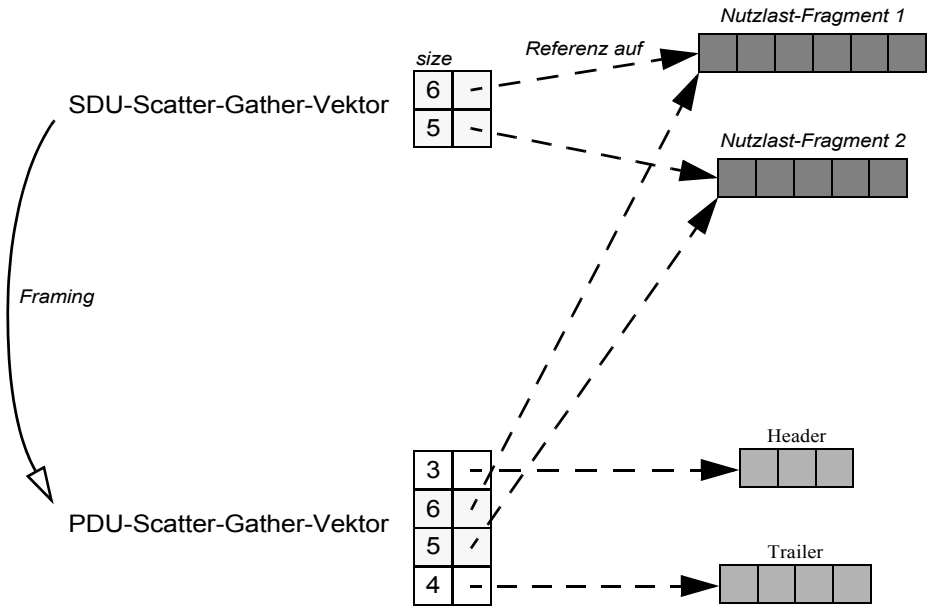


Abbildung 6-12: Ablauf des Framings mit Scatter-Gather-Vektoren

Diese Implementierungsmethode hat folgende Vorteile:

- Wie beim ALF wird hier jedes Datum (Nutzdaten wie auch Paketheader und -Trailer) beim Senden (Framing) wie auch beim Empfang (Unframing) jeweils nur *einmal* in einem (global zugreifbaren) Speicherbereich abgelegt und niemals kopiert oder verschoben. Dadurch wird nicht nur der Datentransport zwischen den Dienstschichten, sondern auch das Framing ohne Kopieroperationen realisiert.

20. Aus einem Fragment (mit einem Eintrag im Vektor) werden zwei Fragmente (mit zwei Einträgen im Vektor), wobei u. U. keine Datenkopieroperationen notwendig sind (siehe unten).

21. Eine solche Situation ergibt sich immer dann, wenn an der Grenze der zu extrahierenden SDU nicht bereits eine Fragmentgrenze liegt. Dies geschieht z. B. immer dann, wenn der Basisdienst das empfangene (mehrgliedrige) Paket als zusammenhängende Bytesequenz ablegt und damit ein Scatter-Gather-Vektor mit nur einem Element entsteht.

- Das Ergebnis des Framingvorgangs ist zwar keine zusammenhängende Bytesequenz, aber einige Betriebssystemplattformen[22] unterstützen die direkte Übergabe von Scatter-Gather-Vektoren an Systemfunktionen. So ist u. U. auch hier das Versenden des Pakets mit nur einem Funktionsaufruf als Sendeauftrag an einen Netzwerk-Basisdienst möglich.[23]
- Wenn der Dienstnutzer die Nutzdaten bereits als zusammenhängende Bytesequenz in einem global zugreifbaren Speicherbereich erzeugt, kann diese beim Framing direkt von allen Scatter-Gather-Vektoren genutzt werden, ohne die Daten nochmals kopieren zu müssen. Weitere Anforderungen an die Datenablage durch den Dienstnutzer bestehen nicht, insbesondere müssen keine protokollspezifischen Platzreservierungen für das Anfügen von Headern und Trailern berücksichtigt werden. Auch muss der Dienstnutzer in diesem Fall keine Kenntnis über die Verwendung von Scatter-Gather-Vektoren in den Diensterbringern haben: Aus Sicht des Dienstnutzers kann die Datenverwaltung der Diensterbringer weitestgehend transparent bleiben.
- Scatter-Gather-Vektoren unterstützen auch das Fragmentieren und Reassemblieren von Paketen ohne Kopieren von Paketinhalten, sofern die dabei meist notwendige Spaltung von Fragmenten des Scatter-Gather-Vektors ohne Kopieroperationen möglich ist (siehe unten).

Scatter-Gather-Vektoren stellen somit eine nahezu optimale Lösung für die o. g. Anforderungen für eine effiziente Datenverwaltung in Kommunikationsprotokollen dar. Da bei einigen Operationen jedoch Mehrfachreferenzen *auf* Fragmente (siehe Abb. 6-12) und sogar *in* Fragmenten (siehe Abb. 6-13) entstehen können, ist ggf. eine Erweiterung dieses Konzepts notwendig, um diese verwalten zu können. Lässt man keine Mehrfachreferenzen zu, so werden bei einigen Operationen (z. B. Pufferung von Paketen für eine potentielle Wiederholung) Kopieroperationen auf den Nutzdaten unumgänglich.

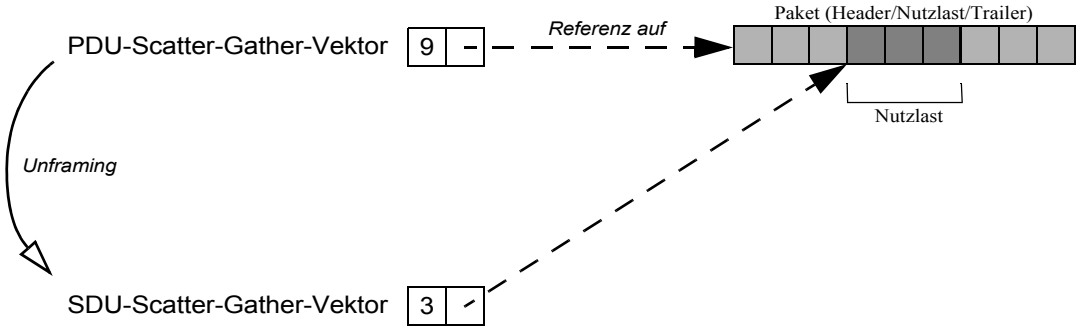


Abbildung 6-13: Mehrfachreferenzen in Scatter-Gather-Fragmente durch Unframing

22. z. B. viele UNIX-Derivate über die „`sendmsg(...)`“-Funktion

23. Auch intern werden in vielen Betriebssystemen Scatter-Gather-Vektoren für solche Ein-Ausgabe-Operationen eingesetzt, da im virtuellen Adressraum zusammenhängende Datenbereiche in der physischen Speicherstruktur (über die „intelligente“ E/A-Geräte meist adressieren) auf nicht notwendigerweise aufeinanderfolgende Speicherfragmente (Kacheln) verteilt sind (siehe auch [DrPeDa94], [DeKo92]).

Ein Ansatz zur Verwaltung der Freigabe[24] von Mehrfachreferenzen ist die Erweiterung der Vektorelemente um eine Referenz auf einen Deskriptor für den Puffer, der von dem jeweiligen Fragment (ganz oder teilweise) belegt wird. Enthält ein solcher Deskriptor einen Referenz-Zähler, so kann bei der Freigabe der letzten Referenz auch der Puffer als ganzes freigegeben bzw. erneut genutzt werden.

6.3.4 Schnittstelle zu fremden Dienstnutzern und Diensterbringern

Ohne spezifische Hardwareunterstützung (siehe Abschnitt 6.3.6) ermöglicht erst die globale Einhaltung von (zum Teil komplexen) Regeln für die Handhabung der als Referenzen übergebenen Daten die Anwendung der o. g. Mechanismen zur Vermeidung von Kopieroperationen auf den zwischen Protokollkomponenten ausgetauschten Daten. Die Einhaltung dieser Regeln ist nur bei entsprechend angepassten Kommunikationspartnern garantiert, die meist Teil einer monolithisch entwickelten Protokollimplementierung sind.

An den Grenzen eines solchen Systems ist dagegen eine wesentlich einfachere Gestaltung der Kooperationsregeln gefordert, wenn nicht sogar ein konkretes Kommunikationsschema bereits vorgegeben ist. Ein Beispiel für Letzteres sind die Schnittstellen zu Basisdiensten, die oft über Betriebssystemfunktionen zugegriffen werden. Hier wird häufig die Datenübergabe durch eine *Kopieroperation* realisiert, um eine semantisch einfache und einheitliche Schnittstelle zu solchen Diensten zu erreichen. Ein Beispiel dafür ist die unter UNIX für solche Zugriffe vorgesehene `send`-Funktion.[25]

Aber auch die Schnittstelle zu fremden Dienstnutzern eines monolithisch implementierten Protokollstacks verdeckt häufig die Details der internen Datentransportmechanismen der Protokollimplementierungen, indem ein vereinfachtes Modell für die Datenübergabe genutzt wird, das letztlich oft auf einer Kopieroperation der Nutzdaten basiert. Dadurch wird die Notwendigkeit einer Kooperation mit dem Dienstnutzer bezüglich des Verbots einer nachträglichen Manipulation des ansonsten an den Diensterbringer per Referenz übergebenen Datums vermieden.

So könnte bei ALF z. B. ein Dienstnutzer nach der Weitergabe eines Sendeauftrags (mit Puffer-Referenz) an den Diensterbringer nachträglich Manipulationen am Pufferinhalt vornehmen, die nicht nur die Nutzlast, sondern auch die von den Dienstnutzern angefügten Header und Trailer betreffen. Dies wäre durch böswillig eingesetzte Dienstnutzer, aber auch z. B. durch irrtümliches vorzeitiges Wiederverwenden des Puffers möglich. Ein solcher Eingriff kann dann zu einer Kompromittierung (Protokoll- oder Programm-„Absturz“) anderer Protokollkomponenten führen.

Ein weiterer Vorteil der Anfertigung einer Kopie der von einem Dienstnutzer übergebenen Daten ist die Möglichkeit, diese Daten dabei in einen global zugreifbaren Speicherbereich abzulegen, durch den der gemeinsame Zugriff z. B. über Prozessgrenzen erst möglich wird. Ein Beispiel dafür finden wir in der im nächsten Abschnitt dargestellten Handimplementierung Sandia-XTP.

---

24. genauer: der *expliziten* Freigabe (alternativ existiert die implizite Freigabe durch Garbage-Collection)

25. `send(int socket, const void* data, size_t length, int flags)`

6.3.5 Beispiel SandiaXTP

SandiaXTP 1.4 ([SNL95a], [SNL95b]) ist eine handkodierte und stark optimierte Implementierung von XTP 4.0, die von den Sandia National Laboratories entwickelt wurde. Die XTP-Protokollmaschine ist hier als UNIX-Dämon-Prozess realisiert, der u. a. über UDP oder IP eine Kommunikation zwischen verschiedenen Hostrechnern erlaubt. Die Anwendungen von XTP arbeiten als getrennte Prozesse, die über Sockets (für Synchronisation und Austausch von Kontrollinformationen) und gemeinsamen Speicher („*shared memory*“ für die Übertragung der Nutzdaten) mit dem XTP-Dämon am selben Host kommunizieren (siehe Abb. 6-14). Die Ankopplung an den Basisdienst erfolgt im Falle von IP und UDP vollständig über Sockets.

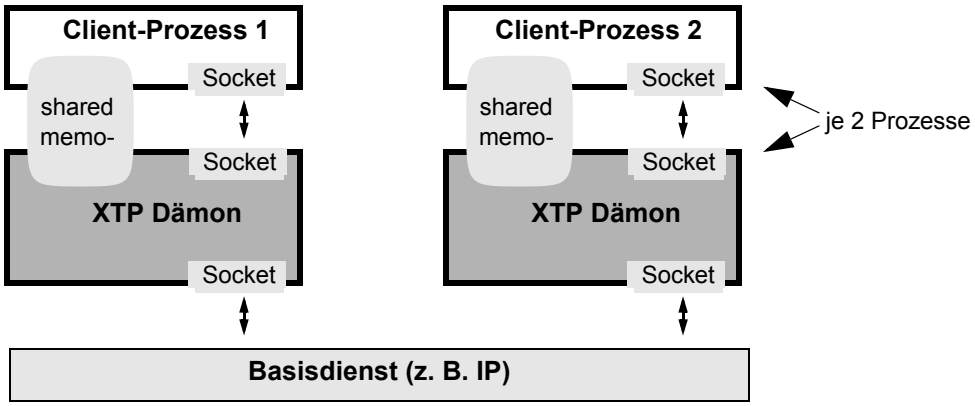


Abbildung 6-14: Die Kommunikationsstruktur von SandiaXTP

SandiaXTP hat eine durchgehend objektorientierte Struktur und ist in C++ implementiert. Die Implementierung wurde besonders bzgl. der Handhabung von Daten sehr effizienzorientiert entwickelt. So wird z. B. der Nutzdatenanteil eines Sendeauftrags nur zweimal kopiert: Beim Sendeaufruf eines Clients werden die Daten in einen Shared-Memory-Puffer kopiert, vom XTP Dämon ohne weiteres Kopieren bearbeitet und schließlich direkt aus diesem Puffer als IP-Sendeauftrag an das Betriebssystem übergeben.

Da die SandiaXTP-Protokollmaschine als separater (User-Space) Dämon-Prozess implementiert ist, sind zur Übertragung von Nutzdaten *von* einem oder *an* einen Dienstnutzer in einem separaten Unix-Prozess Interprozesskommunikationsmechanismen erforderlich. Der Dienstnutzer setzt dazu eine API-Bibliothek ein, die den Zugriff auf die Datenübertragungsdienste transparent über Schnittstellenfunktionen anbietet. Die Nutzdatenübergabe erfolgt durch Aufruf der Funktion „`xtpif::send(void* pData, int nSize, ...)`“, über die eine im Benutzerprozess lokale Bytesequenz verschickt wird. Diese Funktion kopiert die Bytesequenz in einen Shared-Memory-Puffer, über den sie für den XTP-Dämon-Prozess zugreifbar wird (siehe Abb. 6-14).

Der XTP-Dämon-Prozess belässt diese Kopie der Nutzdaten für die folgende Verarbeitung in diesem Puffer, ohne nochmals eine Kopie anzufertigen. Dazu wird das Framing mit Hilfe von Scatter-Gather-Vektoren durchgeführt, die neben den (prozesslokal abgelegten) Paketheadern und -Trailern auch diese Nutzdatenkopie im Shared-Memory-Puffer referenzieren. Im Laufe der Paketverarbeitung wird das so entstehende XTP-Paket schließlich durch Übergabe des beschreibenden Scatter-Gather-Vektors an die Betriebssystemfunktion `sendmsg(...)` in die Übertragungspuffer der betriebssysteminternen Basisdienst-Protokollmaschinen (z. B. UDP/IP) zur weiteren Übertragung übergeben. Als Transportprotokoll speichert XTP diese Nutzdaten über das erstmalige Versenden hinaus für eine potentielle Wiederholung von Paketen. Dazu erzeugt die Protokollmaschine Mehrfachreferenzen auf die (immer noch im o. g. Shared-Me-

mory-Puffer abgelegten) Nutzdaten. Die Verwaltung der Mehrfachreferenzen erfolgt durch zentrale Kontrolldatenstrukturen innerhalb der XTP-Protokollmaschine. Insbesondere gilt die Konvention, dass mehrfach referenzierte Daten nicht verändert werden, um so die notwendige Copy-Semantik der durch die Mehrfachreferenzen eingesparten semantischen Kopieroperationen einzuhalten.

Der Empfang von Nutzdaten erfolgt in Gegenrichtung durch die Annahme einer Nachricht vom Basisdienst durch entsprechende Betriebssystemfunktionen (`recvmsg(...)`) direkt in einen der o. g. Shared-Memory-Puffer. Dadurch können die enthaltenen Nutzdaten auch ohne weitere Kopieroperationen bis in den Ziel-Nutzerprozess übertragen werden, wo sie vom Dienstnutzer über die Funktion „`xtpif::receive(void* pBuffer, int nSize, ...)`“ in einen lokalen Empfangspuffer kopiert werden. Da beim Empfang keine weitere Aufbewahrung der Nutzdaten erfolgt, entstehen hier auch keine Mehrfachreferenzen wie in Senderichtung. Entsprechend ist der Shared-Memory-Puffer nach Aufruf der Funktion „`xtpif::receive`“ auch wieder sofort zur Wiederverwendung verfügbar.

Bei einem XTP-Ende-zu-Ende-Transport wird ein zu übertragendes Nutzdatum (ohne Berücksichtigung des zu Grunde liegenden Basisdienstes selbst) also viermal kopiert: zweimal an der Schnittstelle zum Basisdienst und zweimal an der Schnittstelle zum XTP-Dienstnutzer. Diese Kopieroperationen erfolgen in erster Linie zur Vereinfachung der Schnittstellen zwischen den beteiligten Komponenten, indem der jeweilige Dienstnutzer durch den Kopiervorgang die volle Kontrolle über den jeweils von ihm eingesetzten Datenpuffer behalten kann und keine weitere Koordination beim Zugriff auf diese Puffer erforderlich ist.[26] Insbesondere kann durch den Kopiervorgang an der XTP-Dienstschnittstelle die XTP-Protokollmaschine intern zum Datentransport spezielle Shared-Memory-Bereiche nutzen, ohne dass der Dienstnutzer diesen Umstand beachten oder auch nur kennen müsste.

6.3.6 Speicherverwaltung bei virtueller Adressierung

In den vorangegangenen Abschnitten wurde häufig das physische Kopieren von Nutzdaten eingesetzt, um eine vollständige Entkopplung zwischen zwei Protokollkomponenten bezüglich ihrer Operationen auf diese Nutzdaten zu realisieren. Auf Systemen mit virtueller Adressierung lassen sich auf Implementierungsebene diese Kopieroperationen unter bestimmten Umständen teilweise oder sogar ganz vermeiden.

Derartige Speicherverwaltungen bilden den Inhalt von physischen Speicherbereichen durch einen (normalerweise Hardware-implementierten) Übersetzungsmechanismus in einen oder mehrere so genannte virtuelle Adressräume ab. Bei dieser Abbildung können zusätzlich Schutzattribute wie z. B. das Verbot von ändernden Zugriffen auf Teile des virtuellen Adressraums (Schreibschutz) realisiert werden. Eine Verletzung dieser Schutzattribute durch einen Benutzerprozess wird durch die Speicherverwaltung erkannt und als Ausnahmebedingung von einer entsprechenden Betriebssystemsroutine behandelt.

---

26. Es ist zu beachten, dass die vorgestellte XTP-Dienstschnittstelle, in der ein Dienstnutzer einen beliebigen (typischerweise lokalen) Datenbereich als Quelle und Ziel der Datenübertragung nutzen kann, genau der obersten Dienstschnittstelle des in Abschnitt 6.2.4 vorgestellten Datenübertragungsbenchmarks entspricht.

Solche Speicherverwaltungen erlauben es häufig, den selben physischen Speicherbereich mehrmals im selben virtuellen Adressraum oder auch in verschiedenen virtuellen Adressräumen unterschiedlicher Prozesse einzublenden. Diese Art von Shared-Memory macht es möglich, beliebige (ursprünglich lokale) Speicherobjekte eines Prozesses ohne physische Kopieroperationen in einem anderen Prozess sichtbar zu machen, doch beinhaltet sie auch den für Shared-Memory typischen Mangel an Entkopplung beim ändernden Zugriff auf den gemeinsamen Speicher.

In Kombination mit den oben genannten Schutzattributen kann jedoch die geforderte Entkopplung der beiden Einblendungen des selben physischen Speicherobjekts im Sinne einer Copy-Semantik erreicht werden. So kann z. B. die Einhaltung eines Verbots der Modifikation des gemeinsam benutzten Speicherbereiches durch die Anwendung von Schreibschutz-Attributen auf die entsprechenden Teile der jeweiligen virtuellen Adressräume erzwungen werden. Verletzt ein Prozess dieses Verbot, so wird dies über den oben beschriebene Mechanismus vom Betriebssystem erkannt und die modifizierende Operation (notfalls durch Termination des Prozesses) wirksam unterbunden.

Häufig ist ein solches Verbot der Modifikation von Speicherbereichen nicht sinnvoll. So ist es zum Zwecke der Abstraktion von der eingesetzten Datenübertragungstechnik häufig wünschenswert, dass für die beteiligten Komponenten die Copysemantik bis hin zur Möglichkeit eines ändernden Zugriffs auf die übertragenen Datenbereiche erhalten bleibt. Zudem ist eine Modifikation solcher gemeinsam genutzter Speicherbereiche auf Grund der Granularität der Speicherverwaltung[27] häufig nicht zu vermeiden, wenn direkt vor oder hinter dem Nutzdatenbereich (z. B. beim Application Layer Framing, siehe Abschnitt 6.3.2) Datenmanipulationen erforderlich sind.

In diesen Fällen kommt die *Copy-on-Write-Technik* zum Einsatz, bei der die gemeinsam genutzten Speicherbereiche in beiden virtuellen Adressräumen zunächst mit einem Schreibschutz-Attribut versehen sind, die Verletzung dieses Attributs (also der schreibende Zugriff auf diesen Speicherbereich) aber transparent für alle beteiligten Prozesse behandelt wird, indem eine (dann nicht mehr schreibgeschützte) Kopie des betroffenen physischen Speicherbereichs erstellt und in den entsprechenden virtuellen Speicherbereich eingeblendet wird. Durch diesen Mechanismus werden physische Kopien der Nutzdaten nur dann erzeugt, wenn dies auf Grund von Schreiboperationen zur Einhaltung der Copy-Semantik unvermeidlich wird. Darüber hinaus können – abhängig von der Granularität der Speicherverwaltung – die Kopieroperationen auch auf die betroffenen Teile des Nutzdatenbereichs (z. B. die beschriebenen Speicher-Seiten) beschränkt werden. Beim o. g. Application Layer Framing wären dies schlimmstenfalls die erste und die letzte Seite des Nutzdatenbereichs, da nur diese (bei teilweiser Belegung) von einer Ergänzung eines Headers und Trailers betroffen sein können.

Offensichtlich erfordert der Einsatz solcher Mechanismen Betriebssystem- und Hardware-spezifische Anpassungen der Implementierung in erheblichem Umfang, die die Portabilität einer Implementierung selbst auf konzeptioneller Ebene beeinträchtigt. Auch gibt es Abweichungen von dem semantischen Vorbild der physischen Kopieroperation, die sich häufig aus der groben Granularität (bei Paging häufig einige kB) der Speicherverwaltungsmechanismen ergibt. So kann der Empfänger einer derartigen Datenübertragung meist den Offset der zu empfangenden Daten innerhalb einer Speicherverwaltungseinheit nicht kontrollieren. Etwas subtiler sind die (Sicherheits-) Probleme, die sich durch die Mitübertragung von Daten am Rand des zu verschi-

27. Gerade bei Speicherverwaltungen mit festen Seitengrößen wird im Allgemeinen ein größerer Bereich übertragen und damit mehrfach eingeblendet werden, als bei der Datenübertragung ursprünglich angefordert wurde.

ckenden Speicherbereichs ergeben. Aus diesen Gründen werden Datenübertragungsmechanismen auf der Basis von virtuellen Speicherverwaltungen meist nur direkt an der Betriebssystemschnittstelle bzw. innerhalb des Betriebssystemskerns eingesetzt.

Wir werden später (u. a. in Abschnitt 6.4.1) nochmals auf die praktische Anwendung dieser Technik zurückkommen.

6.4. Optimierungsansätze in automatisch generierten Implementierungen

Wie wir in den vorangegangenen Abschnitten gesehen haben, stehen bei der Handimplementierung verschiedene Techniken zu Verfügung, um die Übertragung von Nutzdaten über verschiedene Protokollschichten hinweg effizient zu gestalten. Die wesentliche Zielsetzung von Abschnitt 6.4 ist die Übertragung derartiger effizienter Datenübertragungstechniken auf automatisch generierte Implementierungen von formalen Protokollspezifikationen. Dazu betrachten wir zunächst in Abschnitt 6.4.1 die Möglichkeiten, für beliebige Spezifikationen automatisch semantikkonforme Optimierungen der Datenübertragungen zu erreichen, ohne die Spezifikationen für diese Zielsetzung anpassen zu müssen. Anschließend betrachten wir in Abschnitt 6.4.2 pragmatische Ansätze zur Optimierung der Datenübertragung, die die bei Handimplementierungen eingesetzten Techniken zur Optimierung der Datenübertragung verfügbar machen, indem Mechanismen (wie z. B. primitive Funktionen und Annotationen) eingesetzt werden, die über die Ausdrucksfähigkeit der formalen Techniken selbst hinausgehen. Schließlich untersuchen wir in den Abschnitten 6.4.3 bis 6.4.6 verschiedene Spracherweiterungen für Estelle, die es bereits auf *Spezifikationsebene* erlauben, die Datenübertragung derart problemangemessen zu beschreiben, dass eine effiziente Implementierung automatisch abgeleitet werden kann.

6.4.1 Automatische semantikkonforme Optimierungen

Wir haben in den vorangegangenen Abschnitten die in Datenübertragungsprotokollen typischerweise auftretenden Operationen auf Nutzdaten sowohl für formale Protokollbeschreibungen in Estelle, als auch in Handimplementierungen kennen gelernt. Auch haben wir gesehen, wie die Datenverarbeitungsoperation von formalen Beschreibungstechniken automatisch in eine semantikkonforme Implementierung umgesetzt werden kann (siehe Abschnitt 6.2). Die dort beschriebene universelle Implementierungsmethode kann sich jedoch, wie wir gesehen haben, bezüglich ihrer Effizienz mit den in Handimplementierungen eingesetzten Operationen zur Datenübertragung nicht messen, da sie eine weitaus größere Zahl von physischen Kopieroperationen auf den Nutzdaten erfordert.

In diesem Abschnitt beschäftigen wir uns mit der Frage, inwieweit formale Protokollspezifikationen automatisch auf die effizienten Datenübertragungstechniken von Handimplementierungen abgebildet werden können. Dabei gilt es zu unterscheiden, inwieweit die formale Spezifikation auf die Anforderungen der Implementierungsmethode zugeschnitten ist.

Die meisten der in Abschnitt 6.3 vorgestellten Handimplementierungstechniken basieren auf der Übertragung von Nutzdaten zwischen Protokollkomponenten per Referenz. Dies erfordert ohne Ausnutzung der in Abschnitt 6.3.6 beschriebenen Hardware-Speicherschutzmechanismen die globale Einhaltung von Zugriffskonventionen auf die derart zwischen verschiedenen Protokollkomponenten gemeinsam genutzten Datenbereiche. Formale Protokollspezifikationen können entsprechend nur dann unter Einsatz solcher Mechanismen semantikkonform implementiert werden, wenn auch hier die Einhaltung dieser Zugriffskonventionen sichergestellt ist. Da formale Beschreibungstechniken wie Estelle jedoch die Datenübertragung semantisch als Ko-

pieroperation realisieren, ist die Einhaltung dieser Konventionen im Nachhinein[28] weder zu erwarten, noch auf Ebene einer abstrakten, problemorientierten Protollspezifikation angemessen zu motivieren.

Ein weiteres Problem stellen die typischen Protokolloperationen *Framing* und das *Kopieren von Nutzdaten für eine spätere Paketwiederholung* dar. Die Abbildung der zur formalen Spezifikation dieser Operationen eingesetzten Estelle-Stereotypen auf die gewünschten kopierlosen Implementierungen gelingt nur unter systemweiter Planung und Koordination der beteiligten Protokollkomponenten. Nicht zuletzt deshalb ist keiner der bekannten Implementierungsgeneratoren fähig, eine solche Abbildung durchzuführen.

Eine wesentliche Vereinfachung der Implementierungsaufgabe ergibt sich durch die Ausnutzung der in Abschnitt 6.3.6 beschriebenen Hardware-Speicherschutzmechanismen, speziell durch die *Copy-on-Write-Technik*. Mit ihrer Hilfe könnte die Übertragung von Nutzdaten zwischen Protokollkomponenten in vielen Fällen ohne physische Kopieroperationen realisiert werden, indem die von einem Sender als Interaktionsparameter übergebenen Nutzdaten durch die Copy-on-Write-Technik als gemeinsames, geschütztes Datum zum Empfänger übertragen werden. Solange keine der beteiligten Parteien einen Schreibzugriff auf das Datenobjekt ausführt, ist hier auch keine physische Kopieroperation erforderlich. Der Nachteil des Verfahrens besteht darin, dass alle beteiligten Protokollkomponenten (und insbesondere der Sender) zur Vermeidung solcher nachträglicher physischer Kopieroperationen ihre Datenpuffer erst dann wiederverwenden (also überschreiben) dürfen, wenn im Gesamtsystem keine Referenzen mehr auf Kopien dieses Objekts bestehen.[29] Auch bietet die Copy-on-Write-Technik bei spezifikationsnaher Implementierung von strukturierten Typen[30] keine effiziente Implementierung der oben beschriebenen Framing-Operationen, da die beschränkte Granularität der Speicherabbildungsmechanismen bei der Einbettung eines Datums als Strukturkomponente in ein bestehendes Datenobjekt (Framing) den Einsatz von physischen Kopieroperationen erzwingt. Vor diesem Hintergrund erscheint der Einsatz solcher – zudem noch extrem plattformabhängiger – Mechanismen nur bedingt durch den zu erwartenden Nutzen gerechtfertigt. Dies mag die Ursache sein, warum auch hier keiner der bekannten Implementierungsgeneratoren solche Techniken praktisch einsetzt.

Eine wesentlich günstigere Situation ergibt sich, sobald man von der Forderung nach einer universellen Implementierungsmethode abrückt, und sich auf Protokollspezifikationen beschränkt, die unter Einhaltung geeigneter (und meist für den Implementierungsgenerator spezifischer) *Spezifikationskonventionen* erstellt wurden. So könnte z. B. die Konvention, dass die Weitergabe von Daten als Interaktionsparameter ausschließlich als letzte Operation der Transitionsausführung erfolgt, die direkte Weitergabe des entsprechenden Datenpuffers vom Sender an den Empfänger ermöglichen, da durch diese Konvention (syntaktisch überprüfbar) garantiert werden kann, dass nach dem Verschicken des Datums keine Manipulation der Datenquelle erfolgt.

Voraussetzung ist dabei, dass der eingesetzte Sendedatenpuffer auf Spezifikationsebene nur die Lebensdauer der Transitionsausführung hat (d. h. es gibt beim Sender keinen Zugriff mehr nach dem Verschicken) und er in der Implementierung von vorneherein zur Weitergabe vorgesehen war, also dort gerade nicht als transitionslokale Variable (z. B. auf dem Aufrufstack), sondern

28. also ohne von vornherein auf die Implementierungsmethode angepasste Spezifikationen

29. Dies schließt auch Kopien für mögliche Paketwiederholungen ein. Diese haben üblicherweise eine Lebensdauer, die wesentlich über der zu erwartenden eigentlichen Ende-zu-Ende-Transportdauer liegt.

30. z. B. bei der Implementierung von Estelle-Records als zusammenhängender Record (`struct`) in C

als langlebiges, den Sendern und Empfängern[31] gemeinsam zugreifbares Datenobjekt implementiert wird. Diese Methode der Datenübergabe wird häufig im Zusammenhang mit *Activity-Thread-Implementierungen* beschrieben, und tatsächlich bietet sie sich im Zusammenhang mit dieser Implementierungstechnik auch als Datenübergabetechnik an. Jedoch ererbt sie damit auch die mit einer Activity-Thread-Implementierung verbundenen konzeptionellen Nachteile (siehe Abschnitt 4.2.1.2).

Die beschriebene Methode genügt jedoch nicht, um *alle* Kopieroperationen zu vermeiden. So wird z. B. bei der lokalen Zwischenspeicherung von Nutzdaten zum Zwecke einer Paketwiederholung weiterhin eine physische Kopieroperation notwendig bleiben, solange ein Schreibzugriff auf das ansonsten entstehende mehrfach referenzierte Datenobjekt nicht sicher ausgeschlossen werden kann. Auch unterbricht die Anwendung von Framing-Operationen die Kette der per Referenz zwischen den Protokollkomponenten weitergegebenen Nutzdaten. Hier kann die Anwendung von Application Layer Framing (siehe Abschnitt 6.3.2) eine Lösung bieten: Wird bereits beim Dienstnutzer das endgültige Paketformat des untersten Diensterbringers als Nutzdatentyp eingesetzt, wobei von jeder Protokollkomponente immer nur der jeweils für die Protokollabwicklung relevante Teil des Paketes belegt wird, so kann das Framing mit der beschriebenen Methode ohne Kopieroperationen abgewickelt werden (siehe Abb. 6-15). Offensichtlich geschieht dies um den Preis, dass auf allen Ebenen bis hin zum obersten Dienstnutzer die Paketstruktur des gesamten Protokollstacks in der Spezifikation präsent ist und so insbesondere mit den bereits in Abschnitt 5.1.4 dargestellten Problemen mit heterogenen Diensten das Abstraktionsniveau der Spezifikation erheblich beeinträchtigt wird (siehe auch Abb. 5-7 auf Seite 192).

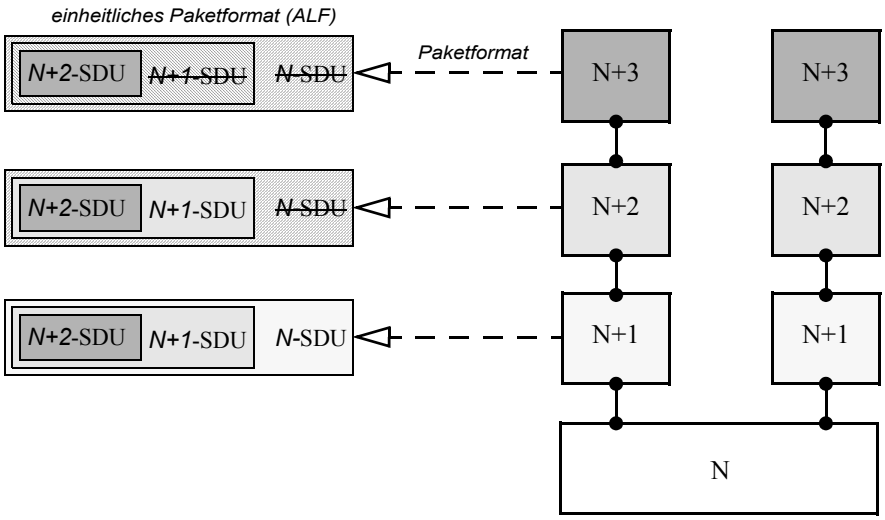


Abbildung 6-15: Einheitliches SDU-Format für ALF-orientierte Spezifikationen

In nächsten Abschnitt betrachten wir eine noch weiter gehende Anpassung der Spezifikation auf das Implementierungsziel, indem wir die Spezifikation und die Implementierung des Datentransportmechanismus explizit angeben.

31. Bei einer ggf. verketteten Weitergabe des Datums über mehrere Protokollkomponenten hinweg muss der zu transportierende gemeinsame Datenbereich mehrmals weitergereicht werden.

6.4.2 Pragmatische Ansätze

Zur Erlangung einer automatisch generierten Implementierung, deren Datenübertragungs-Effizienz sich mit der einer Handimplementierung messen kann, werden in der Praxis meist pragmatische Ansätze gewählt, bei denen die geschilderte Datenübertragungsproblematik nicht dem Implementierungsgenerator überlassen wird, sondern explizit durch den Einsatz primitiver Funktionen und Prozeduren direkt auf Implementierungsebene modelliert wird. Dadurch lassen sich die Beschränkungen der zu Grunde liegenden formalen Beschreibungstechnik bezüglich globaler Datenobjekte oder der Weitergabe von Datenobjekten per Referenz verhältnismäßig einfach umgehen, ohne den Implementierungsgenerator selbst modifizieren zu müssen.

Ein sehr einfacher Ansatz besteht darin, das Verbot der Weitergabe von Pointern in Interaktionsparametern zu umgehen, indem man mit Hilfe einer primitiven Funktion einen solchen Pointer reversibel in einen für den Implementierungsgenerator „*unverdächtigen*“ Typ umwandelt, ihn als Interaktionsparameter überträgt und ihn dann beim Empfänger durch eine entsprechende primitive Umkehrfunktion wiedergewinnt (siehe z. B. [Cat98]). So kann bei entsprechenden Implementierungs-Datentypgrößen ein Pointer mit Hilfe einer einfachen Konvertierungsoperation innerhalb einer primitiven Funktion[32] z. B. in einen Integer-Typ umgewandelt werden.

Die so gewonnene Möglichkeit, Pointer in Interaktionsparametern zu übertragen, ermöglicht es zum Beispiel, bei einem Dienstnutzer die zu übertragenden Nutzdaten direkt in einem dynamisch allozierten Datenobjekt zu erzeugen, um dann als (scheinbares) Nutzdatum den Pointer auf dieses Objekt durch alle Protokollschichten hindurch unverändert weiter zu reichen. Kopieroperationen wären dann nur noch für die von den einzelnen Protokollschichten hinzugefügten Paketheader und -Trailer erforderlich. Bei großen Nutzdatenobjekten wäre der Kopieraufwand für diese Daten (und natürlich mehr noch für den übertragenen Pointer selbst) praktisch zu vernachlässigen. In einer geschlossenen Spezifikation könnte dieser Pointer beim Empfänger wiedergewonnen werden, und die Auswertung der übertragenen Daten könnte wiederum direkt auf dem dynamisch allozierten Datenobjekt erfolgen. Soll hingegen eine Übertragung über einen realen (externen) Basisdienst erfolgen, so ist an der Schnittstelle zu diesem Dienst eine entsprechende Ersetzung des enthaltenen Pointers durch die referenzierten Daten erforderlich, sofern die übertragene Datenreferenz nicht sogar erst beim potenziellen Empfänger aufgelöst werden kann (z. B. durch Allokation des Datenpuffers in einem Shared-Memory-Bereich mit entsprechender Sichtbarkeit). Alternativ könnten auch global zugreifbare Speicherbereiche bereits durch eine primitive Funktion alloziert werden und dabei direkt z. B. mit einem `Integer` (ein sog. „*Handle*“) identifiziert werden. Mit einer weiteren primitiven Funktion könnte dann aus diesem `Integer` bei Bedarf ein Pointer auf den entsprechenden Datenpuffer gewonnen werden.

Offensichtlich können solche Datenübertragungsmechanismen (zumindest in einem geschlossenen Datenübertragungssystem) gänzlich ohne Kopieroperationen auf den Nutzdaten[33] einen Ende-zu-Ende-Übertragungsdienst realisieren. So kann der in Abschnitt 6.2.4 vorgestellte Datenübertragungsbenchmark mit genau 2 Kopieroperationen der Nutzlast beim Dienstnutzer einen Ende-zu-Ende-Transport realisieren:

---

32. in C z. B. mit
„`int pointer_as_int(void* p) {return (int)p;}`“,
oder etwas präziser in C++ mit
„`int pointer_as_int(void* p) {return reinterpret_cast<int>(p);}`“

33. mit Ausnahme der initialen Erzeugung der Nutzdaten beim Sender bzw. ihrer Verwertung beim Empfänger

- Kopie des zu versendenden Datums in den dynamisch allozierten Nutzlast-Puffer in Transition „`sending`“ von Modul „`USR`“
- Kopie des empfangenen Datums in lokalen Nutzlast-Puffer in Transition „`receiving`“ von Modul „`USR`“

Erkauft wird dieser Vorteil durch die Aufgabe der Trennung zwischen den Zustandsräumen der einzelnen Protokollautomaten bzw. der sie realisierenden Modulinstanzen. So kann vom Implementierungsgenerator nicht verhindert werden, dass verschiedene Modulinstanzen über Referenzen auf das selbe Datenobjekt direkt und unter Umgehung aller vorgegebenen Kommunikationsmechanismen miteinander kommunizieren (was ja der ursprüngliche Zweck des Verbotes der Übertragung von Pointern in Interaktionsparametern war).

Somit wird der in Estelle durch die vorsätzliche Beschränkung der Ausdrucksfähigkeit der Beschreibungstechnik erreichte Zwang zur Trennung der Zustandsräume hier – wenn überhaupt – nur durch die Einhaltung entsprechender Konventionen durch den Spezifizierer ersetzt. Wie wir in Abschnitt 6.3 bei den entsprechenden Handimplementierungstechniken gesehen haben, impliziert dies u. a. den Zwang zur globalen Koordination der Operationen auf diesen Datenobjekten. Im Gegensatz zu monolithisch erzeugten Handimplementierungen, die vom Implementierer weitgehend überschaut werden können, ist bei der hier beschriebenen, gemischt manuellen und automatischen Implementierung jedoch zusätzlich oft ein präzises Verständnis der Arbeitsweise der (weiterhin größtenteils automatisch erzeugten) Implementierung erforderlich, um die semantischen Implikationen der Handhabung dieser gemeinsamen Datenobjekte zu verstehen.

So könnte beispielsweise ein Implementierungsgenerator gemäß des zu Grunde liegenden semantischen Modells, das ja auf einer Trennung der Zustandsräume der Modulinstanzen beruht, Transitionen in Geschwistersmodulinstanzen oder verschiedenen Subsystemen echt parallel ausführen (z. B. durch mehrere Betriebssystem-Threads auf einem Multiprozessorsystem). Durch den möglichen Zugriff dieser nunmehr echt parallel ausgeführten Transitionen auf die nach oben geschildertem Muster realisierten gemeinsamen Datenobjekte kann dabei jedoch die Atomarität der Transitionsausführung verletzt werden.

Offensichtlich sind solche pragmatischen Umgehungen von Restriktionen der formalen Beschreibungstechniken zwar häufig leicht in die Spezifikation einzubringen und auf Implementierungsebene sehr effizient, sie sind semantisch aber nicht unproblematisch. Insbesondere wird dadurch der bereits bei den Handimplementierungstechniken unerwünschte Zwang zur genauen Abstimmung und globalen Koordination der Einzelkomponenten nun mit dem häufig nicht leicht zu verstehenden Modell des Implementierungsgenerators verknüpft, wobei die Semantik der von ihm generierten Implementierungen zudem noch auf einem wesentlich eingeschränkteren Kommunikationsmodell basiert.

In den folgenden Abschnitten werden wir uns deshalb der Frage widmen, ob und wie die Effizienz von Datenübertragungsmechanismen durch geeignete Erweiterungen der Ausdrucksfähigkeit der formalen Beschreibungstechniken verbessert werden kann. Dies setzt zwar (meist) einen entsprechend modifizierten Implementierungsgenerator voraus, entlastet den Spezifizierer dafür jedoch von der Frage, inwieweit mögliche Implementierungen die Semantik dieser Spezifikation einhalten oder nicht. Die Fragestellungen der Korrektheit und der Effizienz der generierten Implementierungen betreffen dann – wie es bei formalen Beschreibungstechniken zu erwarten sein sollte – allein die Entwickler der entsprechenden Implementierungswerkzeuge bzw. einer aus der Spezifikation manuell erzeugten Implementierung.

6.4.3 Formale Spezifikation gemeinsam genutzter Datenobjekte

Wie wir gesehen haben, ist die Übermittlung von Nutzdaten per Referenz auf ein von den beteiligten Komponenten gemeinsam genutztes Datenobjekt ein effektives Mittel zur Optimierung der Effizienz der Nutzdaten-Übertragung in Protokollspezifikationen. Ein nahe liegender Ansatz zur Übertragung dieses Konzepts auf formale Protokollspezifikationen ist der Einsatz explizit spezifizierter, gemeinsam genutzter Datenobjekte. Wir untersuchen in diesem Abschnitt zunächst die Ausdrucksmöglichkeiten von Standard-Estelle zur Spezifikation gemeinsam genutzter Datenobjekte zur Übertragung von Nutzdaten zwischen (als Modulinstanzen realisierten) Protokollkomponenten und werden uns dann mit möglichen Estelle-Erweiterungen beschäftigen.

In Standard-Estelle existieren zwei grundlegende Kommunikationsparadigmen zwischen Modulinstanzen: Neben der bereits eingehend behandelten asynchronen Übertragung von Interaktionen über externe Interaktionspunkte können auch gemeinsame Variablen zwischen einer Modulinstanz und ihren Sohnmodulinstanzen genutzt werden. Diese gemeinsamen Variablen werden als vom Sohnmodul exportierte Variablen in dessen Header definiert und erlauben den beiden beteiligten Modulinstanzen konkurrierend lesend wie auch schreibend darauf zuzugreifen. Die Zugriffskoordination erfolgt durch gegenseitigen Ausschluss bei der Transitionsausführung, wobei durch die Vater-Sohn-Priorität das Vatermodul ggf. Vorrang hat.[34]

Auf Grund der Definition gemeinsamer Variablen als Teil der Schnittstelle zu dem jeweiligen Vatermodul sind sie ausschließlich als gemeinsame Datenobjekte von jeweils genau diesen beiden Modulinstanzen[35] nutzbar, ein Datenaustausch zwischen einer größeren Zahl von Modulinstanzen erfordert also wiederum Kopieroperationen (z. B. durch das gemeinsame Vatermodul, siehe Abb. 6-16). Entsprechend sind sie auch bezüglich der Zielsetzung der Weitergabe von Nutzdaten per Referenz über mehr als zwei (als Modulinstanzen realisierte) Protokollkomponenten nicht effektiv.

Datenobjekte, die konkurrierend von einer größeren Zahl von Modulinstanzen direkt zugegriffen werden können, sind in Standard-Estelle nicht beschreibbar. Eine mögliche Estelle-Erweiterung zur Einführung eines solchen Konzepts müsste neben der reinen Spezifikation solcher Datenobjekte auch die Problematik der erforderlichen Koordination beim Zugriff auf diese Datenobjekte regeln.

Bei der Spezifikation von *Datenobjekten, die von mehreren Modulinstanzen gemeinsam genutzt werden können*, sind verschiedene Ansätze denkbar. Sinnvollerweise sollte sich die gemeinsame Nutzung der Modulinstanzhierarchie unterordnen, die Anlage solcher Objekte also in der Modulinstanz erfolgen, die für jede auf das Objekt zugreifende Modulinstanz (außer ihr selbst) direkt oder indirekt Vatermodulinstanz ist. Dies impliziert natürlich auch eine analoge Anforderung an die Hierarchie der beteiligten Moduldefinitionen. So könnte z. B. ein Datenpuffer,

34. Die an anderer Stelle vorgestellten Spracherweiterungen zur Beseitigung der Vater-Sohn-Priorität zwischen Modulinstanzen machen entweder unter Beibehaltung des gegenseitigen Ausschlusses bei der Transitionsausführung lediglich Vater und Sohn gleichberechtigt bei der Transitionsauswahl (*Independent-Module*-Erweiterung in Abschnitt 4.2.2.4), oder sie eliminieren die Möglichkeit eines konkurrieren Zugriffs durch ein gänzliches Verbot exportierter Variablen bei entsprechend attributierten Modulen (*Asynchronous-Process*-Erweiterung, siehe auch [BrGo94]).

35. Im Folgenden betrachten wir die Kind-Modulinstanz nicht als Teil ihrer Vater-Modulinstanz, sondern gemäß der Trennung der lokalen Zustandsräume als eigenständige Modulinstanz.

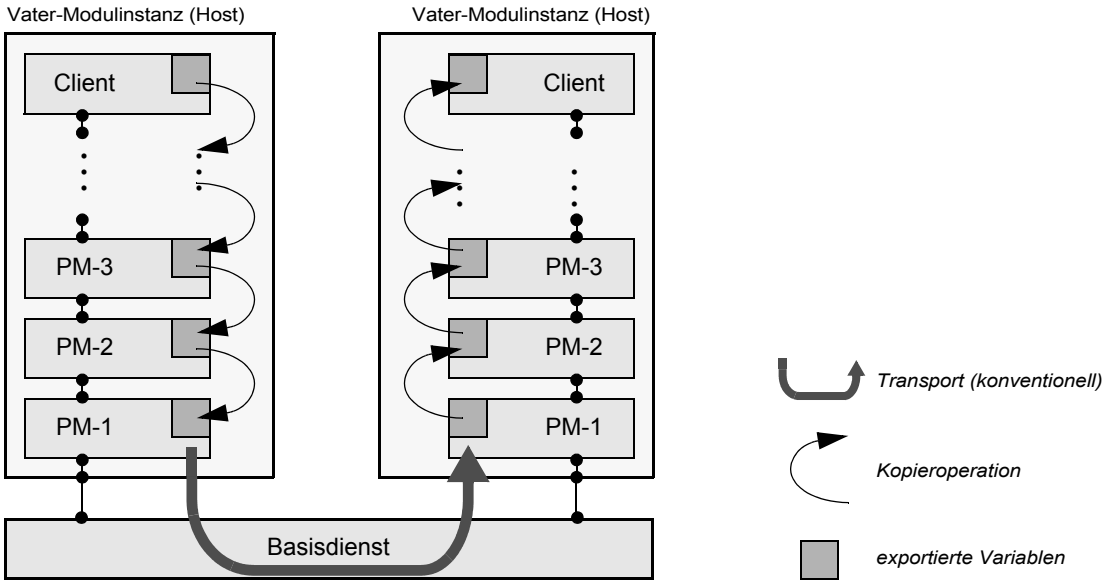


Abbildung 6-16: Host-interne Datenübertragung durch exportierte Variablen

der gemeinsam von mehreren Protokollmaschinen innerhalb eines Host-Moduls zum Datenaustausch per ALF genutzt wird (siehe Abb. 6-17), sinnvollerweise gerade in diesem Host-Modul angelegt werden.

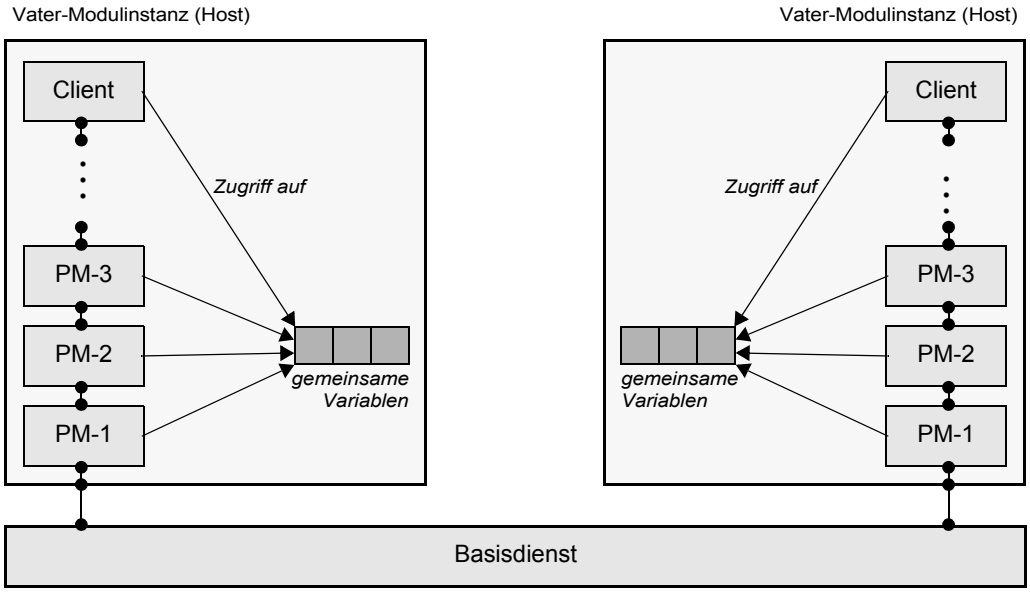


Abbildung 6-17: Datenübertragung durch gemeinsame Variablen mehrerer Modulinst.

Eine Einschränkung der gemeinsamen Nutzung von Datenobjekten durch mehrere Modulinstanzen auf nur eine Hierarchiestufe (also auf eine Modulinstanz und ihre unmittelbaren Kindmodulinstanzen) ist dabei offenbar nicht sinnvoll, da sie bei einer Unterstrukturierung der beteiligten Kind-Modulinstanzen in weitere Modulinstanzen letztere vom Zugriff auf die gemeinsamen Datenobjekte ausschließt (siehe auch die Struktur der XTP-Spezifikation in Abschnitt 6.1.1).

Um jedoch den Gültigkeitsbereich der gemeinsam genutzten Variablen auf die zur Problemlösung davon betroffenen Komponenten einschränken zu können, sollte die Ermöglichung des Zugriffs explizit erfolgen. Eine einfache Lösung wäre der explizite Import von im Vatermodul-

Kontext zugreifbaren (also dort definierten oder rekursiv von einem Vatermodul importierten) Variablen in das Kindmodul. Dieser Import könnte bereits im Modulheader (der externen Schnittstelle des Sohnmoduls zum Vatermodul) z. B. folgendermaßen definiert werden:

Beispiel 6.56: Estelle-Erweiterung zur Definition gemeinsam genutzter Datenobjekte

```
VAR packet_buffer: PACKET_TYPE;
{ ... }
MODULE pm_hdr activity;
  IMPORT VAR packet_buffer: PACKET_TYPE;
  IP ipup: { ... };
  END;
```

(Ende von Beispiel 6.56)

Die hier im Vatermodul definierte Variable[36] `packet_buffer` wird durch die „**`IMPORT VAR`**"-Deklaration im Modulheader von `pm_hdr` in allen daraus abgeleiteten Modulinstanzen zugänglich gemacht. Die syntaktischen Details dieser beispielhaft angegebenen Erweiterung sind nahe liegend und wir verzichten daher an dieser Stelle auf eine präzise Formalisierung.[37]

Betrachtet man nun die Auswirkungen solcher von mehreren Modulinstanzen gemeinsam zugreifbaren Variablen auf die formale Semantik und auf mögliche (dazu konforme) Implementierungen des Systems, so wird deutlich, dass diese wesentlich von den jeweiligen Modulattributierungen und den daraus resultierenden Konkurrenzsituationen der zugreifenden Modulinstanzen abhängen. Der konkurrierende Zugriff verschiedener Modulinstanzen auf das selbe Datenobjekt kann in folgenden grundlegenden Konkurrenzsituationen[38] erfolgen:

- zwischen (aktiven) Vater- und Sohnmodulinstanzen,
- zwischen Geschwistermodulinstanzen mit `ACTIVITY`- oder `SYSTEMACTIVITY`-attributiertem Vatermodul,
- zwischen Geschwistermodulinstanzen mit `PROCESS`- oder `SYSTEMPROCESS`-attributiertem Vatermodul,
- zwischen Modulinstanzen aus verschieden Subsystemen.

Eine Untersuchung der Estelle-Semantik bezüglich dieser Konkurrenzsituationen macht deutlich, dass die Interleaving-Semantik in all diesen Fällen den unmittelbar konkurrierenden Variablenzugriff dadurch entschärft, dass die Auswahl und Ausführung von Transitionen im seman-

36. Derart importierte Variablen sollten aus nahe liegenden Gründen „*not pointer-containing*" sein (siehe Abschnitte 7.4.3.2 und 7.3.4.2 aus [ISO97]).

37. Die beschriebene Erweiterung der Modulheader-Definition impliziert natürlich auch Anforderungen an die Umgebung der daraus abgeleiteten Modul-Instanzen, nämlich das Vorhandensein einer entsprechenden Variableninstanz. Auf der Basis von Standard-Estelle ist dies anhand der statischen Modulhierarchie bereits anhand der Modulheader-Definition prüfbar. In *Open-Estelle* (siehe Abschnitt 7) ist dies jedoch erst beim Import eines derartigen offenen Systems in eine konkrete Umgebung möglich. Dies und die Möglichkeit einer eigenständigen Prüf- und Implementierbarkeit eines solchen offenen Systems sind mithin die Gründe, warum in der vorgeschlagenen Erweiterung bei der „**`IMPORT VAR`**"-Deklaration der Typ der importierten Variablen mit angegeben wird (siehe Beispiel 6.56). Somit ist die vorgeschlagene Erweiterung mit Open-Estelle voll verträglich.

38. Die angegebenen Konkurrenzsituationen können aufgrund des Imports von gemeinsamen Variablen über mehrere Modulhierarchiestufen auch kombiniert auftreten, jedoch ergeben sich für die nachfolgenden Überlegungen dadurch keine fundamental neuen Situationen.

tischen Modell jeweils voll sequentialisiert erfolgt. Dadurch wird jeder mögliche konkurrierende Zugriff auf gemeinsame Zustandsraumkomponenten sequentialisiert, es sind also zunächst keinerlei Anpassungen an die Semantik von Standard-Estelle erforderlich.

Lediglich in den beiden letzten Fällen, den konkurrierenden Zugriffen aus verschiedenen Subsystemen[39] bzw. aus PROCESS-synchronisierten Geschwistermodulen, ergibt sich bei genauerer Betrachtung eine potentiell kritische Situation. In der Estelle-Semantik werden Subsysteme in jeweils voneinander unabhängigen Zyklen aus Transitionsauswahl und anschließender Transitionsausführung betrieben. Dabei kann z. B. die Ereignissequenz auftreten, dass in einem Subsystem A eine Transition T_A ausgewählt wird, in einem anderen Subsystem B eine (zuvor ausgewählte) Transition T_B gefeuert wird und schließlich im Subsystem A die ausgewählte Transition T_A gefeuert wird. Hängen nun die Transitionsauswahl und die Transitionsausführung von T_A von einer zwischen A und B gemeinsam genutzten Variablen ab und modifiziert T_B diese beim Feuern, so ergibt sich für T_A eine sichtbare Zustandsänderung zwischen Transitionsauswahl und -ausführung.

Betrachten wir dazu die in Beispiel 6.57 dargestellte Transition (als o. g. T_A), die in ihrer **PROVIDED**-Klausel die Schaltbedingung enthält, dass die Variable x von Null verschieden ist. In Standard-Estelle kann daher gefahrlos bei der Transitionsausführung von der Einhaltung dieser Bedingung ausgegangen werden und z. B. der Wert von x als Divisor bei einer mathematischen Berechnung dienen.

Beispiel 6.57: Annahme der Einhaltung von Schaltbedingungen bei der Transitionsausführung

```
TRANS
  PROVIDED x <> 0
    BEGIN
    y := 1/x;      {nur zulässig, wenn x <> 0}
    END
```

(Ende von Beispiel 6.57)

Wird x als importierte Variable jedoch aus einem anderen Subsystem heraus (durch die o. g. Transition T_B) zwischen Transitionsausführung und Transitionsauswahl von T_A auf Null gesetzt, so ist die o. g. Annahme einer Einhaltung der Schaltbedingung bei der Transitionsausführung verletzt und eine unerlaubte Division durch Null tritt auf.

Eine analoge Situation kann bei gemeinsamen Variablen zwischen Geschwistermodulinstanzen mit PROCESS- oder SYSTEMPROCESS-attributiertem Vatermodul auftreten, da hier nach der Transitionsauswahl in den Geschwistermodulen potentiell ebenfalls mehrere Transitionen gefeuert werden.[40]

Offenbar sind solche Situationen nicht wünschenswert und sollten durch geeignete Einschränkungen vermieden werden. Dazu bietet sich die Ergänzung einer der folgenden Anforderungen in die syntaktische bzw. semantische Definition importierter Variablen bzw. des Zugriffs auf diese an:

---

39. In Standard-Estelle ist dies nur beim Import einer gemeinsamen Variablen aus einem unattributierten Vatermodul möglich und damit aufgrund der resultierenden Inaktivität des Vatermoduls unkritisch.

40. Auch die Estelle-Erweiterung „*Asynchronous-Process*“ [BrGo94] kann analog zum Subsystem-Szenario die beschriebene Anomalie verursachen. In diesem Fall sind die im Folgenden für Subsysteme angegebenen Lösungen (soweit sinnvoll übertragbar) auch hier effektiv. Die in Abschnitt 4.2.2.4 eingeführte *Independent-Module*-Erweiterung ist dagegen in dieser Hinsicht unkritisch.

1. *Verbot des Imports von Variablen in Subsysteme bzw. in Module, die in* PROCESS*- oder* SYSTEMPROCESS*-attributierte Module eingeschachtelt sind.*

   Dadurch sind gemeinsame Variablen effektiv nur noch zwischen Modulinstanzen möglich, die im selben Subsystem alle gemeinsam ACTIVITY-synchronisiert sind. Da Transitionen in diesen Modulinstanzen syntaktisch voll sequentialisiert ausgewählt und ausgeführt werden, ist diese Konstellation beim konkurrierenden Zugriff auf gemeinsame Variablen besonders evident (siehe auch Implementierungsaspekte weiter unten).

2. *Verbot der Bezugnahme auf importierte Variablen in Transitionsklauseln.*

   Dadurch erfolgt die Auswahl von Transitionen unabhängig von den Inhalten gemeinsamer Variablen. Diese Lösung erscheint bezüglich der damit verbundenen Einschränkungen der Ausdrucksfähigkeit gerade dann sinnvoll, wenn die gemeinsamen Variablen ausschließlich zur (uninterpretierten) Weitergabe von Nutzdaten per Referenz dienen (siehe auch Abschnitt 6.4.2).

3. *Verbot der Manipulation von importierten Variablen.*

   Dadurch kann nur noch dasjenige Modul ändernd auf die gemeinsame Variable zugreifen, das ihre eigentliche Deklaration enthält und sie daher als einziger Zugriffsberechtigter nicht importiert hat. Diese Lösung ist nur bei lesendem Zugriff durch die Kindmodule bezüglich der Vermeidung von Kopieroperationen effektiv und macht zudem die Anwendung von gemeinsamen Variablen zwischen Subsystemen praktisch nutzlos, da hier die eigentliche Deklaration und damit auch alle Schreibzugriffe in einem unattributierten und damit inaktiven Modul erfolgen müssten.[41]

4. *Explizite oder implizite Koordination beim Zugriff auf importierte Variablen*.

   Beruht die Auswahl einer Transition auf dem Wert einer gemeinsamen Variable (z. B. x in Beispiel 6.57), und/oder erfolgt bei der Transitionsausführung potentiell ein (lesender oder schreibender) Zugriff auf diese, so wird der schreibende Zugriff auf diese Variable vom Zeitpunkt einer Transitionsauswahl bis zu ihrem Feuern für alle anderen Transitionen *gesperrt*. Dies könnte ggf. durch die Einführung einer obligatorischen *Lock-Klausel* (z. B. „**LOCK** x“ in Beispiel 6.58) explizit gemacht werden.

Beim letzten Punkt wären verschiedene Semantiken der Zugriffssperre denkbar, insbesondere vor dem Hintergrund der Frage, ob eine bestehende Sperre auf einer gemeinsamen Variablen die anderen (potentiell) darauf zugreifenden Transitionen bei der Auswahl explizit „nicht bereit“ macht und somit das Ergebnis der Transitionsauswahl beeinflusst oder lediglich die Transitionsauswahl des betroffenen Subsystems bis zur Freigabe ausgesetzt wird. Dieser Ansatz bietet eine explizite Koordination beim Zugriff auf gemeinsame Variablen, er beinhaltet jedoch diverse Detailprobleme bei der semantischen Einbettung (insbesondere bei PROCESS-synchronisierten Modulinstanzen). Wir verzichten an dieser Stelle auf eine eingehende Diskussion dieses Ansatzes.

Beispiel 6.58: LOCK-Klausel

```
TRANS
  LOCK x          {schließt ab Auswahl andere vom Zugriff auf x aus}
    PROVIDED x <> 0
```

41. Ändernde Zugriffe auf diese gemeinsamen Variablen wären in dieser Konstellation nur während der Initialisierung des definierenden (d.h. inaktiven) Moduls möglich.

```
BEGIN
y := 1/x;    {hier gilt dann immer noch "x <> 0"}
END
```

(Ende von Beispiel 6.58)

Zusammenfassend kann man also bisher festhalten, dass gemeinsame Variablen, wie sie hier vorgestellt wurden, unter den genannten Restriktionen mit der Standard-Estelle-Semantik vereinbar sind.

Bei der Untersuchung von *Implementierungsaspekten* solcher Spezifikationen ergeben sich jedoch weitere inhaltlich motivierte Einschränkungen. So ist z. B. eine der Zielsetzungen der semantisch weitgehend unabhängigen Arbeitsweise der einzelnen Subsysteme in Standard-Estelle die Möglichkeit, die Implementierung der Subsysteme auf physisch getrennten (und daher sinnvollerweise nur *asynchron* kommunizierenden) Knoten eines Kommunikationssystems semantikkonform realisieren zu können. Die Möglichkeit zur Spezifikation von gemeinsamen Variablen zur direkten Kommunikation zwischen Subsystemen erzwingt potentiell eine über die spezifizierte Kommunikation hinausgehende Synchronisation[42] dieser Subsysteme auf Implementierungsebene. Dies spricht für einen Ausschluss der Möglichkeit gemeinsamer Variablen zwischen Subsystemen (s. o.).

Ein solcher Ausschluss entspricht ebenfalls der Motivation für das Verbot exportierter Variablen bei asynchron arbeitenden Modulinstanzen gemäß der *Asynchronous-Process-Erweiterung* [BrGo94]. Derartige asynchrone Modulinstanzen arbeiten semantisch analog zu Subsystemen und entsprechend sollte dort auch der Import von Variablen an der äußeren Schnittstelle asynchron arbeitender Instanzen syntaktisch untersagt werden.

Weitere inhaltlich motivierte Einschränkungen ergeben sich auch innerhalb von Subsystemen, wenn bei der Implementierung Verfahren unterstützt werden sollen, die auf Ausführungsebene stärker konkurrierend arbeiten, vom Ergebnis her jedoch der Interleaving-Semantik entsprechen. In Standard-Estelle könnte dies z. B. durch eine echt parallele[43] Ausführung von Transitionen in Geschwistermodulinstanzen mit einem `PROCESS`- bzw. `SYSTEMPROCESS`-attributierten Vatermodul geschehen. Aufgrund der beschränkten Auswirkungen solcher konkurrierenden Operationen sind in Standard-Estelle in einigen solcher Situationen sogar echt parallel arbeitende Implementierungen unter Einhaltung der formalen Semantik möglich (siehe auch [FiHo94]). Die beschriebene Erweiterung hemmt u. U. die Anwendbarkeit solcher Implementierungen aufgrund der möglichen Verletzung der Interleaving-Semantik. So könnten konkurrierend ausgeführte Transitionen aus verschiedenen Subsystemen beim unkoordinierten Zugriff auf eine gemeinsame Variable Ergebnisse erzeugen, die durch keine sequentielle Ausführung der Interleaving-Semantik abgedeckt sind.

Um es nochmals zu betonen: Die beschriebenen Effekte sind rein implementierungstechnischer Natur; bezüglich der Standard-Estelle-Semantik ist die Erweiterung unter den oben diskutierten Restriktionen unproblematisch. Es zeigt sich an diesen implementierungstechnischen Einschränkungen jedoch, dass der Grad *spezifikationsinhärenter* (und meist auch *probleminhären-*

42. Solche nicht von der Spezifikation vorgegebenen Synchronisationen wurden auch bei verteilten Implementierung durch DINGO [SiSt93] eingesetzt, um die interne Kooperation der (nicht zwingend auf Subsystemebene partitionierten) verteilten Implementierungen zu erreichen.

43. z. B. auf mehreren Prozessoren eines Multi-Prozessor-Systems; die Problematik ergibt sich hier aus der auf Implementierungsebene vorliegenden zeitlichen Ausdehnung der (semantisch atomaren) Transitionsauswahl und Transitionsausführung (siehe auch Abschnitte 5.3.5, 9.6.4 und 9.6.5 von [ISO97])

ter) *Nebenläufigkeit* durch die gemeinsamen Zustandsraumkomponenten reduziert wurde. Insbesondere bedingen die geforderten Einschränkungen, dass die Datenübertragung über gemeinsame Variablen meist nur abschnittsweise (z. B. nur innerhalb des Subsystems eines Kommunikationsknotens, siehe Abb. 6-16) genutzt werden kann.

Dies und die schon bei anderen Lösungen (siehe Abschnitt 6.3 und Abschnitt 6.4.2) bemängelte Notwendigkeit einer expliziten Verwaltung[44] der in den gemeinsamen Variablen abgelegten Daten und die damit verbundene enge organisatorische Kopplung der beteiligten Komponenten sind die wesentlichen Nachteile beim Einsatz gemeinsamer Variablen zur Datenübertragung.

6.4.4 Leichtgewichtsmodule

Estelle bietet zur Komponenten-Strukturierung von Spezifikationen ausschließlich hierarchische Systeme von Modulen bzw. Modulinstanzen. Diese Module werden, wie wir bereits in Abschnitt 6.1.1 anhand der XTP-Spezifikation gesehen haben, auch zur Unterstrukturierung von lokalen Komponenten wie z.B einer Protokollmaschine eingesetzt. Ein Nachteil des Einsatzes von Modulen zur Unterstrukturierung von lokalen Komponenten ist die damit einhergehende „Schwergewichtigkeit“ dieses Strukturierungsmechanismus, der ursprünglich zur Abgrenzung verteilter, asynchron kommunizierender und nebenläufig agierender Teilsysteme gedacht war.

In [Bre97] wird als Ansatz zur Verringerung dieser Schwergewichtigkeit die Einführung sogenannter Leichtgewichtsmodule („*Light-Weight Sub-Modules*“) diskutiert. Motivation war hier im Wesentlichen die Reduktion des Synchronisationsaufwands zwischen diesen Teilkomponenten z. B. einer Protokollmaschine, indem das Modul in einem syntaktischen Transformationsprozess (siehe Abb. 6-18) mit seinen Kindmodulen verschmolzen wurde. Das resultierende Gesamtmodul ist dann (mit gewissen Einschränkungen bezüglich einiger Synchronisationsaspekte) äquivalent zum Ausgangs-Modulsystem.

Die Motivation in [Bre97] für diese Transformation basierte ganz wesentlich auf der Beobachtung, dass der vom verwendeten Implementierungsgenerator DINGO [SiSt93] erzeugte Code in seiner Laufzeit-Effizienz durch solche Verschmelzungen erheblich verbessert werden konnte. Dieses Ergebnis ist jedoch spezifisch für DINGO und nicht verallgemeinerbar.

Die Grundidee solcher Leichtgewichtsmodule ist jedoch in erweiterter Form ein interessanter Ansatz zur Realisierung gemeinsamer Variablen zwischen diesen zu verschmelzenden Unterkomponenten: Spezifiziert man einen Modul-Teilbaum mit von den Kindmodulen (unter Einsatz einer geeigneten Estelle-Erweiterung, s. o.) gemeinsam genutzten Variablen und verschmilzt man anschließend diese Module, die Zugriff auf eine gemeinsame Variable nehmen, zu *einem* Estelle-Modul, so werden die ursprünglich gemeinsam genutzten Variablen nunmehr lokale Variablen des entstandenen Moduls. Die fortgesetzte Anwendung solcher Transformationen liefert schließlich eine Standard-Estelle-Spezifikation.

Der Vorteil dieser Vorgehensweise gegenüber einer unmittelbaren Spezifikation aller auf die gemeinsame Variable zugreifenden Teilmodule in einem einzigen Modul ist die bessere Übersichtlichkeit bei der strukturierten Spezifikation der Teilkomponenten. Wesentliche Nachteile

44. Die (im Sinne einer Verarbeitungs-*Pipeline*) konkurrierende Verarbeitung verschiedener Pakete durch mehrere Modulinstanzen eines Protokollstacks macht eine dynamische (und auf Spezifikationsebene explizit anzugebende) Verwaltung der in globalen Variablen abgelegten Paketinhalte erforderlich.

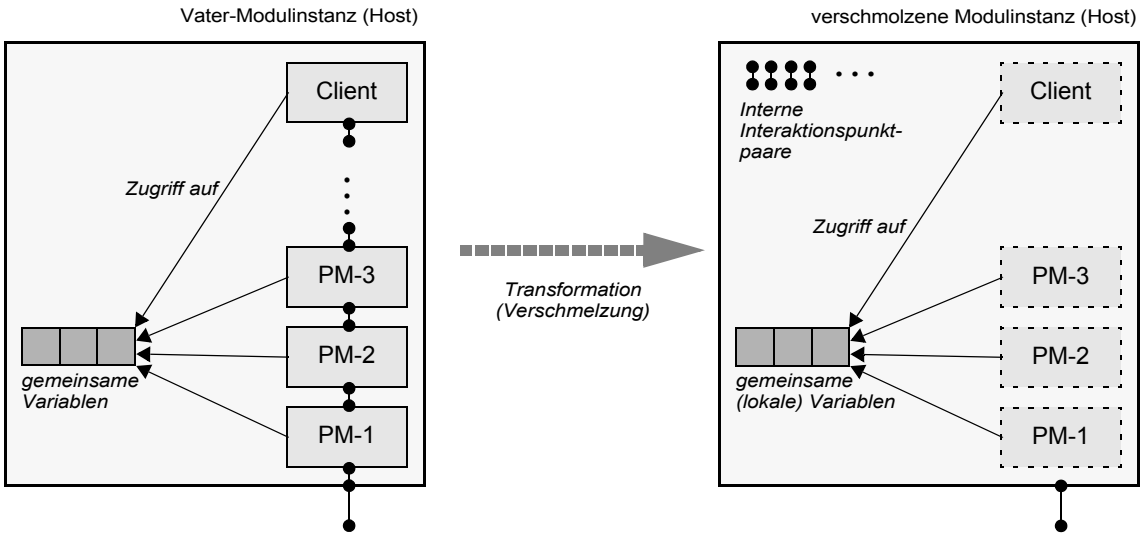


Abbildung 6-18: Gemeinsame Variablen bei der Verschmelzung von Leichtgewichtsmodulen

sind die reduzierte Nebenläufigkeit der verschmolzenen Komponenten und mangelnde Anwendbarkeit der Methode bei dynamischen Modulinstanzhierarchien, wie sie z. B. bei der Estelle-Spezifikation von XTP zur dynamischen Erzeugung und Termination von Kontext-Modulinstanzen eingesetzt werden (siehe Abschnitt 6.1.1).

Diese Nachteile wiegen besonders schwer vor dem Hintergrund, dass bei der Transformation alle Protokollkomponenten, die Zugriff auf gemeinsam genutzte Variablen haben, schließlich in eine Modulinstanz zusammengeführt werden müssten. Beim Einsatz der gemeinsamen Variablen zur Ende-zu-Ende-Übertragung von Nutzdaten würde dies große Teile des Systems – wenn nicht sogar in letzter Konsequenz das ganze System – zu einer Modulinstanz fusionieren. Dies reduziert nicht nur die spezifikationsinhärente Nebenläufigkeit unangemessen, es behindert auch durch die hohe Komplexität des resultierenden Moduls u. U. die automatische Gewinnung einer effizienten Implementierung (siehe Abschnitt 4.3).

6.4.5 Erweiterung „Explizite Referenzübergabe“

Wie wir in den vorangegangen Abschnitten gesehen haben, ist die explizite Spezifikation globaler Datenobjekte mit zum Teil erheblichen konzeptionellen Problemen bezüglich der resultierenden Nebenläufigkeit zwischen den darauf konkurrierend zugreifenden Modulinstanzen verbunden. Ursächlich dafür ist der Umstand, dass prinzipiell alle Modulinstanzen, für die diese globalen Variablen *sichtbar* sind, zu jedem Zeitpunkt auf diese *zugreifen* können. Es existiert also *keine explizite Zuordnung* der globalen Variablen (und der darin enthaltenen zu transportierenden Daten) zu derjenigen Modulinstanz, die im Sinne eines sukzessiven Datentransports zu einem gegebenen Zeitpunkt (möglicherweise exklusiven) Zugriff benötigt. Dies wird besonders deutlich, wenn man solche globalen Variablen zur Ablage von Nutzdaten für einen Ende-zu-Ende-Transport zwischen verschiedenen Knoten eines Netzwerks modelliert. Obwohl solche Systeme auf Implementierungsebene auf Grund der ausschließlich asynchronen Kommunikation durch das Netzwerk keinen gemeinsamen Speicher besitzen[45], könnte mit den oben genannten Mechanismen auf Spezifikationsebene ein solcher gemeinsamer Speicher in Form gemeinsamer Variablen zur Datenübertragung modelliert werden, was die (insbesondere automatische) Ableitung einer konformen Implementierung erheblich erschwert oder sogar unmöglich macht.[46] Andererseits sollten Komponenten, die auf *Implementierungsebene* gemeinsamen

Speicher besitzen, auch auf *Spezifikationsebene* zur Datenübertragung solche Mechanismen (wie z. B. die oben genannten gemeinsamen Variablen) einsetzen, die eine Abbildung der Datenübertragungsmechanismen auf diesen gemeinsamen Speicher zur Vermeidung von Datenkopieroperationen unterstützen. Eine Abgrenzung, wo solche gemeinsamen Variablen also konzeptionell sinnvoll sind und wo nicht, ist offensichtlich nur im Hinblick auf ein konkretes Implementierungsszenario möglich und somit auf der Ebene einer abstrakten Protokoll-Spezifikation nur bedingt wünschenswert.

Wir suchen also ein Ausdrucksmittel zur Beschreibung der Weitergabe von Daten zwischen Protokollkomponenten, das auf Spezifikationsebene *hinreichend abstrakt* ist, um auf Implementierungsebene keine spezifischen Implementierungsmechanismen (wie z. B. gemeinsamen Speicher oder das Kopieren der Daten) zu erzwingen, gleichzeitig aber die automatische Ableitung einer bezüglich der Datenübertragungs- und Synchronisationsproblematik effizienten Implementierung unterstützt.

Die im Folgenden vorgestellte Lösung dieses Problems besteht aus einer geradezu subtilen syntaktischen und semantischen Erweiterung von Estelle, die auf überraschend einfache Art und Weise hohe Abstraktion auf Spezifikationsebene und effiziente Implementierbarkeit (sowohl mit als auch ohne gemeinsamen Speicher) verbindet.

6.4.5.1 Formale Definition

Die Lösung besteht darin, das Verbot der Übertragung von Pointern als Interaktionsparameter teilweise aufzuheben, dieser Übertragung jedoch eine spezifische Semantik zu geben, die die Entstehung gemeinsamer Variablen zwischen verschiedenen Modulinstanzen effektiv verhindert.

Dazu wird zunächst syntaktisch die Übertragung von Pointer-Typen als Interaktionsparameter erlaubt. Betrachten wir zunächst die Constraints einer „Channel Definition“ des Estelle-Standards [ISO97]:

Definition 6.8: Constraints einer „Channel Definition“ (Abschnitt 7.3.4.2 aus [ISO97])

[...]

A type shall be designated pointer-containing if it is a POINTER-TYPE or it is a STRUCTURED-TYPE possessing a COMPONENT-TYPE that is pointer-containing.

A VALUE-PARAMETER-SPECIFICATION of an INTERACTION-DEFINITION of a CHANNEL-DEFINITION shall not contain a TYPE-IDENTIFIER denoting a pointer-containing type.

(Ende von Definition 6.8)

---

45. Mit Hilfe von Hardware-Einrichtungen zur Verwaltung von virtuellem Speicher kann auch auf Implementierungsebene über Kommunikationsnetzwerke hinweg *gemeinsamer Speicher* nachgebildet werden, indem Zugriffe auf solche Seiten (sofern sich der betroffene Inhalt nicht bereits auf dem lokalen Knoten befindet) vom Betriebssystem erkannt und durch den Transport der betroffenen Seiten über das Kommunikationsnetzwerk auf diesen Knoten aufgelöst werden. Offensichtlich führt dieses Verfahren bei konkurrierenden Zugriffen verschiedener Knoten auf die selben Seiten jedoch zu einem erheblichen Kommunikationsaufwand. Durch die meist grobe Granularität der Seitenverwaltung tritt dieser Effekt auch bei konkurrierenden Zugriffen auf disjunkte Speicherbereiche auf, die (u. U. zufällig) die selbe Seite belegen.

46. Dieser Aspekt spricht nochmals für den oben bereits diskutierten Ausschluss von gemeinsamen Variablen zwischen Subsystemen (siehe Abschnitt 6.4.3).

Wir erweitern nun die Syntax einer Kanaldefinition folgendermaßen, wobei der erste Aufzählungspunkt (a) äquivalent zur bisherigen Definition ist und der zweite Punkt (b) die eigentliche Erweiterung einführt.

Definition 6.9: Syntax der Estelle-Erweiterung als Erweiterung von Def. 6.8

[...]

A type shall be designated pointer-containing if it is a POINTER-TYPE or it is a STRUCTURED-TYPE possessing a COMPONENT-TYPE that is pointer-containing.

A VALUE-PARAMETER-SPECIFICATION of an INTERACTION-DEFINITION of a CHANNEL-DEFINITION shall contain only TYPE-IDENTIFIERs where each is denoting a type that

(a) is not pointer-containing
or

(b) is a POINTER-TYPE to a type that is not pointer-containing.

(Ende von Definition 6.9)

Durch diese syntaktische Erweiterung kann nun als aktueller Interaktionsparameter nicht nur ein Datum, dessen Typ nicht pointer-containing ist, übergeben werden, sondern alternativ auch ein *Zeiger auf solch ein Datum*. Wir illustrieren dies am folgenden Beispiel:

Beispiel 6.59-a: Anwendung der syntaktischen Estelle-Erweiterung (Kanal-Definition)

```
TYPE T = {...};                    { T ist nicht pointer-containing }
TYPE TP = ^T;                      { TP ist Zeiger auf T }

CHANNEL ch(sender, receiver);
  BY sender: msg(p2: TP);          { p2 ist Zeiger auf T }
{...}
```

(Ende von Beispiel 6.59-a)

Um zu verhindern, dass durch die Weitergabe von Zeigern als Interaktionsparameter effektiv von mehreren Modulinstanzen konkurrierend zugreifbare Datenobjekte entstehen, gestalten wir die Semantik der Übergabe eines Zeigers als aktuellen Parameter einer Output-Anweisung abweichend von der Übergabe der üblichen, nicht pointer-containing Typen:

Definition 6.10: (Informelle) Semantik der Estelle-Erweiterung von Def. 6.9

*Passing a value p1 of a **pointer-type** as an actual parameter to an output-statement in module instance m1, shall match the following semantics:*

(i) if p1 is not a pointer to a valid (dynamically allocated) variable in the context of m1 then the result is undefined.

(ii) if p1 is a pointer to a valid (dynamically allocated) variable in the context of m1 then

- *the value saved as stored interaction parameter during transport shall be the same as the value of the object pointed to by p1 at the execution of the output statement and*
- *as side effect of the output statement the (dynamically allocated) variable pointed to by p1 shall be disposed (as if it was passed to the required procedure dispose as in "`dispose(p1)`")*

In case (ii), receiving the resulting interaction in (the same or a different) module instance m2 with the value p2 (of the same type as p1) of the corresponding formal parameter, shall match the following semantics:

- *p2 shall be a pointer to a new valid (dynamically allocated) variable of the type pointed to by p1 in the context of m2 (as if it was created there by "`new(p2)`") and*
- *the value of the object pointed to by p2 shall be the value saved as stored interaction parameter during transport (i.e. the same as the value of the object pointed to by p1 at the execution of the output statement)*

(Ende von Definition 6.10)

Die Semantik der Weitergabe eines Pointer-Wertes als Interaktionsparameter unterscheidet also zunächst beim Versenden einer Interaktion durch ein Output-Statement, ob der übergebene Zeiger ein gültiges (dynamisch alloziertes) Datenobjekt referenziert *(ii)* oder nicht *(i)*. Der Fall *(i)*, die Übergabe eines ungültigen Zeigerwertes (z. B. er hat den Wert `nil` oder das referenzierte Objekt wurde bereits freigegeben) als aktueller Parameter, ist naheliegenderweise als *Fehler* zu betrachten, die Semantik ist daher undefiniert.[47]

Der Kernpunkt der vorgestellten Erweiterung tritt jedoch im Fall *(ii)* zu Tage, nämlich dass als aktueller Parameter beim Versenden einer Interaktion ein gültiger Zeiger auf ein (dynamisch alloziertes)[48] Datenobjekt übergeben wird. Hier wird abweichend von der bisherigen Estelle-Semantik der Übertragung von Interaktionsparametern nicht einfach nur der übergebene *Wert des Zeigers* unverändert zum Empfänger der Interaktion übertragen, da auf diese Weise Sender und Empfänger der Interaktion gleichzeitig eine Referenz auf das selbe Datenobjekt besitzen könnten und so ein konkurrierender Zugriff aus verschiedenen Modulinstanzen heraus möglich wäre. Stattdessen wird an Stelle der *Referenz* auf das Datenobjekt der *Wert des vom Zeiger referenzierten Objektes* selbst übertragen, und beim Empfänger als ein neues (dynamisch alloziertes) Datenobjekt mit dem übertragenen Wert angelegt. Der Empfänger der Interaktion erhält (passend zur Interaktions-Signatur) schließlich den Zeiger auf dieses neue Datenobjekt als Interaktionsparameter. Als Seiteneffekt der `output`-Anweisung wird das ursprüngliche (dynamisch allozierte) Datenobjekt im Kontext des Senders freigegeben, als ob der Pointer an die Funktion `dispose`[49] übergeben worden wäre. Dadurch hat in jedem Fall zu einem Zeitpunkt immer nur *höchstens eine Modulinstanz* Zugriff auf das referenzierte Datenobjekt (siehe auch Abb. 6-19 auf Seite 296).

Wir betrachten dazu das folgende Beispiel als Fortsetzung zu Beispiel 6.59-a:

---

47. Dies entspricht der Estelle-Semantik der Dereferenzierung eines solchen Zeigers (siehe Abschnitt 6.5.4, Annex C von [ISO97]).

48. Wir gehen hier davon aus, dass derartige dynamisch allozierte Datenobjekte durch Aufruf der Funktion `new(p)` (siehe Abschnitt 6.6.5.3 des Annex C von [ISO97]) bzw. äquivalent durch den hier definierten Empfang eines Pointer-Wertes als Interaktionsparameter erzeugt wurden.

49. `new(p)` und `dispose(q)` sind so genannte „*required functions*“ (siehe Abschnitt 6.6.5.3 des Annex C von [ISO97]). Wir verzichten an dieser Stelle auf die Diskussion der dort ebenfalls definierten erweiterten Varianten `new(p, c1, ..., cn)` und `dispose(q, k1, ..., kn)` und gehen in diesem Abschnitt von der exklusiven Nutzung der einstelligen Varianten der Prozeduren aus.

Beispiel 6.59-b: Anwendung der syntaktischen Estelle-Erweiterung (siehe Beispiel 6.59-a)

Die von der Erweiterung betroffenen Teile sind unterstrichen dargestellt.

```
IP ip_s: ch(sender);
   ip_r: ch(receiver);
{...}

TRANS
  VAR p1: TP;
    NAME sending:
    BEGIN
      new(p1);                { alloziere p1^ }
      p1^ := {...};
      OUTPUT ip_s.msg(p1);    { reicht Wert von p1^ weiter }
                              { und gibt damit p1 implizit frei }
    END;

TRANS
  WHEN ip_r.msg{p2: TP}       { empfange p2^ (neu alloziert) }
    NAME receiving:
    BEGIN
      {... werte p2^ aus ...}
      dispose(p2);            { gibt p2^ (irgendwann) explizit frei }
    END;

TRANS
  WHEN ip_r.msg{p2: TP}       { empfange p2^ (neu alloziert) }
    NAME forwarding:
    BEGIN
      {... werte p2^ aus ...}
      OUTPUT ip_s.msg(p2);    { reicht Wert von p2^ weiter }
                              { gibt damit p2 implizit frei }
    END;
```

(Ende von Beispiel 6.59-b)

In der ersten Transition (`sending`) wird das Versenden eines (gültigen) Zeigers als Interaktionsparameter demonstriert, indem in der Interaktion zunächst durch Aufruf der Funktion `new` ein neues Datenobjekt dynamisch alloziert wird, dieses Objekt initialisiert wird und schließlich der Zeiger auf dieses Objekt als Parameter an eine entsprechende Interaktion übergeben wird. Dabei wird das zuvor dynamisch allozierte Datenobjekt als Seiteneffekt der `output`-Anweisung freigegeben, sodass weitere Zugriffe auf das Objekt (z. B. durch Dereferenzierung des Zeigers) nicht mehr erlaubt sind.

Die zweite Transition (`receiving`) empfängt eine derart erzeugte Interaktion, wobei der dabei empfangene Zeiger (`p2`) auf ein neues (dynamisch alloziertes) Datenobjekt zeigt, das den Wert des oben genannten (bei der Erzeugung der Interaktion freigegebenen) Datenobjekts zum Zeitpunkt der Erzeugung der Interaktion besitzt. Über diesen Zeiger kann im Kontext des Empfängers auf das derart übertragene Datum zugegriffen werden. Schließlich kann das (dynamisch allozierte) Datenobjekt später durch expliziten Aufruf von `dispose` oder – wie in der dritten Transition (`forwarding`) demonstriert – durch Weitergabe des Pointers als Interaktionsparameter wieder freigegeben werden.

6.4.5.2 Transformation nach Standard-Estelle

Die vorgestellte Erweiterung kann durch eine verhältnismäßig einfache syntaktische Transformation semantikkonform auf Standard-Estelle abgebildet werden. Dazu werden

(i) in den Kanaldefinitionen jeweils der Pointer-Typ durch den referenzierten Typ[50] ersetzt,

(ii) beim Versenden von Interaktionen über solche Kanäle an Stelle des Zeiger-Wertes („`p`“) der referenzierte Wert („`p^`“) als Interaktionsparameter übergeben und nach Ausführung[51] der Output-Anweisung der Zeiger-Wert an die Funktion `dispose` übergeben („`dispose(p)`“) und

(iii) beim Empfang von Interaktionen über solche Kanäle am Anfang des Transitionsblocks die nunmehr als Wert übergebenen Interaktionsparameter in ein explizit durch Aufruf der Funktion `new` erzeugtes Datenobjekt kopiert und die dadurch entstehenden Zeiger auf (dynamisch allozierte) Datenobjekte mit dem übertragenen Inhalt in gewohnter Weise weiterverwendet.[52]

Wir illustrieren das Ergebnis dieses Transformationsvorgangs an folgendem Beispiel:

Beispiel 6.60: Transformation von Beispiel 6.59-a und 6.59-b nach Standard-Estelle

Die modifizierten oder hinzugefügten Teile sind unterstrichen dargestellt.

```
TYPE T = {...};
TYPE TP = ^T;

CHANNEL ch(sender, receiver);
  BY sender: msg(p2aux: T);     { früher: "p2: TP" }
{...}

IP ip_s: ch(sender);
   ip_r: ch(receiver);
{...}

TRANS
  VAR p1: TP;
    NAME sending:
    BEGIN
      new(p1);
      p1^ := {...};
```

50. Der referenzierte Typ ist nicht *pointer-containing* (s. o.) und kann deshalb als Interaktionsparameter in Standard-Estelle dienen.

51. Dazu wird ggf. die Output-Anweisung zusammen mit den hinzugefügten dispose-Anweisungen in einen neu hinzugefügten STATEMENT-BLOCK („**BEGIN**“ ... „**END**“) eingeklammert.

52. Dazu muss der formale Interaktionsparameter bei der Kanaldefinition geeignet umbenannt werden, damit beim Empfang der transformierten Interaktion in einer Transition mit entsprechender `WHEN`-Klausel eine lokale Variable vom entsprechenden Pointer-Typ mit dem ursprünglichen Namen angelegt werden kann, so dass diese schließlich den von `new` zurück gelieferten Wert aufnehmen und während der Transitionsausführung in der oben beschriebenen Weise als Zeiger auf das übertragene Datum verwendet werden kann.

```
      OUTPUT ip_s.msg(p1^);      { früher: "(p1)" }
      dispose(p1);               { neu }
    END;

TRANS
  WHEN ip_r.msg{p2aux: T}        { früher: "p2: TP" }
    VAR p2: TP;                  { neu }
      NAME receiving:
      BEGIN
        new(p2);                 { neu }
        p2^ := p2aux;            { neu }
        {...}
        dispose(p2);
      END;

TRANS
  WHEN ip_r.msg{p2aux: T}        { früher: "p2: TP" }
    VAR p2: TP;                  { neu }
      NAME forwarding:
      BEGIN
        new(p2);                 { neu }
        p2^ := p2aux;            { neu }
        {...}
        OUTPUT ip_s.msg(p2^);    { früher: "(p2)" }
        dispose(p2);             { neu }
      END;
```

(Ende von Beispiel 6.60)

6.4.5.3 Implementierungsaspekte

Der eigentliche Nutzen der vorgestellten Erweiterung wird jedoch erst vor dem Hintergrund einer unmittelbaren Implementierung[53] der entsprechend erweiterten Spezifikationen offenbar: Durch die Erweiterung kann bereits auf Spezifikationsebene die Übertragung von Nutzdaten zwischen Modulinstanzen mittels Weitergabe einer Referenz erfolgen, ohne dass dabei gleichzeitig mehrere Referenzen aus verschiedenen Modulinstanzen auf das selbe Datenobjekt entstehen (siehe Abb. 6-19). Die Implementierung einer derart spezifizierten Datenübertragung per Referenz kann dabei (abhängig vom jeweiligen Implementierungsszenario)

(i) durch eine *Kopieroperation* auf den zu übertragenden Daten (einschließlich der Freigabe des gesendeten Datenobjekts und der Allokation eines neuen Datenobjekts mit gleichem Inhalt beim Empfänger) oder

(ii) durch die schlichte *Weitergabe der Referenz* (also des Pointers) von einer Modulinstanz an die andere implementiert werden (wenn sendende und empfangende Modulinstanz im selben Speicher-Adressraum operieren bzw. das referenzierte Objekt in einem Shared-Memory-Bereich der beiden Modulinstanzen angelegt wurde).

---

53. d.h. einer Implementierung ohne Umweg einer Transformation nach Standard-Estelle; alternativ wäre auch die Erkennung und spezifische Implementierung der oben gezeigten und als *Muster* identifizierbaren transformierten Form der Spezifikation denkbar.

Im letzteren Fall stellt die Einhaltung der oben eingeführten informellen Semantik der Erweiterung sicher, dass die sendende Modulinstanz nach dem Verschicken der Interaktion keinen weiteren Zugriff mehr auf das referenzierte Datenobjekt hat. Auf der Empfängerseite kann entsprechend eine Implementierung gefahrlos auf die in der Semantik beschriebene Allokation eines neuen Datenobjekts und anschließende Übertragung des Inhalts des ursprünglichen Datenobjekts in das neue Datenobjekt verzichten und direkt auf dem von der sendenden Modulinstanz ursprünglich allozierten und als Interaktionsparameter per Referenz weitergegebenen Datenobjekt weiterarbeiten. Es hat also in jedem Fall zu einem Zeitpunkt immer nur *höchstens eine Modulinstanz* Zugriff auf das referenzierte Datenobjekt (siehe Abb. 6-19).

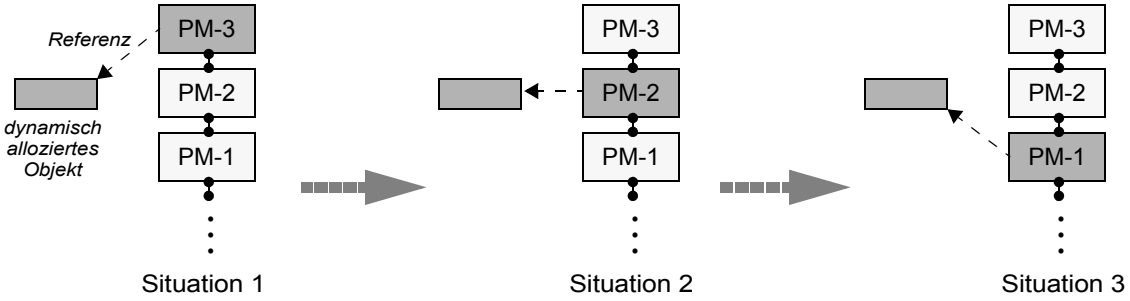


Abbildung 6-19: Weitergabe einer (exklusiven) Referenz als Interaktionsparameter

Die vorgestellte Implementierungsmethode ermöglicht es also, die Datenübertragung auf Implementierungsebene *spezifikationsnah* und gleichzeitig im Rahmen der Existenz eines gemeinsamen Adressraums zwischen verschiedenen Modulinstanzen *optimal* bezüglich der Anzahl der physischen Kopieroperationen zu implementieren. Dies geschieht auf Spezifikationsebene ohne Einschränkung der Nebenläufigkeit zwischen Modulinstanzen oder anderweitiger Anpassungen an die spezifischen Eigenschaften möglicher Implementierungen.

Als Referenzimplementierung für die vorgestellte Erweiterung wurde das XEC-Toolkit angepasst, um entsprechende Spezifikationen syntaktisch prüfen und automatisch implementieren zu können. Dazu wurde zunächst der Compiler-Frontend PET geeignet erweitert, um die oben beschriebenen Pointer als Interaktionsparameter in Estelle-Spezifikationen zu akzeptieren. Zur Aktivierung der Erweiterung muss die Kommandozeilenoption[54] „`-xptria`“ angegeben werden. Da der Implementierungsgenerator XEC gegenwärtig nur Implementierungen erzeugt, die aus einem Prozess bestehen, und somit auch alle Modulinstanzen im selben Speicher-Adressraum arbeiten, sind bei der Übertragung von als Referenz angegebenen Interaktionsparametern keine physischen Kopieroperationen auf den referenzierten Daten notwendig, sodass in der Implementierung lediglich die übergebenen Referenzen (d. h. die Pointer-Werte) von der sendenden Modulinstanz an die empfangende Modulinstanz übertragen werden müssen. Da dies der standardmäßigen Behandlung von Interaktionsparametern entspricht, war für die Implementierung der vorgestellten Erweiterung keine Modifikation von XEC oder der Laufzeitbibliothek XECRT erforderlich.

54. Abkürzung für „E**x**tension: **P**oin**te**r**s** as **I**nter**a**ction Parameters“ (siehe Anhang A.1)

6.4.5.4 Praktische Anwendung in Protokollspezifikationen

Die praktische Anwendung der vorgestellten Erweiterung zur Spezifikation eines effizient implementierbaren Datentransports in einer Protokollspezifikation erfordert zunächst einige kurze Vorüberlegungen zum *PDU-Format* der verschiedenen Dienstschichten. Um die Ableitung einer effizienten Implementierung auf Spezifikationsebene zu unterstützen, sollte das zu übertragende Nutzdatenobjekt so weit wie möglich per Referenz als Interaktionsparameter übertragen und weiterverarbeitet werden.

Leider ist eine Darstellung der SDU-PDU-Verschachtelung in der bisher vorgestellten Form als *schrittweise ineinandergeschachtelte Typstruktur* hier jedoch nicht sinnvoll, da die von der höheren Dienstschicht per Referenz übergebenen Nutzdaten nicht einfach als Referenz in die (*nicht pointer-containing*) PDU eingebettet werden können[55].

Alternativ dazu existieren jedoch zwei Lösungsansätze, die bezüglich der auf den Nutzdaten erforderlichen Kopieroperationen optimal sind, indem sie bei Transporten (zumindest innerhalb des selben Adressraums) gänzlich ohne Kopieroperationen auf den Nutzdaten auskommen. Der naheliegendste Ansatz dieser Art besteht darin, das Nutzdatum von Ende zu Ende als *eigenständigen Interaktionsparameter* per Referenz zu übertragen und unabhängig davon die im Verlauf des Framings und Unframings bearbeiteten *Paketheader und -Trailer in separaten Interaktionsparametern* zu übertragen.

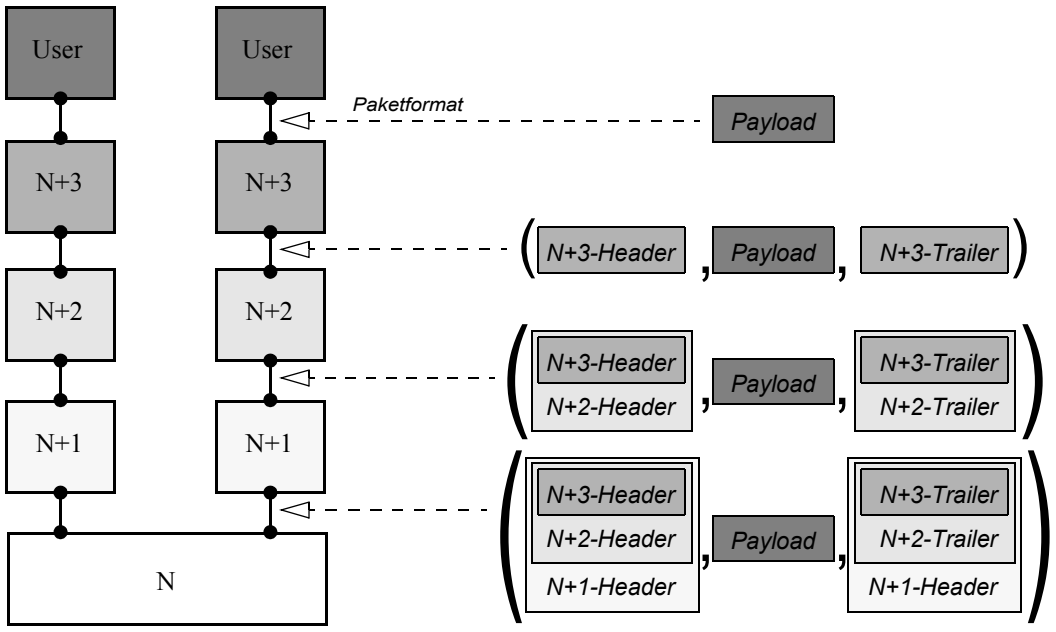


Abbildung 6-20: Parametertripel zur PDU-Darstellung mit separierter Nutzlast

Dazu werden als Interaktionsparameter an den unteren Dienstschnittstellen der Protokollmaschinen statt nur einer monolithischen PDU ein *Parameter-Tripel*[56] (siehe Abb. 6-20) übertragen, das vor und nach der ursprünglich vom obersten Dienstnutzer angelegten Nutzlast („Payload“) die ineinandergeschachtelten Paketheader und -Trailer der darüber liegenden Schichten enthält. Durch diese Maßnahme kann in einfachen[57] Datentransport- bzw. Framing- und Unfra-

55. Die vorgestellte Erweiterung erlaubt die Übertragung per expliziter Referenz (d.h. als Pointer) nur auf Ebene der unmittelbaren Interaktionsparameter, die referenzierten Typen dürfen also nicht „pointer-containing“ sein.

ming-Szenarien die Nutzlast als separater Interaktionsparameter während der gesamten Ende-zu-Ende-Übertragung unverändert weitergereicht werden. Dies ermöglicht es, mit Hilfe der oben eingeführten Estelle-Erweiterung diesen Transport der eigentlichen Nutzlast durch Weitergabe einer Referenz auf ein dynamisch alloziertes Datenobjekt ohne physische Kopieroperationen abzuwickeln. Lediglich die bei den hier betrachteten großen Nutzlasten verhältnismäßig kleinen Paketheader und -Trailer müssen beim Datentransport noch mit den bereits früher betrachteten Framing- und Unframing-Mechanismen bearbeitet und ggf. physisch kopiert werden.

Der Nachteil dieses Verfahrens ist eine gewisse Unhandlichkeit bei der Darstellung der SDU als Parameter-Tripel. Um dies zu vermeiden kann man die vorgestellte Erweiterung auch mit dem in Abschnitt 6.3.2 vorgestellten *Application Layer Framing* kombinieren, indem man bereits auf Applikationsebene einen Paket-Puffer alloziert, der alle Paketformate der darunter liegenden Schichten aufnehmen kann. Dieser Puffer wird dann jeweils zwischen den Protokollkomponenten per Referenz ausgetauscht, und auf jeder Schicht werden nur die jeweils protokollrelevanten Teile des Paketpuffers belegt bzw. interpretiert (siehe Abb. 6-21).

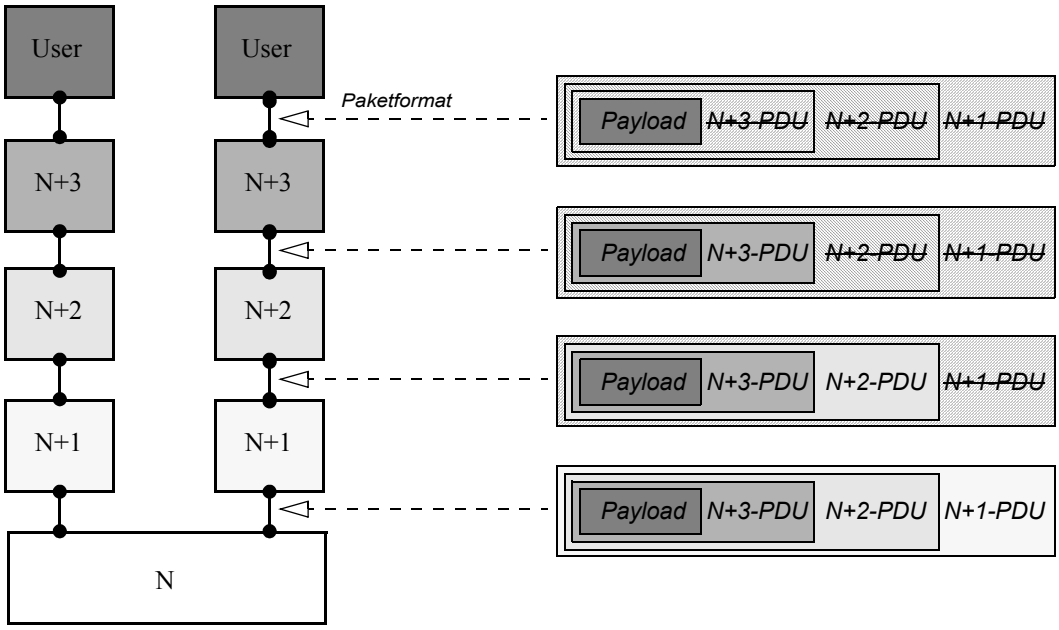


Abbildung 6-21: ALF-orientierte Datenübertragung zur Datenübergabe per Referenz

Dieses Verfahren optimiert in Ergänzung zu der vorherigen Variante auch die Handhabung der Paketheader und -Trailer und ist somit zunächst als eine bzgl. der effizienten Implementierung der Datenübertragung optimale Lösung[58] anzusehen. Zudem ist das Ergebnis des Framings auch hier ein zusammenhängendes Datenobjekt, so dass bei der Implementierung eine spezifi-

56. Diese Darstellung als *Tripel* besitzt (bereits auf Spezifikationsebene) eine gewisse Verwandtschaft zu der in Abschnitt 6.3.3 vorgestellten Implementierungsmethode „*Scatter-Gather-Vektor*“, wobei die Vektorlänge auf 3 beschränkt ist und nur der mittlere Parameter per Referenz übertragen wird. Zur Nachbildung von längeren Vektoren wäre auch eine Verallgemeinerung auf *n-Tupel* mit $n > 3$ denkbar.

57. Paket-Fragmentierungen und Defragmentierungen bleiben hier unberücksichtigt.

58. Optimal, sofern man von einer Parameter-Weitergabe auf Implementierungsebene per Referenz ausgeht; eine kopierende Parameter-Weitergabe bedingt auf allen Ebenen die Kopie des gesamten ALF-Puffers, wodurch sich u. U. die Menge der zu kopierenden Daten vergrößert. Zudem sind explizite Kopieroperationen (s. u.) hier nicht berücksichtigt.

kationsnahe Abbildung auf eine ebenfalls zusammenhängende Repräsentation des binären Paketformats unterstützt wird. Dies vereinfacht die Anbindung an einen realen Basisdienst erheblich (siehe Abschnitt 6.3.2). Nachteilig ist auf Spezifikationsebene jedoch die durch das ALF verursachte enge Kopplung der höheren Dienstnutzer an die PDU-Formate der unteren Diensterbringer, die wir ebenfalls bereits in Abschnitt 6.3.2 kritisch betrachtet haben.

Ein alternativer Ansatz basiert auf einer Verallgemeinerung der hier vorgestellten Estelle-Erweiterung, indem Pointer nicht nur als *unmittelbare Interaktionsparameter*, sondern allgemein *auch innerhalb* der als Interaktionsparameter übertragenen Datentypen vorkommen können, Interaktionsparameter also beliebig *pointer-containing* sein dürfen. Zusammen mit einer entsprechenden syntaktischen Ergänzung, die *alle* in einem aktuellen Interaktionsparameter vorkommenden Zeiger in der o.a. Weise durch Freigabe des dynamisch allozierten Speicherobjekts beim Versender und erneute Allokation und Initialisierung beim Empfänger überträgt, könnte so die Verschachtelung der PDUs durch physische Einbettung der jeweiligen Referenzen ohne Datenkopieroperationen realisiert werden. Dieser Ansatz ist jedoch sowohl semantisch[59] als auch auf Spezifikationsebene bedeutend unübersichtlicher und erfordert im Falle einer Implementierung durch eine Kopieroperation auch eine vollständige Behandlung der zu übertragenden Datenstrukturen. Wir verfolgen ihn an dieser Stelle nicht weiter.

6.4.5.5 Effizienzbewertung

Bei der praktischen Anwendung der ursprünglichen Erweiterung werden unabhängig von den genannten Spezifikationsansätzen einige Vor- und Nachteile deutlich. Ein *Nachteil* ergibt sich aus der Tatsache, dass auf ein als Interaktionsparameter per Referenz weitergegebenes Datenobjekt seitens des Senders kein Zugriff mehr besteht. Daher kann die schon früher diskutierte Zwischenspeicherung von Paketen zum Zwecke der Paketwiederholung nur durch eine explizite Kopieroperation erfolgen. Wichtige *Vorteile* der Erweiterung bestehen jedoch in der direkten Zugänglichkeit der Sende- und Empfangspuffer für den Endnutzer: So kann die Erzeugung der zu übertragenden Nutzdateninhalte direkt in dem (vom Benutzer allozierten) Puffer erfolgen, in dem auch der weitere Transport erfolgt, ohne dass eine zusätzliche Synchronisation beim Zugriff erforderlich wäre. Analog kann der Empfänger direkt auf die Inhalte des per Referenz empfangenen Datenpuffers zugreifen und den Puffer nach Belieben weiterverwenden oder freigeben.

In einer geschlossenen Implementierung eines Protokollstacks innerhalb des selben Adressraums kann somit der Kopieraufwand für den gesamten Ende-zu-Ende-Transport (ohne Paketwiederholungen) auf die Erzeugung und Auswertung des Nutzdatenpuffers durch den Dienstnutzer beschränkt werden. Dies wird von einer Analyse der Kopieroperationen anhand einer entsprechend angepassten Version des in Abschnitt 6.2.4 vorgestellten Datenübertragungsbenchmarks (siehe Anhang B.2) bestätigt: Es treten bei einem Ende-zu-Ende-Transport nur genau 3 (physische) Kopieroperationen der Nutzlast auf:

(i) Kopie des zu versendenden Datums in den dynamisch allozierten Nutzlast-Puffer in Transition „`sending`“ von Modul „`USR`“ (Zeile 88 von Anhang B.2),

59. Semantisch müssen alle derart per Referenz eingebundenen Komponenten eines Interaktionsparameters zu einem (in einer Interaktions-Queue ablegbaren) Wert verbunden werden. Dabei sind zudem insbesondere Probleme im Zusammenhang mit Pointern in varianten Records, Mehrfachreferenzierungen des selben Datenobjekts im selben Interaktionsparameter und Rekursionen in der aufgespannten Datenstruktur zu berücksichtigen.

(ii) Kopie der Nutzlast zum Zwecke einer möglichen Paketwiederholung in Transition „`frame`“ von Modul „`PM_2`“ (Zeile 175 von Anhang B.2)[60] und

(iii) Kopie des empfangenen Datums in lokalen Nutzlast-Puffer in Transition „`receiving`“ von Modul „`USR`“ (Zeile 102 von Anhang B.2).[61]

Insbesondere ist die Anzahl der für eine komplette Ende-zu-Ende-Übertragung eines Nutzdatums erforderlichen physischen Kopieroperationen *vom Parameter H unabhängig*.

Die Auswertung dieses Benchmarks auf Basis von XEC unter Variation der Größe des zu übertragenden Nutzdatentyps (siehe auch Abschnitt 6.2.4) erforderte in der Testumgebung[62] die in Tabelle 6.27 angegebenen Zeiten für eine Ende-zu-Ende-Übertragung.

Tabelle 6.27: Ergebnisse des Datenübertragungsbenchmarks für die explizite Referenzübergabe

Größe Nutzdatentyp [Byte]	**Übertragungsdauer**			**Speedup bzgl. Referenzimpl.**[a]
	absolut [ms] [b]	**absolut normiert**	**relativ** [µs/Byte]	
10	0,0590 ±0,001	= 1,0	5,90	1,0
100	0,0604 ±0,001	1,0	0,60	1,2
1´000	0,0735 ±0,001	1,2	0,074	2,4
10´000	0,1766 ±0,001	3,0	0,018	8,7
100´000	1,526 ±0,018	25,9	0,015	10,1
1´000´000	16,43 ±0,013	278	0,016	9,3

a. im Vergleich mit der Auswertung des Referenz-Benchmarks in Abschnitt 6.2.4
b. Die angegeben Genauigkeiten beziehen sich auf die Zeitmessung und die statistische Streuung der gemessenen Ergebnisse (siehe Abschnitt 2.3).

Offensichtlich überwiegt für kleine Nutzdatentypen (10 bis 1000 Byte) der Zeitanteil für das Protokollmanagement den Datenübertragungsaufwand so weit, dass kein nennenswerter Einfluss auf die (absolute) Ausführungszeit besteht, die absolute Ausführungsdauer also konstant bleibt. Ab etwa 10´000 Byte kehrt sich dieser Zusammenhang um, so dass sich näherungsweise ein linearer Zusammenhang zwischen Größe des Nutzdatentyps und absoluter Übertragungsdauer ergibt. Die relative Übertragungsdauer (d.h. der Quotient der beiden Werte) bleibt nahezu unverändert bei ca. 0,016 µs/Byte.

60. Es ist bei diesem Kopiervorgang zu beachten, dass im Gegensatz zur Ausgangsspezifikation (Zeile 169 von Anhang B.1) das Anfertigen der Kopie *zwingend vor* dem Weitersenden des Paketes erfolgen muss, da nach der Übergabe der Referenz auf das Paket (genauer: auf den ALF-Puffer) als Output-Parameter kein weiterer Zugriff auf den referenzierten Puffer mehr möglich ist (s. o.).

61. Durch die volle Verfügungsgewalt des Users über den per Referenz empfangenen (dynamisch allozierten) Datenpuffer wäre die letzte Operation im Sinne einer dauerhaften Ablage des empfangenen Datums sogar durch eine entsprechende Optimierung des Benchmarks auf diese Spezifikations- und Implementierungsmethode vermeidbar.

62. Plattform: Sun Sparc Ultra-5, 1x 440 MHz CPU, SunOS 5.8, gcc 2.95.2, xec 1.3.0 („-m optimize“)

Vergleicht man nun diese Ergebnisse mit den in Abschnitt 6.2.4 anhand der Referenzimplementierung der Datenübertragung gewonnenen (siehe Tabelle 6.26 auf Seite 265 bzw. Spalte „*Speedup bzgl. Referenzimplementierung*“ in Tabelle 6.27), so zeigt sich, dass bei kleinen Paketen (10 und 100 Byte) erwartungsgemäß kein nennenswerter Performancevorteil durch die Datenübertragung per Referenz gewonnen werden konnte, da hier der Einfluss der Datenkopieroperationen am Gesamtzeitverbrauch nicht signifikant ist. Mit steigender Nutzdatengröße steigt der Gewinn jedoch schnell an, um bei Paketgrößen ab 10´000 Byte bereits eine Größenordnung (Faktor 8,7 bis 10,1) zu erreichen. Dies deckt sich ebenfalls mit den erwarteten Ergebnissen, da durch die Datenübertragung per Referenz statt der 27 Kopieroperationen bei der Referenzimplementierung nun nur noch 3 Kopieroperationen der Nutzdaten für die selbe Transportstrecke erforderlich sind und somit ein Speedup von 9 zu erwarten gewesen wäre.

Zusammenfassend kann man also festhalten, dass durch die Anwendung der vorgestellten expliziten Referenzübergabe ein *erheblicher Leistungsgewinn* bei der Datenübertragung innerhalb von (Teil-) Systemen mit gemeinsamem Speicher (shared memory) erreicht werden kann, ohne auf Spezifikationsebene konkrete Details der Implementierung (wie eben gerade die Kenntnis der Verfügbarkeit gemeinsamen Speichers zwischen verschiedenen Modulinstanzen) berücksichtigen zu müssen.

So hätte die Übertragung eines zwischen zwei Modulinstanzen per expliziter Referenz übertragenen Datums auf Implementierungsebene durchaus auch ohne gemeinsamen Speicher (z. B. mittels physischer Kopieroperationen oder durch Übertragung über einen realen Netzwerk-Basisdienst) realisiert werden können, ohne dass eine Anpassung der Spezifikation oder eine Beeinträchtigung ihrer Semantik erforderlich gewesen wäre.

Wir werden im nächsten Abschnitt eine alternative Implementierungstechnik kennen lernen, die ohne explizite Darstellung der zu übertragenden Daten als referenzierte Objekte und damit auf höherem Abstraktionsniveau eine ähnlich effiziente Implementierung der Datenübertragungsoperationen erlaubt.

6.4.6 Containertyp-Erweiterung zur verdeckten Referenzübergabe

Die vorangegangenen Abschnitte haben gezeigt, dass die explizite Spezifikation der Übertragung von Datenobjekten durch gemeinsame Variablen bzw. die Weitergabe von Referenzen als Interaktionsparameter ein probates Mittel zur Erlangung einer automatisch generierten, effizienten Implementierung des spezifizierten Systems sind. Wir haben auch gesehen, dass gerade der Einsatz gemeinsamer (oder gar globaler) Variablen negativen Einfluss auf die Nebenläufigkeit eines Systems hat oder gar eine problemangemessene Implementierung gerade auf verteilten Systemen behindern kann.

Im letzten Abschnitt haben wir diese Problematik auf Spezifikationsebene dadurch gelöst, dass wir zwar durch eine syntaktische Erweiterung den *Austausch von Referenzen auf Datenobjekte* zwischen verschiedenen Modulinstanzen (als Interaktionsparameter) ermöglichen, gleichzeitig durch eine semantische Anpassung dieser Erweiterung aber den konkurrierenden Zugriff verschiedener Modulinstanzen auf das selbe Datenobjekt ausgeschlossen haben. Dadurch wird effektiv verhindert, dass zwei Modulinstanzen konkurrierend auf das selbe Datenobjekt zugreifen können und so ihre Nebenläufigkeit eingeschränkt wird. Andererseits muss dadurch zur Erzeugung einer lokalen Kopie eines derart per Referenz übertragenen Datenobjekts (z. B. zur Vor-

bereitung einer möglichen Paketwiederholung) immer auch eine explizite Kopieroperation stattfinden, die auf Implementierungsebene in den meisten Fällen auch zu einer physischen Kopieroperation führt.

Bei genauerer Betrachtung ist es jedoch bereits ausreichend, wenn ein von mehreren Modulinstanzen konkurrierend zugreifbares Datenobjekt lediglich vor *Veränderungen* durch diese Modulinstanzen geschützt wird, da ein konkurrierender *lesender* Zugriff auf ein solches Datenobjekt die Nebenläufigkeit zwischen den zugreifenden Modulinstanzen nicht einschränkt. Dadurch können in beliebigem Umfang (virtuelle) Kopien von Datenobjekten erzeugt und auch nach Weitergabe des Datenobjekts über einen längeren Zeitraum gespeichert werden, ohne dass auf Implementierungsebene tatsächlich physische Kopien des Datenobjekts erzeugt werden müssen.

Andererseits sollte der Datenübertragungsmechanismus von den Details möglicher späterer Implementierungen der Datenübertragung durch Weitergabe einer Referenz auf ein gemeinsames Datenobjekt stärker abstrahieren als dies in den bisherigen Ansätzen der Fall war und stattdessen auf Spezifikationsebene die Standard-Estelle zu Grunde liegende Idee der Übertragung von Interaktionsparametern „*by-value*“ wieder in den Vordergrund rücken.

Diese beiden Forderungen nach *Einhaltung der Copy-Semantik* bei der Übergabe von Interaktionsparametern auf Spezifikationsebene und gleichzeitiger effizienter Implementierung dieser Kopieroperationen durch die *Weitergabe von Referenzen* auf ein und das selbe Datenobjekt widersprechen sich offensichtlich und sind, wie wir bereits in Abschnitt 6.3 gesehen haben, im Allgemeinen nicht sinnvoll automatisch auf der Basis von Standard-Estelle zu implementieren.

Eine überraschend einfache Lösung ergibt sich jedoch auf der Basis der *Containertyp-Erweiterung*, die wir in Abschnitt 5.2.2 ursprünglich eingeführt haben, um Datenübertragungsdienste formal (und insbesondere typsicher), zugleich aber auch Nutzdatentyp-unabhängig spezifizieren und anwenden zu können. Der zu Grunde liegende ANY-TYPE ist dabei zuweisungskompatibel zu jedem anderen (nicht pointer-containing) Estelle-Typ, entspricht ansonsten aber – insbesondere bezüglich der Copy-Semantik – den gewohnten Standard-Estelle-Typen. Insofern bietet der ANY-TYPE auf Spezifikationsebene ein hohes Abstraktionsniveau, ohne explizit Zugeständnisse an die effiziente Implementierbarkeit zu machen.

Die interessante Eigenschaft des ANY-TYPEs bezüglich der effizienten Implementierung von Datenübertragungsoperationen ist jedoch, dass es auf ANY-TYPE-Variablen neben der Zuweisung eines neuen Wertes keinerlei modifizierende Operationen gibt. Die ANY-TYPE-Werte können zudem lediglich auf andere ANY-TYPE-Variablen zugewiesen werden oder durch Rückkonvertierung in ihren Ausgangstyp (kopierend) ausgelesen werden.

Für einen ANY-TYPE-Wert ergibt sich damit folgender Lebenszyklus:

(i) Erzeugung durch Zuweisung eines Datums anderen Typs auf eine ANY-TYPE-Variable,

(ii) ggf. Kopie durch Zuweisung eines ANY-TYPE-Wertes auf eine andere ANY-TYPE-Variable,

(iii) ggf. Kopie durch Rückkonvertierung eines ANY-TYPE-Wertes in seinen Ausgangstyp

(iv) Freigabe durch Zuweisung eines anderen Wertes auf die referenzierende ANY-TYPE-Variable bzw. Ende der Gültigkeit[63] der ANY-TYPE-Variablen.

63. u. a. durch Verlassen der enthaltenden Scope-Region, z. B. bei einer transitionslokalen Variable nach dem Feuern der Transition

Da der Wert einer ANY-TYPE-Variablen selbst nur durch die zuletzt genannte Freigabeoperation *(iv)* modifiziert werden kann, ist eine Implementierung der Kopieroperation zwischen ANY-TYPE-Variablen *(ii)* auch durch Weitergabe einer Referenz auf das selbe Datenobjekt (anstatt durch Anfertigung einer physischen Kopie) konform zur früher eingeführten Semantik der Containertyp-Erweiterung möglich. Durch diese Maßnahme entstehen ANY-TYPE-Werte als eigenständige Objekte, die von einer oder mehreren ANY-TYPE-Variablen referenziert werden.

Für die o. g. Operationen auf ANY-TYPE-Werten ergeben sich bei einer Darstellung mit Mehrfachreferenzen auf Implementierungsebene folgende Aktionen:

(i) Die ANY-TYPE-Wert-Objekte werden erzeugt bei der Zuweisung eines Datums anderen Typs auf eine ANY-TYPE-Variable. Dabei wird eine *vollständige Kopie* des Ausgangsdatums angefertigt. Die ANY-TYPE-Variable stellt dabei initial die einzige Referenz auf das neue entstandene ANY-TYPE-*Wert-Objekt* dar.

(ii) Wird eine Kopie eines ANY-TYPE-Wertes durch Zuweisung auf eine andere ANY-TYPE-Variable erzeugt, so wird lediglich in Form dieser ANY-TYPE-Variablen eine zweite Referenz auf das bereits existierende ANY-TYPE-Wert-Objekt erzeugt.

(iii) Bei einer Rückkonvertierung eines ANY-TYPE-Wertes in seinen Ausgangstyp wird wie bereits in (i) eine vollständige Kopie des enthaltenen Datums erzeugt.

(iv) Bei der Freigabe eines ANY-TYPE-Wertes durch Zuweisung eines anderen Wertes auf die referenzierende ANY-TYPE-Variable (siehe *(i)*) oder bei Ende der Gültigkeit der ANY-TYPE-Variablen selbst wird die entsprechende Beziehung zwischen der ANY-TYPE-Variablen und dem referenzierten ANY-TYPE-Wert-Objekt aufgehoben. Falls es danach keine weiteren Referenzen auf das ANY-TYPE-Wert-Objekt gibt, kann dieses Objekt naheliegenderweise ebenfalls freigegeben werden.

Durch diesen Implementierungsmechanismus entstehen Kopien auf den in den ANY-TYPE-Werten enthaltenen Daten nur noch bei der Konvertierung in den ANY-TYPE bzw. bei der Rückkonvertierung vom ANY-TYPE in den Ausgangstyp. Dies ist, wie wir früher bereits gesehen haben, bei der Ende-zu-Ende-Übertragung von Nutzdaten nur an den Endpunkten der Übertragung notwendig. Bei der sonstigen Verarbeitung kann jedoch bei allen auftretenden Kopieroperationen zwischen ANY-TYPE-Variablen[64] in vorteilhafter Weise vollständig auf die Anfertigung physischer Kopien verzichtet werden. Stattdessen werden bei den Kopieroperationen lediglich Mehrfach-Referenzen auf das selbe Objekt erzeugt. Dies ist semantikkonform, da keine Modifikationen an den referenzierten Objekten möglich sind.

Auf diese Weise kann die Anzahl der physischen Kopieroperationen auf zu übertragenden Nutzdaten innerhalb von Teilsystemen, die über gemeinsamen Speicher verfügen, auf genau zwei beschränkt werden. Dies gilt insbesondere für den in Abschnitt 6.2.4 vorgestellten Datenübertragungsbenchmark, bei dem die innerhalb des Protokollstacks angefertigten Kopien von PDUs für mögliche Paketwiederholungen ohne Kopieroperationen auf den enthaltenen Nutzdaten implementiert werden (s. u.).

64. insbesondere auch bei der Anfertigung von (langlebigen) Kopien für eine potentielle Paketwiederholung und beim Framing und Unframing von Paketen (siehe auch Abschnitt 5.3)

6.4.6.1 Implementierung

Zur Realisierung dieser alternativen Implementierungsmethode auf Basis der bereits in Abschnitt 5.4.2 vorgestellten any-type-Implementierungsmethode wurde im Wesentlichen in der Klasse `Abstract_Container` ein Referenzzähler (`uRefCount`) integriert, der bei der Erzeugung neuer Referenzen durch ein `AnyType`-Objekt inkrementiert und bei der Auflösung von Referenzen (durch Destruktion des `AnyType`-Objekts aufgrund des Endes der Gültigkeit der Variablen oder Zuweisung eines anderen Wertes) dekrementiert wird. Erreicht der Zähler dabei den Wert 0, so wird das nunmehr unreferenzierte Container-Objekt gelöscht.

Die strukturellen und programmtechnischen Anpassungen zur Realisierung dieser alternativen Implementierung von any-type-Objekten per Referenz sind verhältnismäßig geringfügig, wie man an dem in Abb. 6-22 dargestellten UML-Diagramm der any-type-Klassenstruktur in der XEC-Laufzeitbibliothek bereits erkennen kann (die modifizierten Teile sind unterstrichen dargestellt, vgl. auch Abb. 5-22 auf Seite 234).[65]

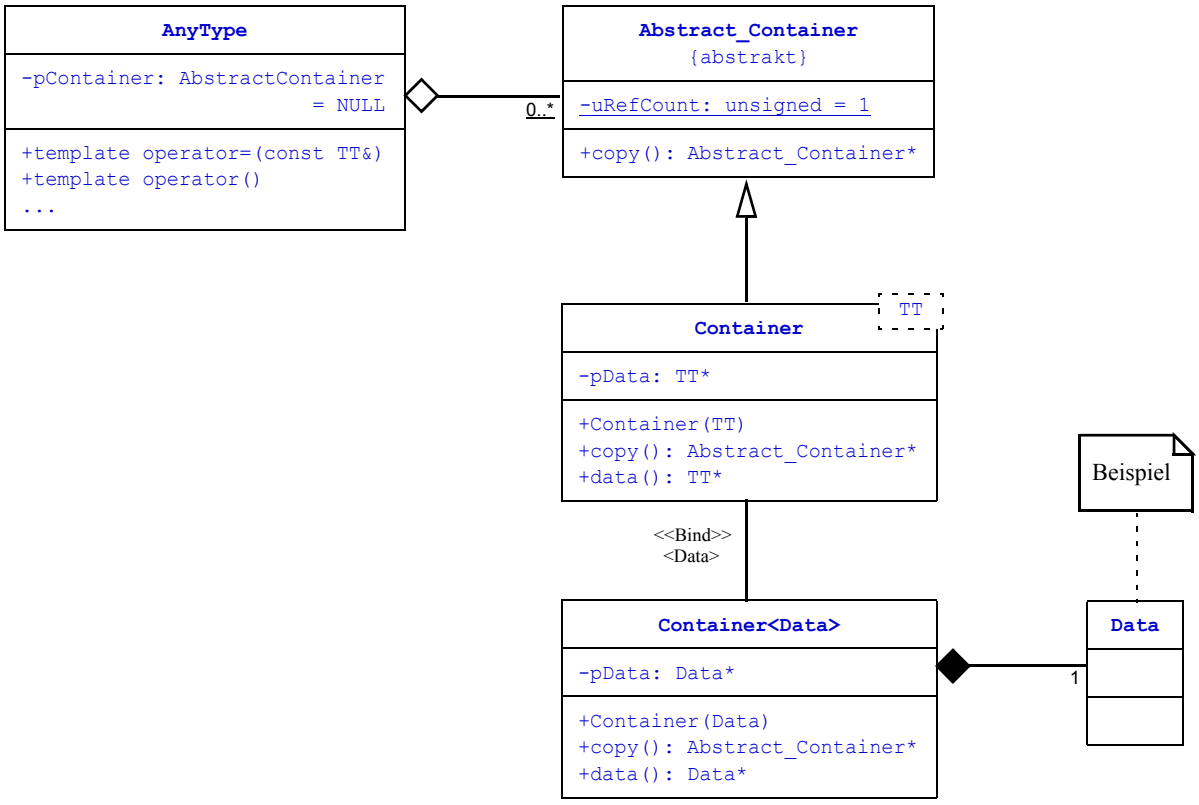


Abbildung 6-22: UML-Diagramm der XECRT-Klasse „`AnyType`“ mit Mehrfachreferenzen

Neben der effizienten Implementierbarkeit der Datenübertragungsoperationen bietet der Einsatz der Containertyp-Erweiterung zudem den wesentlichen Vorteil, dass im Gegensatz zu den früher diskutierten Maßnahmen zur Unterstützung einer effizienten Implementierung auf Spezifikationsebene hier das Abstraktionsniveau der Spezifikation nicht *verringert* sondern sogar *erhöht* wird, was ja die ursprüngliche Motivation zur Einführung der Containertyp-Erweiterung

65. Die im Zusammenhang mit der alternativen Implementierung der Containertyp-Erweiterung eingeführten Modifikationen der Quellen der Laufzeitbibliothek werden im C++-Quelltext durch das Präprozessor-Symbol `ANYTYPE_EXT_BYREF` aktiviert und sind somit leicht als solche zu identifizieren.

war (siehe Kapitel 5). So könnte durch die mit der Containertyp-Erweiterung ermöglichte Vereinheitlichung der Dienstschnittstellen[66] der einzelnen Protokollmaschinen nicht nur die Definition von „PM_1“ und „PM_3“ unifiziert werden, sondern auch die vertikale Struktur des Benchmarks, die bisher fest auf 3 Protokollmaschinen festgelegt war, vollständig dynamisch zur Laufzeit auf eine beliebige Anzahl von Protokollhierarchien konfiguriert werden. Somit verbindet diese Erweiterungen in vorteilhafter Weise Abstraktion auf Spezifikationsebene und Effizienz auf Implementierungsebene.

Wie leicht insbesondere in typischen Kommunikationssystem-Spezifikationen die Containertyp-Erweiterung auch *nachträglich* erst zur Gewinnung einer effizienten Implementierung eingesetzt werden kann, demonstrieren wir nun anhand des bereits genannten Datenübertragungsbenchmarks. Im Gegensatz zu den früher diskutierten Ansätzen, die zur Umstellung einer Standard-Estelle-Spezifikation auf die Nutzung gemeinsamer Variablen (siehe Abschnitt 6.4.3 und 6.4.4) oder expliziter Datenübertragungen per Referenz (siehe Abschnitt 6.4.5) umfangreiche Modifikationen der Spezifikation erforderlich machten, genügt zur Umstellung des Datenübertragungsbenchmarks (siehe Anhang B.1) die Anpassung der Typdefinition des Interaktionsparameters in der Kanaldefinition zwischen Dienstnutzer („user“) und oberstem Diensterbringer („PM_3“). Dieser Typ entspricht nicht länger dem vom Dienstnutzer intern genutzten Nutzdatentyp „T_UserData“, sondern wird nun als „ANY TYPE“ definiert:

Beispiel 6.61: Umstellung des Datenübertragungsbenchmarks aus Anhang B.1 auf „ANY-TYPE“

Die modifizierten Teile sind unterstrichen dargestellt.

```
TYPE SDU_T_3 = ANY TYPE;                    { ehemals: T_UserData; }
TYPE PDU_T_3 = RECORD header: INTEGER;
                      payload: SDU_T_3;
                      trailer: INTEGER
               END;
{ ... }

BODY body_usr FOR hdr_usr;
  VAR data_buf: T_UserData;
  { ... }

  TRANS
    { ... }
    NAME sending:
    BEGIN
      OUTPUT ipdown.msg(data_buf);   { jetzt implizite Konvertierung }
      { ... }
    END;


  TRANS
    WHEN ipdown.msg{data: SDU_T_3}
    NAME receiving:
    BEGIN
      data_buf := data;              { jetzt implizite Konvertierung }
```

66. Die Dienste der verschiedenen Schichten und damit die externen Schnittstellen der Protokollmodule unterscheiden sich nur aufgrund der verschiedenen (verschachtelten) PDU-Formate.

```
            { ... }
        END;

    END; {BODY}
```

(Ende von Beispiel 6.61)

Dadurch wird beim Verschicken eines Nutzdaten-Pakets durch den Dienstnutzer (Transition „sending“) bei der Parameter-Übergabe des Nutzdatentyps (aktueller Parameter vom Typ „T_UserData“) an die Output-Anweisung (formaler Parameter vom Typ „ANY TYPE“) die implizite Konvertierung in den ANY-TYPE durchgeführt. Von nun an erfolgt der gesamte weitere Transport durch das Kommunikationssystem bis zum Empfang beim Partner-Dienstnutzer abstrahiert als ANY-TYPE und damit ohne weitere physische Kopieroperationen auf Implementierungsebene. Erst beim empfangenden Dienstnutzer (Transition „receiving“) muss zur Gewinnung des Inhalts der übertragenen Daten die Rückkonvertierung des ANY-TYPEs in den ursprünglichen Nutzdatentyp („T_UserData“) durch eine Kopieroperation erfolgen. Dies geschieht bereits in der ursprünglichen Benchmarkspezifikation durch die Zuweisung des entsprechenden Interaktionsparameters an eine modullokale Variable („data_buf“) vom Nutzdatentyp.

Dem Vorteil, dass im Verlauf der Übertragung Kopieroperationen auf dem ANY-TYPE auf Implementierungsebene ohne physische Kopieroperationen realisiert werden können, steht jedoch der Nachteil gegenüber, dass im Gegensatz beispielsweise zur Übertragung von Nutzdaten per expliziter Referenz (siehe Abschnitt 6.4.5) die beiden *Kopieroperationen bei der Konvertierung* vom Nutzdatentyp in den ANY-TYPE (beim erstmaligen Senden des Nutzdatums) bzw. in Gegenrichtung (beim finalen Empfang des Nutzdatums) aus den bekannten Gründen nicht vermeidbar sind.[67]

In einer geschlossenen Implementierung eines Protokollstacks innerhalb des selben Adressraums kann somit der Kopieraufwand für den gesamten Ende-zu-Ende-Transport – selbst mit einer beliebigen Zahl von Paketwiederholungen – auf die Erzeugung und Auswertung des Nutzdatenpuffers durch die Dienstnutzer beschränkt werden. Dies wird von einer Analyse der Kopieroperationen anhand einer entsprechend angepassten Version des in Abschnitt 6.2.4 vorgestellten Datenübertragungsbenchmarks (siehe Beispiel 6.61 bzw. Anhang B.3) bestätigt: Es treten bei einem Ende-zu-Ende-Transport nur genau 2 (physische) Kopieroperationen der Nutzlast auf:

(i) Kopie des zu versendenden Datums bei der Konvertierung in den ANY-TYPE (bei der Parameterübergabe in der OUTPUT-Anweisung) in Transition „sending“ von Modul „USR“ (Zeile 83 von Anhang B.3) und

(ii) Kopie des empfangenen Datums in lokalen Nutzlast-Puffer in Transition „receiving“ von Modul „USR“ (Zeile 94 von Anhang B.3).

Insbesondere ist die Anzahl der für eine komplette Ende-zu-Ende-Übertragung eines Nutzdatums erforderlichen physischen Kopieroperationen vom Parameter *H unabhängig*.

67. Beim Empfang eines Nutzdatums per *expliziter Referenzübergabe* (siehe Abschnitt 6.4.5) konnte u. U. der Inhalt des referenzierten Objekts direkt ausgewertet werden, ohne eine weitere explizite Kopieroperation beim Empfänger (z. B. in eine lokale Variable) zu erzwingen.

6.4.6.2 Effizienzbewertung

Die Auswertung dieses Benchmarks auf Basis von XEC unter Variation der Größe des zu übertragenden Nutzdatentyps (analog zu und im Vergleich mit der Auswertung des Referenz-Benchmarks in Abschnitt 6.2.4) erforderte in der Testumgebung[68] die in Tabelle 6.28 angegebenen Zeiten für eine Ende-zu-Ende-Übertragung.

Tabelle 6.28: Ergebnisse des Datenübertragungsbenchmarks für die Containertyp-Erweiterung

Größe Nutzdatentyp [Byte]	**Übertragungsdauer**			**Speedup bzgl. Referenzimpl.**[a]
	absolut [ms] [b]	**absolut normiert**	**relativ** [µs/Byte]	
10	0,0765 ±0,001	= 1,0	7,65	0,78
100	0,0733 ±0,001	1,0	0,73	0,96
1´000	0,0798 ±0,001	1,0	0,080	2,2
10´000	0,1802 ±0,003	2,4	0,018	8,6
100´000	1,312 ±0,023	17,2	0,013	11,7
1´000´000	13,68 ±0,032	179	0,014	11,2

a. im Vergleich mit der Auswertung des Referenz-Benchmarks in Abschnitt 6.2.4
b. Die angegeben Genauigkeiten beziehen sich auf die Zeitmessung und die statistische Streuung der gemessenen Ergebnisse (siehe Abschnitt 2.3).

Offensichtlich überwiegt für kleine Nutzdatentypen (10 bis 1000 Byte) auch hier wiederum der Zeitanteil für das Protokollmanagement den Datenübertragungsaufwand so weit, dass kein nennenswerter Einfluss auf die (absolute) Ausführungszeit besteht, die absolute Ausführungsdauer also praktisch konstant bleibt. Ab etwa 10´000 Byte kehrt sich dieser Zusammenhang einmal mehr um, so dass sich näherungsweise ein linearer Zusammenhang zwischen Größe des Nutzdatentyps und absoluter Übertragungsdauer ergibt, die relative Übertragungsdauer (d.h. der Quotient der beiden Werte) nahezu unverändert bei ca. 0,015 µs/Byte bleibt.

Vergleicht man nun diese Ergebnisse mit den in Abschnitt 6.2.4 anhand der Referenzimplementierung der Datenübertragung gewonnenen (siehe Tabelle 6.26 auf Seite 265 bzw. Spalte „Speedup bzgl. Referenzimplementierung“ in Tabelle 6.28), so zeigt sich, dass bei kleinen Paketen (10 und 100 Byte) ein Performancenachteil von ca. 22% (Speedup 0,78) gegenüber der Implementierung der semantischen Kopieroperationen durch physische Kopieroperationen auftritt. Diese Performanceeinbuße ergibt sich aus dem höheren Verwaltungsoverhead bei der internen Verwaltung der ANY-TYPE-Datenobjekte. So werden in der zu Grunde liegenden Benchmarkspezifikation alle Kopieroperationen (und damit insbesondere alle Framing- und Unframing-Operationen) durch die Erzeugung von Mehrfachreferenzen auf das als any-type-Objekt abstrahierte Nutzdatum realisiert. Der damit verbundene Managementaufwand ist augenscheinlich bei kleinen Paketen größer als eine „brute-force“-Kopieroperation des Pakets, wie sie bei der Referenzimplementierung zum Einsatz kam.

68. Plattform: Sun Sparc Ultra-5, 1x 440 MHz CPU, SunOS 5.8, gcc 2.95.2, xec 1.3.0 („-m optimize“)

Mit steigender Nutzdatengröße kehrt sich das Verhältnis zwischen Vor- und Nachteilen der Implementierung der Datenübertragung auf Basis der Containertyp-Erweiterung jedoch schnell um: Bei 1´000 Bytes Paketgröße wird bereits ein Speedup von über 2 und bei Paketgrößen ab 10´000 Byte dann eine ganze Größenordnung (ca. Faktor 8,6 bis 11,7) erreicht. Dies deckt sich ebenfalls weitgehend mit den erwarteten Ergebnissen, da durch die Datenübertragung per Referenz statt 27 Kopieroperationen (bei der Referenzimplementierung) nur noch 2 Kopieroperationen der Nutzdaten für die selbe Transportstrecke erforderlich sind und somit ein Speedup von 13,5 zu erwarten gewesen wäre.[69]

Im Vergleich zu dem in Abschnitt 6.4.5 ermittelten Gewinn durch die Kombination von expliziter Referenzübergabe und ALF (siehe Tabelle 6.27 auf Seite 300) zeigen sich zwar bei kleinen Paketgrößen Nachteile (-22% bei 10 Bytes) durch den erhöhten Verwaltungsaufwand, aufgrund der Einsparung von physischen Kopieroperationen bei der Anfertigung lokaler Kopien für mögliche Paketwiederholungen übersteigt der erzielte Speedup bei großen Paketen jedoch den der expliziten Referenzübergabe (+20% bei 1´000´000 Bytes).[70]

Zusammenfassend kann man festhalten, dass durch die Anwendung der vorgestellten Implementierungsmethode für die Containertyp-Erweiterung im Vergleich zur Referenzimplementierung ein *erheblicher Leistungsgewinn* bei der Datenübertragung innerhalb von (Teil-) Systemen mit gemeinsamem Speicher (shared memory) erreicht werden kann. Er ist insgesamt vergleichbar zu dem der Kombination von expliziter Referenzübergabe und ALF (siehe Abschnitt 6.4.5).

Auf konzeptioneller Ebene bietet die Containertyp-Erweiterung zusammen mit der hier vorgestellten Implementierungsmethode jedoch ein wesentlich günstigeres Verhältnis zwischen Abstraktion und Implementierungsperformance, da der Einsatz der Containertyp-Erweiterung im Gegensatz zur expliziten Referenzübergabe nicht nur aus implementierungstechnischen Erwägungen heraus vorteilhaft ist, sondern auch das Abstraktionsniveau und die Wiederverwendbarkeit der Spezifikation erhöht.

So verbindet die Containertyp-Erweiterung in vorteilhafter Weise die Anforderungen der formalen Spezifikation und der automatischen Generierung effizienter Implementierungen.

69. Der erwartete Speedup wird bei den angegebenen Messergebnissen um ca. 13% verfehlt. Die Ursache für die Abweichung bleibt teilweise unklar, da bei großen Paketen der Overhead für die Verwaltung der any-type-Mehrfachreferenzen nicht mehr signifikant in Erscheinung treten sollte. Vermutlich ist die Abweichung überwiegend auf einen der in Abschnitt 2.3 dargestellten Randeffekte bei Performanceevaluationen auf realen Systemen zurückzuführen.

70. Würden in dem Benchmark tatsächlich Paketwiederholungen stattfinden, so würde der Gewinn noch deutlicher ausfallen, da jetzt hierzu keine physischen Kopieroperationen mehr nötig sind.

6.5. Übertragbarkeit auf andere FDTs

Zuletzt wollen wir uns jetzt noch der Frage zuwenden, inwieweit die gewonnenen Ergebnisse auf andere formale Beschreibungstechniken übertragbar sind. Grundsätzlich gelten die gewonnenen Ergebnisse in analoger Weise für alle formalen Beschreibungstechniken, die die Spezifikation konkurrierend zugreifbarer gemeinsamer Variablen zwischen nebenläufig agierenden Komponenten verbieten.

Als Beispiel betrachten wir wiederum SDL [ITU94]. In SDL können gemeinsame Variablen angelegt werden, die von verschiedenen (konkurrierend agierenden) Prozessen lesend wie auch schreibend zugegriffen werden können. Diese wären prinzipiell zur Ablage von zu übertragenden Nutzdaten geeignet. Um jedoch die bereits oben diskutierte Koordinations-Problematik beim konkurrierenden Zugriff zu lösen, ist der Zugriff auf solche Variablen als syntaktisch zu expandierendes Makro definiert, bei dem durch den Austausch von asynchronen Nachrichten Zugriff auf die Variablen und ihre Inhalte genommen wird. Die Parameterübertragung bei diesen Nachrichten erfolgt wiederum im Sinne der Copy-Semantik. Entsprechend ist der Zugriff auf solche gemeinsamen Variablen aus Implementierungsgesichtspunkten kein probates Mittel zur Vermeidung von semantischen oder physischen Kopieroperationen. Somit ist die dem Abschnitt 6 zu Grunde liegende Problemstellung auch in SDL gegeben.

Die Übertragung der in Abschnitt 6.4.5 eingeführten „*expliziten Referenzübergabe*“ scheitert in SDL am Fehlen von Referenzen auf dynamisch allozierte Speicherobjekte. Dagegen scheint die in Abschnitt 6.4.6 eingeführte Implementierung der *Containertyp-Erweiterung* und die damit verbundene Möglichkeit zur effizienten Implementierung von Datenübertragungsoperationen durchaus möglich und wirksam.

Da SDL das Konzept der *abstrakten Datentypen* (ADTs) beinhaltet, wäre an dieser Stelle möglicherweise auch eine effiziente Implementierung von Daten-Kopieroperationen auf Basis der durch die ADTs darstellbaren Abstraktionen denkbar. Inwieweit dies Grundlage einer effizienten und insbesondere automatischen Implementierung werden kann, bleibt zu untersuchen.

6.6. Zusammenfassung

Wir haben uns mit der Frage beschäftigt, wie auf der Basis formaler Protokollbeschreibungen automatisch Implementierungen erzeugt werden können, die auf effiziente Art und Weise die *Übertragung von Nutzdaten* innerhalb von Kommunikationssystemen realisieren. Dieser Untersuchung lag die Beobachtung zu Grunde, dass in formal beschriebenen Protokollen bei der Übertragung von Nutzdaten eine erhebliche Zahl von *(semantischen) Kopieroperationen* auf diesen Nutzdaten auftritt. Überwiegt bei der Übertragung kleiner Nutzdaten-Pakete noch der zeitliche Aufwand zur Abwicklung des Kontrollflusses (siehe Kapitel 4), so kann bei mittleren und großen Paketen schnell der Zeitaufwand für die Handhabung der Nutzdaten um Größenordnungen oberhalb des Zeitaufwandes für den Kontrollfluss liegen.

Die gesamte Problemstellung der effizienten Datenübertragung wurde anhand zweier miteinander verbundener Grundfragen untersucht:

- wie können semantische Kopieroperationen *effizient implementiert* werden und
- wie können *semantische Kopieroperationen* bereits auf Spezifikationsebene *vermieden* werden.

Dazu haben wir zunächst anhand typischer Protokollmechanismen untersucht, welche *semantischen Daten-Kopieroperationen* überhaupt auf Spezifikationsebene anzutreffen sind. Neben den semantischen Kopieroperationen zum Zwecke der Übertragung von Nutzdaten-Paketen als *Interaktionsparameter* haben sich als typische Protokolloperationen die auf fast allen Protokollschichten anzutreffenden *Framing- und Unframing-Operationen* auf Nutzdaten-Paketen und das explizite lokale Kopieren von Nutzdaten-Paketen zum Zwecke einer *potenziellen Paketwiederholung* als wesentliche Ursache für semantische Kopieroperationen herausgestellt.

Ein wesentliches Problem formaler Beschreibungstechniken an dieser Stelle ist, dass zur Trennung der Zustandsräume verschiedener (nebenläufiger) Modulinstanzen die asynchrone Übertragung von Nutzdaten von einer Modulinstanz zu einer anderen nur durch eine semantische Kopieroperation erlaubt ist.[71] Dieser Punkt ist für die Möglichkeiten einer automatischen effizienten Implementierung von Datenübertragungsoperationen ausschlaggebend.

Als Referenz für spätere Optimierungsüberlegungen haben wir eine *Referenz-Implementierungsmethode* für semantische Kopieroperationen angegeben, wie sie sowohl in XEC, als auch in allen anderen bekannten automatischen Implementierungsgeneratoren prinzipiell angewendet wird. Anschließend haben wir anhand der vorangegangenen Untersuchungen einen künstlichen *Datenübertragungsbenchmark* entwickelt.

Als Grundlage für mögliche Optimierungen der Datenübertragung von automatisch generierten Implementierungen haben wir dann zunächst die in manuell erstellten Protokollimplementierungen vorzufindenden Optimierungstechniken bei der Datenübertragung untersucht. Viel versprechende Ansätze waren dabei das *Initial Placement* und die sich daraus ergebenden Verfeinerungen *Application Layer Framing* bzw. *Scatter-Gather-Vektoren*. Weiterhin haben wir auch Ansätze auf der Basis virtueller Speicherverwaltungen (*Copy-on-Write*-Techniken) und die damit verbundenen Probleme untersucht.

---

71. In Estelle gibt es in Modulinstanzen und Kindmodulinstanzen zwar gemeinsame Variablen, der Nutzen für eine effiziente Datenübertragung in Protokollhierarchien ist jedoch gering (siehe Abschnitt 6.4.3).

Bei der Untersuchung von Optimierungsansätzen für automatisch generierte Implementierungen (siehe Abschnitt 6.4) haben wir uns dann mit den Problemen bei der Übertragung von manuellen Optimierungstechniken auf die automatische Implementierung beschäftigt. Als wesentliches Problem hat sich dabei die *Einhaltung der Copy-Semantik* der verschiedenen semantischen Kopieroperationen gezeigt. Auch *pragmatische Ansätze*, die eine Datenübertragung per Referenz dadurch realisieren, dass sie den Implementierungsgenerator darüber „täuschen", dass eine Datenreferenz als Interaktionsparameter zwischen verschiedenen Modulen ausgetauscht wird, sind aus konzeptioneller Sicht nicht befriedigend.

Als Lösung der Problematik haben wir zwei völlig neue Ansätze vorgestellt, die durch syntaktische und semantische Erweiterungen der formalen Technik Estelle konzeptionell „*sauberere*" Möglichkeiten bieten, die Datenübertragung in Protokoll-Spezifikationen automatisch effizient zu implementieren. Der erste Ansatz ist die Estelle-Erweiterung „*explizite Referenzübergabe*", eine geradezu subtile syntaktische und semantische Erweiterung, die eine sehr effiziente Übertragung von Nutzdatenpaketen durch Modulhierarchien erlaubt.

Eine sehr viel abstraktere Lösung ergab sich aus einer *alternativen Implementierung* für die bereits in Kapitel 5 eingeführte *Containertyp-Erweiterung*. Diese nutzt das Fehlen von modifizierenden Operationen auf any-type-Werten zur Copy-Semantik-konformen Darstellung der Nutzdaten als mehrfach referenzierte Objekte.

Obwohl die Containertyp-Erweiterung ursprünglich ausschließlich zur Steigerung des Abstraktionsniveaus der formalen Spezifikation entwickelt wurde, liefert sie mit Hilfe dieser Implementierungsmethode auf elegante Weise *äußerst effiziente Implementierungen* der Datenübertragungsproblematik. So war zur Anpassung des ursprünglichen Estelle-Datenübertragungsbenchmarks auf die Containertyp-Erweiterung lediglich eine einzige Änderung des Spezifikationstextes erforderlich. Mit Hilfe der neuen Implementierungsmethode konnte jedoch bei großen Paketen ebenfalls ein Speedup von etwa Faktor 10 im Vergleich zur Referenzimplementierung gewonnen werden. Damit bietet die Containertyp-Erweiterung die günstigste Kombination von *hohem Abstraktionsniveau* auf Spezifikationsebene und *effizienter Datenübertragung* auf Implementierungsebene.

7. Open-Estelle

Die formale Beschreibungstechnik Estelle wurde zur Spezifikation von Kommunikationssystemen und den darin enthaltenen, miteinander interagierenden Protokollkomponenten entwickelt. Meist bilden dabei bestimmte *Kernkomponenten* (z. B. die Protokollmaschineninstanzen „`PM`“ in Abb. 7-1) das eigentliche Spezifikationsziel, während andere Komponenten hingegen nur als *beispielhafte* Umgebungen für die Anwendung der Kernkomponenten dienen (z. B. „`user`“ und „`network`“).

Diese Ausrichtung auf bestimmte Kernkomponenten manifestiert sich nicht zuletzt bei der (automatisch oder manuell) abgeleiteten Implementierung solcher Protokollkomponenten, wenn diese aufbauend auf *real existierenden Basiskommunikationssystemen* eingesetzt werden sollen. So würde eine (örtlich) verteilte Implementierung des in Abb. 7-1 vorgestellten Kommunikationsszenarios (z. B. mit XTP als Protokollmaschine „PM“) in der Praxis typischerweise eine bestehende Basiskommunikationsinfrastruktur (wie z. B. IP über das Internet) nutzen, anstatt diese als Teil der Implementierung jeweils vollständig neu zu realisieren. Es werden dabei also nur Teile der Spezifikation eigens implementiert. Diese interagieren dann mit anderen Systemen, die nicht unmittelbar als Implementierungen von Spezifikationskomponenten entstanden sind. Dabei bleibt insbesondere unklar, in welcher *semantischen Beziehung* solche Teilimplementierungen mit der zu Grunde liegenden Ausgangsspezifikation stehen (siehe auch Abschnitt 7.3).

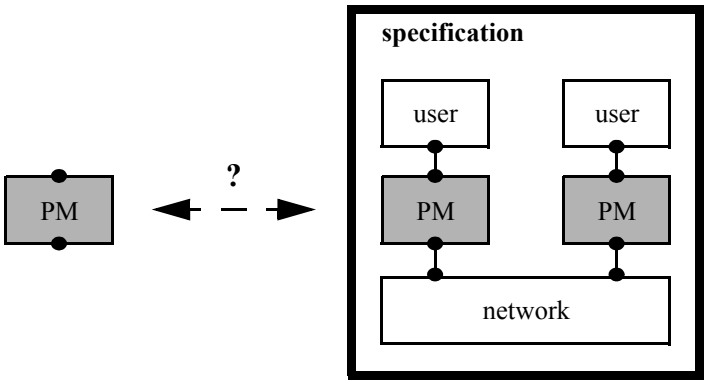


Abbildung 7-1: Offene Systeme (PM) in geschlossenem Gesamtsystem

Offenbar ist zur angemessenen Spezifikation und praxisrelevanten Implementierung solcher Protokoll-Kernkomponenten die Möglichkeit zur Spezifikation und Implementierung *offener (Teil-) Systeme*, also von Systemen mit der Fähigkeit zur Kommunikation mit ihrer Umgebung, wünschenswert. Wir werden uns in diesem Kapitel mit verschiedenen Aspekten der formalen Spezifikation solcher offener Systeme in *Estelle* beschäftigen.

Eine Estelle-Spezifikation beschreibt ein hierarchisches System von Komponenten, den *Modulinstanzen*. Jede Modulinstanz hat eine wohldefinierte *externe Schnittstelle,* durch welche sie mit anderen Modulinstanzen des selben Systems interagieren kann. Das in Estelle formal spezifizierte *Gesamtsystem* ist jedoch *geschlossen*, das heißt, es besteht keine Möglichkeit für eine externe Kommunikation. Auf Grund dieser Restriktion können offene Systeme in Estelle formal nur *zusammen mit* und *eingebettet in* eine konkrete Umgebung spezifiziert werden, sodass das resultierende Gesamtsystem im o. g. Sinne wieder geschlossen ist (siehe Abb. 7-1). Dies bedeutet, dass die *Umgebung* eines offenen Systems vorab bekannt sein muss und somit zusammen mit der Spezifikation des offenen Systems ebenfalls festgelegt wird.

In [GoRoTh96] wurden verschiedene *pragmatische Ansätze* zur Spezifikation von Systemen mit der Fähigkeit zur Interaktion mit existierenden „*realen*“ Umgebungen untersucht. Diese Ansätze basierten auf dem Gebrauch von primitiven Funktionen und Prozeduren oder pragmatischen Modifikationen der Codegeneratoren. Gemäß des Estelle-Standards [ISO97] besitzen derartige Spezifikationen keine formale Semantik,[1] und auch der Import eines derart spezifizierten Systems in einen formalen Kontext (also in eine andere Estelle-Spezifikation) wird durch diesen Ansatz nicht unterstützt.

Die formale Beschreibungstechnik SDL [ITU94] unterstützt die Spezifikation offener Systeme auf syntaktischer Ebene, da es möglich ist, Kanäle mit der Systemgrenze zu verbinden. Offenen Systemen wird auch eine formale Semantik in Form einer Menge kommunizierender Meta-IV-Prozesse zugeordnet. Kommunikation findet jedoch ausschließlich intern statt, da der Austausch von Signalen mit der Umgebung nicht ausgedrückt werden kann. Zudem wird die Komposition der Spezifikationen offener Systeme weder syntaktisch noch semantisch unterstützt.

Die Fähigkeit, offene Systeme *formal* und *unabhängig von einer Umgebung*, sowie ihren davon getrennten *Import*[2] in verschiedene konkrete Umgebungen spezifizieren zu können, hat die folgenden Vorteile:

- Die *Ausdrucksfähigkeit* und das *Abstraktionsniveau* werden gesteigert.

  So können z. B. Protokollmaschinen formal beschrieben und analysiert werden, ohne konkrete Benutzer-Module oder Netzwerk-Module angeben zu müssen. Die formale Semantik der als offenes System spezifizierten Protokollmaschinen zieht dabei alle möglichen Verhaltensweisen der Umgebungen mit in Betracht und ist damit nicht beschränkt auf eine einzelne, explizit spezifizierte Umgebung.

1. Primitive Funktionen und Prozeduren (und damit auch die sie enthaltende Spezifikation) haben keine Semantik, solange keine „*rigorose, implementierungsunabhängige (also mathematische) Definition des relevanten Blocks durch den Spezifizierer angegeben wird“* (Abschnitt 8.2.4.3 von [ISO97]). Wird eine solche primitive Prozedur jedoch zur Kommunikation mit einer Systemumgebung genutzt, so ist die Angabe der Semantik dieser primitiven Prozedur in der geforderten Präzision meist nicht möglich, da dies letztlich die Angabe der Semantik der gesamten Systemumgebung erforderlich macht. Spätestens jedoch bei mehreren Prozeduren als Schnittstelle zur Systemumgebung wird durch eine mögliche (direkte oder indirekte) Kommunikation letztlich eine rekursive Semantikdefinition des Gesamtsystems einschließlich des offenen Systems als Semantik jeder Zugriff-Prozedur erforderlich.

- Die *Dekomposition großer Systeme* in eine Menge von *Komponenten* wird unterstützt.

  Diese begünstigt die praktische Entwicklung von Systemen mit Estelle, da sie es ermöglicht, Systemkomponenten getrennt voneinander zu spezifizieren und analysieren, um sie anschließend zu einem Gesamtsystem zu komponieren.

  Zudem wird die *Wiederverwendung von Komponenten* unterstützt, da diese Komponenten jeweils unabhängig von einer konkreten Umgebung spezifiziert werden können und damit syntaktisch auch nicht von einer solchen abhängen (siehe auch Abb. 7-3).

- Die formale Beschreibung eines offenen Systems kann als Basis zur (automatischen) *Generierung einer Implementierung* mit der Fähigkeit zur Kommunikation mit existierenden „*realen*“ Umgebungen dienen.

  Im Gegensatz zu den oben genannten pragmatischen Ansätzen hat diese Estelle-Spezifikation, auf der die Implementierung basiert, dann eine formale Semantik. Dies ist eine essentielle Grundlage für alle Fragestellungen zur *Korrektheit* dieser Implementierungen.

In diesem Kapitel stellen wir die syntaktische und semantische Estelle-Erweiterung „*Open-Estelle*“ vor. Sie ermöglicht die Spezifikation offener Systeme[3] mit einer wohldefinierten externen Schnittstelle sowie die Spezifikation ihres Imports in verschiedene Umgebungen.

In Abschnitt 7.1 erläutern wir die grundlegenden Konzepte von Open-Estelle. Abschnitt 7.2 führt die syntaktischen Sprachelemente von Open-Estelle ein. In Abschnitt 7.3 definieren wir eine formale Semantik für Open-Estelle, die auf der Standard-Estelle-Semantik basiert und diskutieren auch alternative Semantikansätze. Als eine der wesentlichen semantischen Fundierungen untersuchen wir in Abschnitt 7.4 die Grundlagen der textuellen Verschmelzung einer Menge offener Systeme hin zu einer geschlossenen Standard-Estelle-Spezifikation. Abschnitt 7.5 behandelt Implementierungsaspekte einschließlich der Werkzeugunterstützung für Open-Estelle. Die Wiederverwendung von Protokollkomponenten bis hin zur Bildung von generischen Komponentenbibliotheken wird schließlich in Abschnitt 7.6 untersucht.

---

2. Wir trennen hier zur begrifflichen Vereinfachung nicht zwischen der *Beschreibung eines offenen Systems* (also der Spezifikation) und dem offenen System selbst (also der *Instanz* der Beschreibung). Entsprechend wird der Begriff „*Import*“ in diesem Text ebenfalls dual verwendet: Neben dem (wie auch immer realisierten) Import der Beschreibung eines offenen Systems in die Beschreibung einer Umgebung kann darunter auch der Import des offenen Systems selbst (als Instanz der Beschreibung) in ein umgebendes System verstanden werden. Letzteres kann auch als „Inkorporation“ (also physische Einbettung) eines Teilsystems in ein anderes System verstanden werden. Dies steht jedoch in keinem Zusammenhang zu der Frage, inwieweit z .B. die Beschreibung des offenen Systems textuell in die der Umgebung eingebettet ist oder ein formaler Import erfolgt (s. u.).

   Die Differenzierung der Begriffe hat teilweise recht subtile Auswirkungen. So verbleibt die Aufgabe der Erzeugung einer Instanz eines offenen Systems beim Import auf Beschreibungsebene bei der Umgebung, wohingegen beim Import auf Instanzebene die Instanziierung offensichtlich bereits erfolgt ist. Die Unterscheidung wird insbesondere bei der Betrachtung nicht beliebig oft instanziierbarer Teilsysteme (z. B. das Internet als Ganzes) deutlich.

3. also Systeme mit der Fähigkeit zur Interaktion mit ihrer Umgebung

7.1. Grundkonzepte von Open-Estelle

Die Hauptzielsetzung von Open-Estelle ist die Bereitstellung einer Sprachunterstützung für die *formale Beschreibung offener Systeme* und ihres *Imports in verschiedene Umgebungen*. Dies bedeutet, dass ein offenes System nicht nur eine wohldefinierte syntaktische und semantische Interpretation unabhängig von seiner Umgebung hat, sondern auch als Teil eines zusammengesetzten Systems. Im Folgenden führen wir einige Grundkonzepte der syntaktischen und semantischen Spezifikation offener Systeme und ihres Imports in andere Systeme unter Open-Estelle ein.

7.1.1 Repräsentation offener Systeme

Eine grundlegende Designentscheidung von Open-Estelle betrifft die Repräsentation eines offenen Systems. Standard-Estelle beinhaltet bereits eine Abstraktion zur Beschreibung eingekapselter Systeme mit wohldefinierter externer Schnittstelle und ihrer Aggregation zu komplexeren Systemen: *Module*.

Modulinstanzen könnten prinzipiell eine geeignete Repräsentation offener Systeme in Estelle bieten: Der *Modulheader* beschreibt ein externes Interface der Modulinstanz, ohne Bezug auf ihre interne Beschreibung, der *Modulrumpf* beschreibt das Verhalten der Modulinstanz.

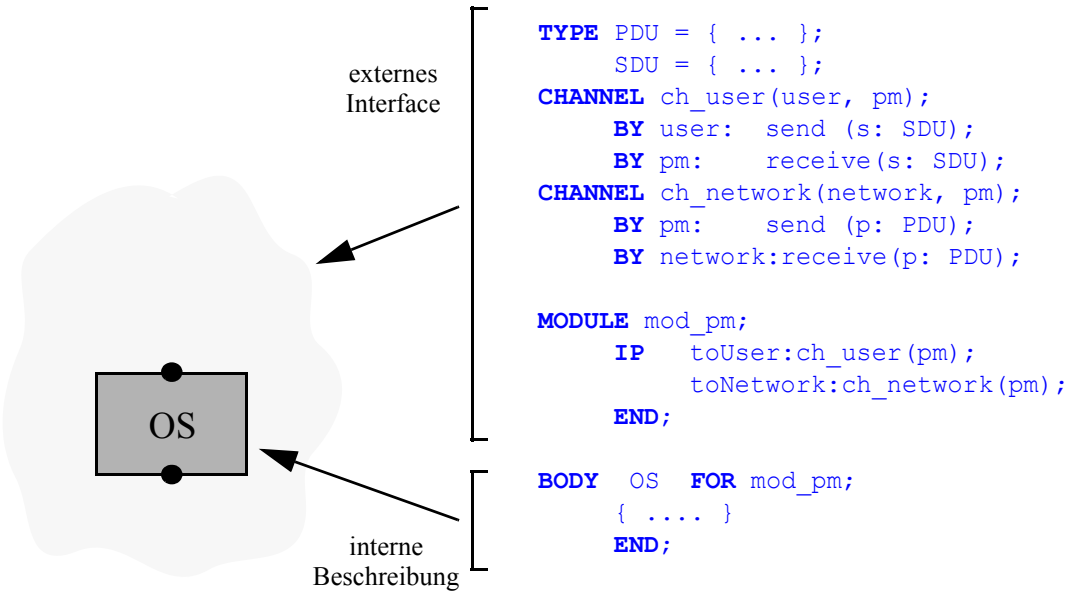


Abbildung 7-2: Externe und interne Beschreibung offener Systeme auf Modulbasis

Während die Modulrumpfdefinition bis auf den Bezug auf den Modulheader zunächst[4] als syntaktisch zusammenhängende Definition formuliert werden kann, sind zur Definition eines Modulheaders typischerweise weitere zu Grunde liegende Definitionen von Kanälen, Typen und Konstanten erforderlich (siehe Abb. 7-2), die als Teil der formalen Definition der externen Schnittstelle ebenfalls formal (d.h. syntaktisch und semantisch eindeutig) definiert sein müssen.

4. Die zur Definition der externen Schnittstelle definierten Kanäle, Typen und Konstanten werden im Normalfall auch in der internen Definition des Moduls bzw. offenen Systems referenziert.

Offensichtlich sind Module geeignete und natürliche Mittel zur Beschreibung offener Systeme, und in der Tat basiert die Beschreibung offener Systeme in Open-Estelle auf Standard-Estelle-Modulen. Der wesentliche Unterschied besteht jedoch darin, dass in Standard-Estelle ein Modul nur *als Teil eines geschlossenen Systems* definiert und genutzt werden kann. In einem geschlossenen System, das ausschließlich das offene System enthält, wäre jedoch die Umgebung im obigen Sinne leer und entsprechend zu keiner sinnvollen[5] Interaktion fähig. Weiterhin hat ein Modul als Fragment einer Spezifikation keine eigene formale Semantik. Entsprechend müssen aufbauend auf Standard-Estelle die folgenden Aspekte behandelt werden:

- Eine *syntaktische Erweiterung* zur formalen Definition von Modulen, die offene Systeme beschreiben, unabhängig von einer Umgebung.
- Eine *formale Semantik* für offene Systeme, sowohl unabhängig von ihrer Umgebung, als auch als Teil eines zusammengesetzten Systems.
- Ein Mechanismus, der es anderen Spezifikationen (also Spezifikationen, die eine konkrete Umgebung für ein offenes System definieren) erlaubt, die externen Definitionen eines offenen Systems syntaktisch zu *importieren* (siehe auch Abb. 7-3).

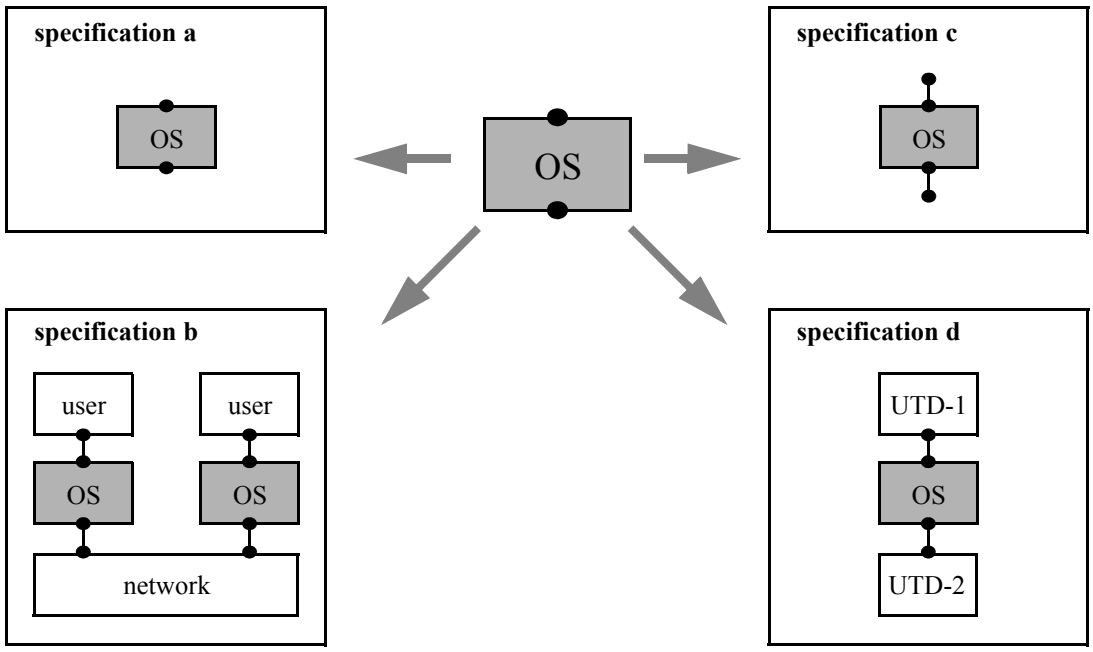


Abbildung 7-3: Import offener Systeme in verschiedene Umgebungen

5. Die Modulinstanz kann natürlich Nachrichten durch ihre (unverbundenen) externen Interaktionspunkte verschicken, jedoch werden diese von der (leeren) Umgebung nicht empfangen, da sie verworfen werden. Analog haben auch Wertzuweisungen auf exportierte Variablen keinen kommunikationsbezogenen Effekt auf die Umgebung. Somit kann zusammen mit dem Ausbleiben jeglicher Stimuli von außen keine sinnvolle Kommunikation über die äußere Schnittstelle erfolgen.

7.1.2 Formaler Import vs. textueller Inklusion

Wie wir bereits gesehen haben, muss die formale Beschreibung eines offenen Systems eine eindeutige syntaktische und semantische Interpretation sowohl für das offene System selbst als auch für seine Importe in alle *potenziellen Umgebungen* haben. Während die semantische Interpretation eines offenen Systems (sein „*Verhalten*“, engl. „*behaviour*“) von den möglichen Interaktionen mit allen Umgebungen abhängt, sollte seine syntaktische Interpretation vollständig unabhängig von den jeweiligen importierenden Umgebungen sein. Insbesondere sollten interne Definitionen einer beliebigen potenziellen Umgebung, die die Beschreibung des offenen Systems importiert, keinerlei Einfluss auf die Interpretation des offenen Systems oder seiner externen Schnittstelle haben.

Die folgenden Beispiele zeigen, dass diese Anforderungen sämtliche Lösungen disqualifizieren, die auf *einfacher textueller Inklusion*[6] der Spezifikation eines offenen Systems in eine importierende Umgebung basieren. In diesen Beispielen gehen wir davon aus, dass ein offenes System durch ein Estelle-Spezifikationsfragment[7] definiert wird. Wir diskutieren dann die Konsequenzen einer textuellen Inklusion (hier dargestellt durch ein Präprozessor-„`#include`“-Statement) dieser Fragmente in eine importierende Estelle-Spezifikation.

Beispiel 7.62-a: Interferenz zwischen einem importierten Interface und seiner importierenden Umgebung.

Das Estelle-Spezifikationsfragment OS1 definiere ein offenes System, das von einer vordefinierten Funktion, Prozedur[8] oder einem vordefinierten Typ[9] Gebrauch macht.
Weiterhin möge die Estelle-Spezifikation S1 diesen Namen redefinieren wie es zum Beispiel anhand des Namens „`integer`“ im folgenden Spezifikationsfragment geschieht:

```
SPECIFICATION S1;
     TYPE integer = 1 .. 10;            (* redefinition of predefined *)
                                        (* type integer in this scope *)
     MODULE M FOR {...};
          #include OS1                (* textually include text fragment *)
     END;
END.
```

Die *textuelle Inklusion* des offenen Systems OS1 in das Modul M von Spezifikation S1 führt zu einer Gesamtspezifikation, die nach den syntaktischen und semantischen Regeln von Estelle interpretiert wird. Dabei werden insbesondere die in OS1 auftretenden Definitionen nunmehr im Kontext von Modul M interpretiert, in diesem Fall also mit dem redefinierten Typ (bzw. Namen) „`integer`“. Während OS1 syntaktisch nach wie vor auf den Namen „`integer`“ Bezug nimmt, bezieht sich dieser Name nicht länger auf den vordefi-

---

6. also die unveränderte textuelle Einbettung eines (Spezifikations-) Textfragmentes in einen anderen Spezifikationstext

7. z. B. eine Modul-Definition zusammen mit allen notwendigen Konstanten-, Typ-, Kanal- und Modulheader-Definitionen (siehe auch Abb. 7-2 auf Seite 316)

8. siehe Abschnitt 6.6.4, ff. von Annex C [ISO97] („*Required Procedures and Functions*“), z. B. „`ord`“ oder „`sin`“

9. siehe Abschnitt 6.4.2.2 von Annex C [ISO97] („*Required Simple-Types*“), z. B. „`integer`“ oder „`boolean`“

nierten Typ, sondern auf seine Redefinition (siehe auch Abb. 7-4). Dies kann erhebliche Auswirkungen auf die syntaktische und semantische Interpretation des offenen Systems OS in dieser konkreten Umgebung haben.

(Ende von Beispiel 7.62-a)

Die im vorangegangenen Beispiel herangezogene Redefinition von vordefinierten Typnamen besitzt in der Praxis zwar wohl nur geringe Relevanz, zeigt aber die Grundidee der Interferenz durch textuellen Import. Im nächsten Beispiel wird dagegen eine Interferenz zwischen zwei parallel importierten offenen Systemen gezeigt, die bei bereits vorgegebenen offenen Systemen letztlich kaum zu vermeiden ist.

Beispiel 7.62-b: Interferenz zwischen zwei parallel importierten offenen Systemen

Die Estelle-Spezifikationsfragmente OS1 und OS2 definieren zwei offene Systeme, die (per Koinzidenz) den selben Namen definieren (z. B. verschiedene Typen namens „PDU“ für verschiedene „*Protocol Data Units*“ der offenen Systeme, siehe auch Abb. 7-4). Die *textuelle Inklusion* beider offenen Systeme in den selben Kontext führt zu einer syntaktisch inkorrekten Spezifikation, da eine unzulässige Namens-Redefinition im selben Kontext entsteht.

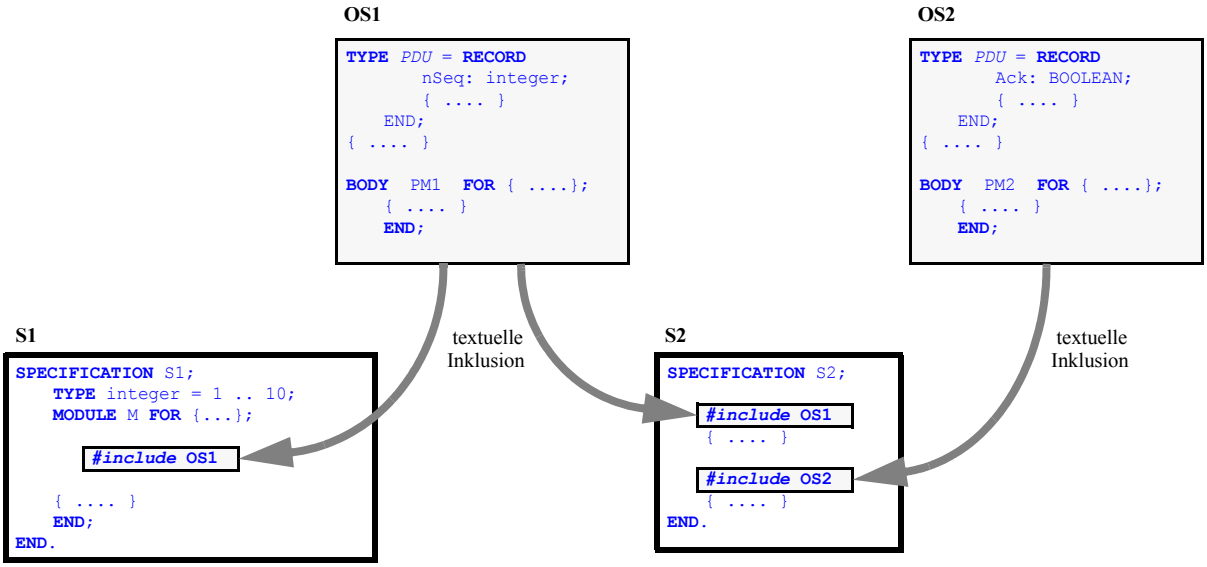


Abbildung 7-4: Interferenzen durch einfache textuelle Inklusion

(Ende von Beispiel 7.62-b)

Offensichtlich hängt die syntaktische Interpretation der Definitionen aus einer *textuell importierten* Beschreibung eines offenen Systems auf kritische Weise sowohl vom Kontext, in den eingefügt wird, als auch von der Kombination der importierten Beschreibungen von offenen Systemen ab. Dies macht die syntaktische Definition derartiger offener Systeme *nicht-formal*.

Um dagegen eine formale Definition eines offenen Systems zu gewinnen, muss seine *Interpretation strikt unabhängig von jeder beliebigen Umgebung* sein, welche Gebrauch von dem offenen System macht. Insbesondere müssen die Namensräume verschiedener offener Systeme und importierender Umgebungen voneinander getrennt sein.

In Abschnitt 7.2 werden wir sehen, dass *Open-Estelle* diese Probleme vermeidet, da

(i) die syntaktischen Definitionen, die innerhalb der Beschreibung eines offenen Systems gegeben werden, auf eindeutige Weise interpretiert werden, unabhängig von einer konkreten importierenden Umgebung (*Wohldefiniertheit*, siehe Abschnitt 7.2.1 und 7.2.2);

(ii) die Definitionen, die in einem offenen System gemacht werden, nicht *textuell importiert*, sondern auf wohldefinierte Art und Weise *formal importiert* werden; dieser formale Import gewährt dabei lediglich Zugriff auf die (eindeutigen) Definitionen des offenen Systems (siehe Abschnitt 7.2.3);

(iii) alle Bezeichner, die aus der Beschreibung eines offenen Systems importiert werden, durch den Namen des offenen Systems *qualifiziert* werden (z. B. „`OS1::PDU`“ und „`OS2::PDU`“ gem. Beispiel 7.62-b); dies vermeidet Namenskonflikte zwischen verschiedenen externen Schnittstellen von offenen Systemen oder mit Namen innerhalb der importierenden Umgebung; darüber hinaus verbessert dies die Lesbarkeit der Spezifikation, da die Herkunft eines jeden importierten Namens explizit angegeben ist (siehe Abschnitt 7.2.3).

7.1.3 Trennung zwischen Deklaration und Definition von offenen Systemen

In Open-Estelle besteht die Beschreibung eines offenen Systems aus der externen Schnittstelle und der internen Beschreibung, welche (im Sinne separater Übersetzungseinheiten) *textuell voneinander getrennt* sind. Beide sind syntaktisch unabhängig von jeder möglichen Umgebung, die Gebrauch von dem offenen System macht. Darüber hinaus beziehen sich Open-Estelle-Umgebungen, die ein offenes System importieren, ausschließlich auf dessen *externe* Schnittstelle (siehe Abb. 7-5).

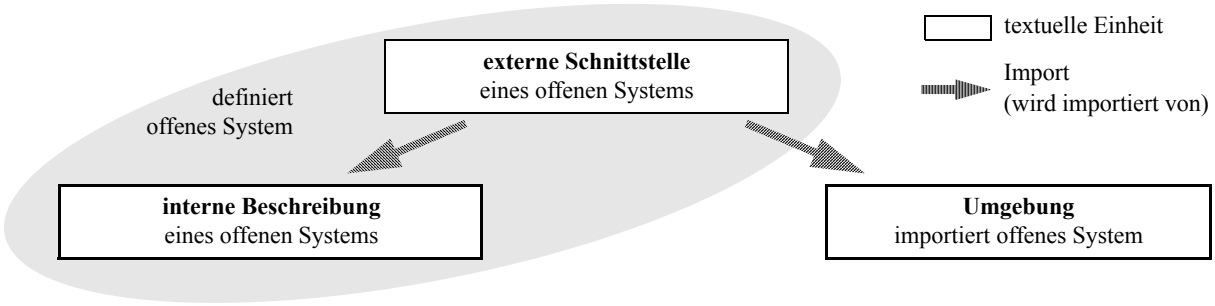


Abbildung 7-5: Unabhängige Beschreibung von offenem System und Umgebung

Eine „*externe Schnittstelle*“ ***deklariert*** offene Systeme (d. h. Module) zusammen mit allen erforderlichen zu Grunde liegenden Definitionen (d. h. Modulheader, Kanäle, Typen und Konstanten; siehe Abschnitt 7.2.1).[10]

10. Externe Schnittstellen können auch andere externe Schnittstellen importieren und Gebrauch von ihren Definitionen machen (siehe Abschnitt 7.2.3).

Eine „*interne Beschreibung*“ ***definiert*** diese offenen Systeme, indem sie die internen Aspekte beschreibt (siehe Abschnitt 7.2.2). Diese Trennung macht es möglich, offene Systeme und importierende Umgebungen unabhängig voneinander zu entwickeln und zu kompilieren, sobald ihre gemeinsame Schnittstelle definiert wurde. Darüber hinaus unterstützt sie die Trennung der jeweils privaten Aspekte von offenen Systemen und ihren möglichen Umgebungen.

Die textuelle Trennung zwischen der äußeren Schnittstelle und der internen Beschreibung eines Systems ist motiviert durch die Grundkonzepte von Modula-2 [Wir85]. Ebenso ist die Unterscheidung zwischen *Deklaration* und *Definition* eines syntaktischen Objekts ähnlich zu den entsprechenden Konzepten in Modula-2, C oder C++: Eine *Deklaration* definiert Namen und Typ (also die externe Schnittstelle) eines Objekts, während eine *Definition* interne Details hinzufügt.[11]

Die Aufteilung der Teilbeschreibungen in *getrennte textuelle Einheiten*[12] ist eine der wesentlichen konzeptionellen Grundlagen von Open-Estelle: Sie ermöglicht die separate und kontextunabhängig interpretierbare Beschreibung der äußeren Schnittstelle eines offenen Systems und vermeidet damit weiter Abhängigkeiten zwischen den von ihr abhängigen Definitionen, also zugehörigen offenen Systemen und importierenden Umgebungen.

Offensichtlich ist zur eindeutigen und syntaktisch korrekten Spezifikation einer solchen abhängigen Definition eines offenen Systems die *Zuordnung* einer entsprechenden externen Schnittstelle (welche wiederum in einer getrennten textuellen Einheit angegeben wird) notwendig. Wir werden später anhand der Open-Estelle-Syntax in Abschnitt 7.2 sehen, dass dies eine *Abbildung* eines Namens[13] auf eine andere textuelle Einheit (nämlich die zu importierende externe Schnittstelle) erfordert.

Eine solche Abbildung überschreitet prinzipbedingt die Grenzen einer einzelnen textuellen Einheit und muss daher *auf der Metaebene* erfolgen. Im Falle eines hierarchischen Dateisystems mit Textdateien zur Darstellung der textuellen Einheiten kann dies z. B. auf Basis einer geeigneten Dateinamenskonvention und eines Verzeichnissuchpfades erfolgen (siehe auch Toolsupport in Abschnitt 7.4 und 7.5).

Wir werden später sehen, dass eine solche Zuordnung bei der *Komposition* offener (Teil-) Systeme und importierender Umgebungen zu einem (offenen oder geschlossenen) Gesamtsystem nochmals erforderlich sein wird. Hintergrund ist, dass Open-Estelle es erlaubt,

- verschiedene importierende Umgebungen zu definieren, die auf die selbe externe Schnittstelle (bzw. auf ein offenes System mit dieser Schnittstelle) zugreifen und
- verschiedene offene Systeme (genauer: verschiedene interne Beschreibungen) zu der gleichen externen Schnittstelle zu definieren, auf die von den importierenden Umgebungen anhand der externen Schnittstelle zugegriffen wird.

---

11. So ist zum Beispiel in C „`int succ(int n);`“ eine Funktions-*Deklaration* und „`int succ(int n) {return n+1;}`“ eine dazu passende Funktions-*Definition*.

    In der auf Standard-Pascal basierenden Estelle-Grammatik werden diese Begriffe bei der Vergabe von Namen für Nichtterminale gelegentlich etwas unschärfer eingesetzt (siehe auch Abschnitt 7.2).

12. Da der Estelle-Standard keine Repräsentation für den Spezifikationstext angibt, benutzen wir den Ausdruck „*textuelle Einheit*“ für abgeschlossene syntaktische Objekte, wie zum Beispiel eine komplette Spezifikation. In einer Unix-Umgebung wird eine solche textuelle Einheit typischerweise als *ASCII-Textdatei* repräsentiert.

13. z. B. aus einem IMPORT-STATEMENT (s. u.)

Dadurch entstehen für offene Systeme und importierende Umgebungen zu einer gemeinsamen Schnittstellendefinition zwei Freiheitsgrade, über die Komponenten zu einem Gesamtsystem ausgewählt werden können (siehe auch Abb. 7-16 auf Seite 343).

Auch hier ist eine geeignete Zuordnung der beteiligten Komponenten[14] erforderlich. Wir kommen im weiteren Verlauf dieses Kapitels gelegentlich noch auf diese Fragestellung zurück. Es genügt zunächst jedoch die Abstraktion, dass die erforderlichen Abbildungen und Zuordnungen eindeutig und zulässig erfolgen.

7.1.4 Semantik offener Systeme

Open-Estelle wurde entwickelt, um *offene Systeme* und ihren *Import in beliebige Umgebungen* formal beschreiben zu können. Entsprechend sollte ein offenes System eine formale Semantik auf zwei Ebenen besitzen:

- als offenes System selbst, *unabhängig von jeder konkreten Umgebung* und
- *importiert in eine konkrete Umgebung*.

Das Design von Open-Estelle ist dabei darauf ausgerichtet, dass gerade der zweite Punkt, der Import eines offenen Systems in eine konkrete Umgebung, vollständig auf die Standard-Estelle-Semantik abgebildet werden kann. Dies gelingt, da Estelle mit dem Konzept des Moduls bereits eine syntaktische und semantische Abstraktion zur Beschreibung offener Systeme beinhaltet, obgleich in Standard-Estelle derartige offene Systeme ausschließlich integriert in ein (letztlich geschlossenes) System dargestellt werden können. Open-Estelle nutzt und erweitert diese existierende Abstraktion eines offenen Systems, indem

(i) offene Systeme als *Estelle-Module* modelliert werden, deren interne Spezifikation nunmehr zusammen mit der Spezifikation ihrer externen Schnittstelle (siehe Abb. 7-2 auf Seite 316) in separaten textuellen Einheiten (Dateien) abgelegt werden kann;

(ii) diese Spezifikationen unabhängig von einer möglichen späteren Verwendung in anderen Kontexten *syntaktisch* (und wie wir später sehen werden auch *semantisch*) *eindeutig interpretiert* werden;

(iii) der Zugriff auf die Spezifikation des offenen Systems und seine externe Schnittstelle durch einen syntaktisch *formalen Import* in eine andere Spezifikation erfolgt, der diese externen Definitionen auf syntaktischer Ebene strikt unverändert zugänglich macht.

Als Ergebnis dieser Kette kann die importierende Spezifikation auf die importierten Definitionen im Wesentlichen zugreifen, als wären es lokale Definitionen.[15] Entsprechend erfolgt die Nutzung des offenen Systems in das derart beschriebene Kontext-System durch die simple Instanziierung der importierten Moduldefinition. Hier besteht auf semantischer Ebene kein Unterschied zur Instanziierung eines lokal beschriebenen Moduls.

Die Instanziierung eines als offenes System importierten Estelle-Moduls erzeugt also auf semantischer Ebene exakt die selbe Modulstruktur wie die Instanziierung eines entsprechenden lokalen (also textuell eingeschlossenen) Kindmoduls. Daraus ergibt sich eine intuitive Semantik

14. genauer: die Abbildung jeweils einer MODULE-BODY-DECLARATION auf eine passende BEHAVIOUR-DEFINITION (siehe Abschnitt 7.2)

15. Die importierten Namen müssen jedoch qualifiziert angegeben werden (siehe Abschnitt 7.2.3).

der Anwendung offener Systeme. Mehr noch macht es diese Semantik möglich, die Beschreibungen von importierenden Umgebungen und den von ihnen importierten offenen Systemen *zu einer einzigen Spezifikation zu verschmelzen*, die semantisch äquivalent ist. Wir werden uns später in Abschnitt 7.4 mit derartigen Verschmelzungen und den dabei notwendigen Transformationen genau beschäftigen.[16]

Ein wichtiger Aspekt von Open-Estelle ist die Definition einer Semantik für *offene Systeme unabhängig von einer konkreten Umgebung*. Im Spezialfall eines offenen Systems ohne irgendwelche externen Interaktionspunkte und exportierten Variablen ist seine Semantik identisch mit der Semantik einer entsprechenden (geschlossenen) Estelle-Spezifikation: Sie ist unabhängig[17] von einer ggf. existierenden konkreten Umgebung, da es keine Möglichkeiten zur Interaktion mit dieser gibt. Wenn das offene System jedoch externe Interaktionspunkte oder exportierte Variablen besitzt, kann eine konkrete Umgebung mit dem offenen System interagieren durch

(i) *Zugriff auf exportierte Variablen* und

(ii) den Austausch von *Nachrichten durch externe Interaktionspunkte*.

Da die Semantik eines offenen Systems ohne eine konkrete Umgebung keinen Bezug auf die spezifischen Eigenschaften und Beschränkungen einer *möglichen*[18] späteren konkreten Umgebung nehmen kann, muss *jede mögliche Beeinflussung über die äußere Schnittstelle auf das offene System* berücksichtigt werden, also jede denkbare Folge von Nachrichten, die durch die externen Interaktionspunkte empfangen werden kann *(ii)*, und jede denkbare Modifikation von exportierten Variablen *(i)*.[19]

Sobald man das offene System jedoch in eine *konkrete Umgebung* importiert, wird diese maximale Menge möglicher Interaktionen reduziert auf die Teilmenge von tatsächlich auftretenden Interaktionen mit der jeweiligen Umgebung.

In Abschnitt 7.3 werden wir dieses Konzept im Detail diskutieren.

16. Eine einfache (unveränderte) textuelle Einbettung wäre dagegen ungeeignet, wie wir bereits in Abschnitt 7.1.2 gesehen haben.

17. Diese Unabhängigkeit gilt bzgl. der Standard-Estelle-Semantik nur auf intuitiver Ebene. Technische Synchronisationsaspekte wie Auswahlphasen, die Beteiligung an Subsystemen und mögliche Vater-Sohn-Prioritäten sind dabei nicht berücksichtigt. (Siehe auch Abschnitt 7.3.3.)

18. Die Umgebung muss natürlich syntaktisch zur externen Schnittstelle des offenen Systems passen. Dadurch werden die möglichen Wechselwirkungen zumindest auf diese Schnittstelle beschränkt.

19. Diese Art der *maximalen Interaktion* ist auch die Grundlage für den „*Universal Test Drivers Generator*“ (UTDG, [LaLeMa91]). Der UTDG generiert Module, die indeterministisch durch ihre externen Interaktionspunkte jede Interaktion senden oder empfangen können und jede Modifikation ihrer exportierten Variablen durchführen können. Er wurde entwickelt, um unspezifizierte Systemkomponenten während des Tests von Estelle-Spezifikationen simulieren zu können. Da dieser Ansatz jedoch die syntaktischen Probleme nicht lösen kann, die wir in Abschnitt 7.1.2 vorgestellt haben, ist er nicht direkt für die formale Spezifikation offener Systeme geeignet.

7.2. Syntax von Open-Estelle

In diesem Abschnitt führen wir die syntaktischen Erweiterungen von Estelle zur Beschreibung

(i) der äußeren Schnittstelle von offenen Systemen (Abschnitt 7.2.1),

(ii) der internen Beschreibung von offenen Systemen (Abschnitt 7.2.2) und

(iii) des formalen Imports offener Systeme in verschiedene Umgebungen (Abschnitt 7.2.3)

anhand von Beispielen ein. Diese Erweiterungen basieren auf dem syntaktischen Konzept des Moduls. Die formale Definition von Open-Estelle ist in Anhang C zu finden (siehe auch [ThGo97a]).

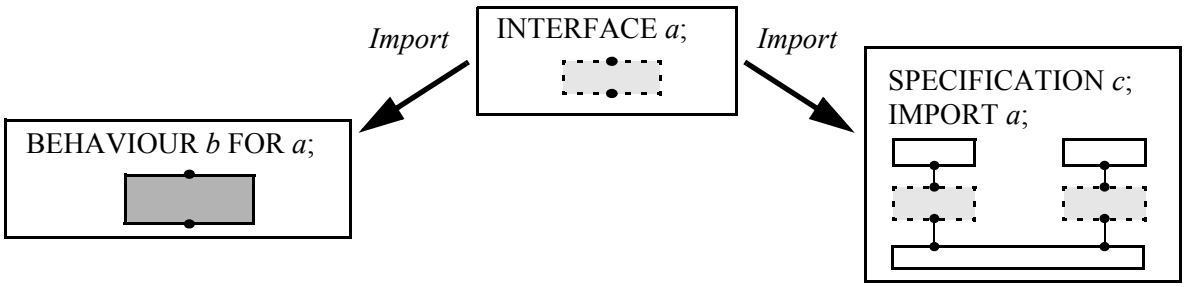


Abbildung 7-6: Trennung von offenen Systemen und importierenden Umgebungen

7.2.1 Interface-Definition offener Systeme

Die Deklaration[20] (und damit die Festlegung der externen Schnittstelle) eines offenen Systems erfolgt in einer INTERFACE-DEFINITION (siehe Abb. 7-7[21]), die in einer separaten textuellen Einheit abgelegt wird. Eine INTERFACE-DEFINITION ist ein Container für die Deklaration (nicht Definition) einer Menge offener Systeme und aller Definitionen, die zur Beschreibung ihres Typs und damit ihrer externen Schnittstelle erforderlich sind (also Modulheader, Kanäle, Typen und Konstanten, siehe Abb. 7-8).

Die INTERFACE-DEFINITION ist eine der beiden *neuen Start-Nichtterminale* der Open-Estelle-Syntax. Interfaces sind eindeutig definiert, unabhängig von jeder importierenden Umgebung (siehe Abschnitt 7.2.3). Dies ist wichtig für eine formale Beschreibung der exportierten Definitionen. Ein Interface beginnt mit dem neuen Schlüsselwort „**INTERFACE**“, gefolgt durch seinen Namen. Die Deklaration eines offenen Systems innerhalb eines Interfaces wird syntaktisch definiert durch eine MODULE-BODY-DECLARATION, welches eine MODULE-BODY-DEFINITION[22] mit dem Schlüsselwort „**EXTERNAL**“ ist (siehe Abb. 7-8). Diese Deklaration bezieht sich auf einen Modulheader, der die äußere Schnittstelle des Moduls als offenes System definiert, jedoch *keine* interne Beschreibung dazu abgibt.

20. Die INTERFACE-DEFINITION stellt in Bezug auf das offene System nur eine *Deklaration* (also die Festlegung einer äußeren Schnittstelle, siehe Abschnitt 7.1.3) dar.

21. Die Nichtterminal-Namen INTERFACE-DECLARATIONS und INTERFACE-DECLARATION-PART wurden analog zu den entsprechenden Nichtterminal-Namen DECLARATIONS und DECLARATION-PART der Standard-Estelle-Grammatik gewählt (siehe Abschnitt 7.3.1 von Annex C [ISO97]). In beiden Fällen beinhalten diese jedoch überwiegend Definitionen.

```
INTERFACE-DEFINITION =          „INTERFACE“ IDENTIFIER „;“
                                [ DEFAULT-OPTIONS ]
                                [ IMPORT-OPTIONS ]
                                INTERFACE-DECLARATION-PART
                                „END“ „.“ .
INTERFACE-DECLARATION-PART =    { INTERFACE-DECLARATIONS } .
INTERFACE-DECLARATIONS =        CONSTANT-DEFINITION-PART
                                | TYPE-DEFINITION-PART
                                | CHANNEL-DEFINITION
                                | MODULE-HEADER-DEFINITION
                                | MODULE-BODY-DECLARATION .
MODULE-BODY-DECLARATION =       „BODY“ IDENTIFIER „FOR“ HEADER-IDENTIFIER „;“
                                „EXTERNAL“ „;“ .
```

Abbildung 7-7: Syntax einer Interface-Definition

Analog zu Standard-Estelle macht das Auftreten des Schlüsselworts „**EXTERNAL**“ innerhalb einer MODULE-BODY-DECLARATION die umgebende INTERFACE-DEFINITION zu einer syntaktisch korrekten, aber *unvollständigen* Beschreibung. Jedoch kann aus einer solchen INTERFACE-DEFINITION bereits durch die *Zuordnung* (siehe Abschnitt 7.1.3) einer geeigneten BEHAVIOUR-DEFINITION (siehe Abschnitt 7.2.2) eine vollständige Beschreibung des deklarierten offenen Systems gewonnen werden. Dabei wird die INTERFACE-DEFINITION weder textuell modifiziert, noch ändert sich ihre syntaktische Interpretation; allein die auf Metaebene erfolgende *logische Zuordnung* der textuell getrennten BEHAVIOUR-DEFINITION zu ihrer INTERFACE-DEFINITION durch den Spezifizierer vervollständigt die Beschreibung des offenen Systems.[23]

```
INTERFACE binaryService;

    TYPE tOperand = RECORD x1, x2: REAL; END;
         tResult = REAL;

    CHANNEL binaryServiceChannel (user, provider);
         BY user: request (x: tOperand);
         BY provider: respond (y: tResult);

    MODULE binaryOperatorHeader ACTIVITY;
         IP toUser: binaryServiceChannel (provider) COMMON QUEUE;
         END;

    BODY binaryOperator FOR binaryOperatorHeader;
         EXTERNAL;

END.
```

Abbildung 7-8: Beispiel einer Interface-Definition

22. Eine solche MODULE-BODY-**DEFINITION** mit dem Schlüsselwort „**EXTERNAL**“ stellt in diesem Zusammenhang eigentlich nur eine *Deklaration* des offenen Systems dar. Wir verzichten an dieser Stelle jedoch auf diese technisch nicht erforderliche Differenzierung innerhalb der Estelle-Grammatik, da das Nichtterminal MODULE-BODY-**DEFINITION** in der Standard-Estelle-Grammatik bereits entsprechend geprägt ist.

23. im Gegensatz zu einer SPECIFICATION oder einer BEHAVIOUR-DEFINITION, bei denen die Unvollständigkeit einer „**EXTERNAL**“-MODULE-BODY-DEFINITION nur durch eine *textuelle Modifikation* aufgelöst werden kann

Es ist zu beachten, dass die Interface-Definition selbst weder einen Zustandsraum,[24] noch ein konkretes Verhalten[25] für das offene System definiert. Entsprechend ist sie nur ein Container für eine Menge von Definitionen, die von anderen Estelle-Komponenten importiert werden (siehe Abschnitt 7.2.3).

Es ist ebenfalls möglich, in eine Interface-Definition andere Interfaces zu importieren. Dies ist insbesondere vor dem Hintergrund zu sehen, dass Interfaces nicht zwingend offene Systeme deklarieren, sondern auch nur zur importierbaren, formalen Definition von Konstanten, Datentypen, Kanälen oder Modulheadern dienen können . So können diese Definitionen in verschiedenen importierenden Umgebungen (also anderen Interfaces, Behaviour-Definitionen oder Spezifikationen) gemeinsam genutzt werden. Dies ermöglicht den Aufbau einer feingranularen Hierarchie von Definitionen und unterstützt dadurch eine saubere Trennung der Angelegenheiten von importierenden Umgebungen und gemeinsamen Definitionen. Wir werden in Abschnitt 7.2.3 anhand des formalen Imports nochmals auf diese Anwendung zurückkommen.

7.2.2 Interne Beschreibung offener Systeme

Die interne Beschreibung eines offenen Systems wird durch eine MODULE-BODY-DEFINITION innerhalb einer BEHAVIOUR-DEFINITION (siehe Abb. 7-9[26]) angegeben, welche in einer separaten textuellen Einheit abgelegt ist. Wie der Ausdruck BEHAVIOUR-DEFINITION bereits suggeriert, definiert die interne Beschreibung eines offenen Systems aus einer abstrakten Sichtweise heraus ausschließlich sein Verhalten (engl. „*behaviour*“), da die syntaktische externe Schnittstelle bereits durch die Interface-Definition vollständig determiniert ist.

```
BEHAVIOUR-DEFINITION =          „BEHAVIOUR“ IDENTIFIER
                                „FOR“ INTERFACE-IDENTIFIER „;“
                                [ DEFAULT-OPTIONS ]
                                [ TIME-OPTIONS ]
                                [ IMPORT-OPTIONS ]
                                BEHAVIOUR-DECLARATION-PART
                                „END“ „.“ .
BEHAVIOUR-DECLARATION-PART =    { BEHAVIOUR-DECLARATIONS } .
BEHAVIOUR-DECLARATIONS =        MODULE-BODY-DEFINITION .
INTERFACE-IDENTIFIER =          IDENTIFIER .
```

Abbildung 7-9: Syntax einer Behaviour-Definition

Die BEHAVIOUR-DEFINITION ist das zweite neue Start-Nichtterminal der Open-Estelle-Syntax. Sie beginnt mit dem neuen Schlüsselwort „**BEHAVIOUR**“,[27] gefolgt von ihrem Namen und einer Referenz auf ihr Interface (siehe Abb. 7-10). Die Definitionen des Interfaces werden dabei implizit formal importiert (siehe Abschnitt 7.2.3).

24. sie definiert keine Variablen oder Kontrollzustände

25. sie enthält keine Transitionen

26. Die Nichtterminal-Namen BEHAVIOUR-DECLARATIONS und BEHAVIOUR-DECLARATION-PART wurden analog zu den entsprechenden Nichtterminal-Namen DECLARATIONS und DECLARATION-PART der Standard-Estelle-Grammatik gewählt (siehe Abschnitt 7.3.1 von Annex C [ISO97]). In beiden Fällen beinhalten diese jedoch überwiegend Definitionen.

```
BEHAVIOUR binaryAdder FOR binaryService;
    BODY binaryOperator FOR binaryService::binaryOperatorHeader;
        TRANS
            WHEN toUser.request(x: tOperand)
            BEGIN
                OUTPUT toUser.respond( x.x1 + x.x2 );
            END;
    END;
END.
```

Abbildung 7-10: Beispiel einer Behaviour-Definition

Eine BEHAVIOUR-DEFINITION ist ein Container für eine Menge von Definitionen offener Systeme: Für jede MODULE-BODY-*DECLARATION* ihres Interfaces enthält sie genau eine passende[28] MODULE-BODY-*DEFINITION* und umgekehrt. Jeder dieser Modulrümpfe definiert das interne Verhalten eines offenen Systems und kann dabei intern von allen (Open-) Estelle-Konstrukten für Modulrümpfe Gebrauch machen. So können die Modulrümpfe in Kindmodule unterstrukturiert werden, und es können sogar andere offene Systeme importiert und benutzt werden. Dies unterstützt insbesondere eine sehr flexible interne Strukturierung offener Systeme.

7.2.3 Formaler Import offener Systeme

Um ein offenes System (oder genauer: seine externe Schnittstelle) in einer in Open-Estelle spezifizierten Umgebung nutzen zu können, muss das Interface mit der entsprechenden Deklaration des offenen Systems in diese Umgebungsspezifikation *formal importiert*[29] werden.

Der formale Import eines Interfaces wird syntaktisch durch ein IMPORT-STATEMENT (siehe Abb. 7-11) beschrieben, das Teil der Beschreibung einer Spezifikation, eines Modulrumpfes, eines Interfaces oder einer Behaviour-Definition ist (wir bezeichnen diese Kontexte als „*importierende Umgebungen*“). Ein IMPORT-STATEMENT besteht aus dem neuen Schlüsselwort „**IMPORT**“, gefolgt von einer Liste von Namen (siehe Abb. 7-11), welche *eindeutig* die zu importierenden Interfaces mit den entsprechenden Namen identifizieren.[30]

---

27. Zur Vermeidung von Irritationen durch die unterschiedlichen Schreibweisen im britischen („*behaviour*“) und amerikanischen Englisch („*behavior*“) unterstützt die um den Open-Estelle-Support erweiterte Version des Compiler-Frontends PET (und damit auch alle darauf aufbauenden Werkzeuge) beide Schreibweisen und behandelt sie als äquivalent (siehe Abschnitt 7.5). Die amerikanische Schreibweise ist jedoch nicht Teil der formalen Syntax von Open-Estelle.

28. d. h. sie haben den selben Namen und beziehen sich auf die selbe Headerdefinition

29. Zur sprachlichen Vereinfachung bezeichnen wir im Folgenden den „*formalen Import*“ im Zusammenhang mit Open-Estelle meist schlicht als „*Import*“. Erst in Abschnitt 7.4 kommen wir auf eine Spezialform der oben bereits diskutierten *textuellen Inklusion* zurück.

30. Die eindeutige Abbildung eines Interface-Namens auf ein Interface erfolgt auf der Metaebene, da diese Operation verschiedene textuelle Einheiten (z. B. Unix Textdateien) involviert. Es wird vorausgesetzt, dass diese Abbildung im mathematischen Sinne eindeutig erfolgt.

```
IMPORT-STATEMENT =                „IMPORT“ INTERFACE-IDENTIFIER { „,“ INTERFACE-IDENTIFIER } „;“ .
BODY-DEFINITION =                 [ IMPORT-STATEMENT ]
                                  DECLARATION-PART
                                  INITIALIZATION-PART
                                  TRANSITION-DECLARATION-PART .
QUALIFIED-IDENTIFIER =            INTERFACE-IDENTIFIER „::“ IDENTIFIER .
CONSTANT-IDENTIFIER =
TYPE-IDENTIFIER =
CHANNEL-IDENTIFIER =                    IDENTIFIER | QUALIFIED-IDENTIFIER .
HEADER-IDENTIFIER =
BODY-IDENTIFIER =
INTERFACE-IDENTIFIER =            IDENTIFIER .
```

Abbildung 7-11: Syntax des Imports von Interfaces

Der Import eines Interfaces definiert eine *qualifizierte Sichtbarkeit* aller Definitionen des Interfaces. Dies beinhaltet (neben Konstanten, Typen, Kanälen und Modulheadern) auch die im Interface deklarierten offenen Systeme. Auf diese Definitionen kann mittels ihres jeweiligen *qualifizierten Namens* (QUALIFIED-IDENTIFIER) Bezug genommen werden, welcher aus ihrem unqualifizierten Namen (gemäß ihrer Definition im Interface) besteht, dem jeweils der Name des definierenden Interfaces und das neue Symbol „`::`“ vorangestellt wird. So bezieht sich zum Beispiel in Abb. 7-12 der Name „`binaryService::binaryOperator`“ auf die Definition von „`binaryOperator`“ innerhalb des Interfaces „`binaryService`“.

```
SPECIFICATION test;
    IMPORT binaryService;
    MODVAR m: binaryService::binaryOperatorHeader;
    INITIALIZE
        BEGIN
            INIT m WITH binaryService::binaryOperator;
        END;
    { ... }
END.
```

Abbildung 7-12: Beispiel des Imports von Interfaces

Eine wesentliche Grundlage für die formale Eindeutigkeit von Open-Estelle bildet gerade dieses Import-Konzept: *Der formale Import eines Interfaces macht lediglich die dort gemachten Definitionen im importierenden Kontext qualifiziert sichtbar*. Die Interpretation der Definitionen in dem Interface erfolgt dagegen vollständig unabhängig von einem späteren Import und damit auch vom importierenden Kontext.[31] Weiterhin ist zu beachten, dass auf Grund der ein-

31. Diese syntaktische Fundierung ist mit Ausnahme der Bedeutung von unterschiedlichen Timescales vollständig kompatibel mit dem syntaktischen Modell von Standard-Estelle. Auf die Probleme durch unterschiedliche Timescales kommen wir in Abschnitt 7.4.3 zurück.

deutigen Identifikation eines Interfaces durch seinen (qualifizierenden) Namen und der eindeutigen Identifikation einer Definition innerhalb des Interfaces durch ihren (unqualifizierten) Namen jeder qualifizierte Name *global eindeutig* ist.

Eine Instanz des offenen Systems wird gebildet, indem bei der „Ausführung“ der Spezifikation eine neue Instanz der beschreibenden Moduldefinition mittels eines entsprechenden INIT-STATEMENTs erzeugt wird (siehe Abb. 7-12). Es ist dabei natürlich auch möglich, mehrere offene Systeme (also Modulinstanzen) dynamisch aus der selben offenen Systembeschreibung (also dem Modulheader zusammen mit dem Modulrumpf) zu erzeugen. Allgemein können alle Mechanismen, die Standard-Estelle zur Handhabung der Instanzen von lokal definierten Kindmodulen anbietet, auch auf offene Systeme angewendet werden. Dies schließt die Kommunikation mit dem offenen System und auch seine Termination mit ein.

Der Import eines Interfaces ist dabei nicht nur nützlich, um Zugriff auf das darin definierte offene System zu erlangen: Da der Import des Interfaces alle darin enthaltenen Definitionen in der importierenden Umgebung sichtbar macht, können Interfaces auch zur *globalen Definition* von Konstanten, Typen, Kanälen und Modulheadern eingesetzt werden, die dann in verschiedenen Spezifikationen, Behaviour-Definitionen oder sogar anderen Interfaces genutzt werden. Dieser Aspekt ist ganz besonders wichtig zur Trennung von unabhängigen Interfaces mit gemeinsamen Definitionen („*separation of concerns*“).

So deklarieren zum Beispiel in Abb. 7-13 die getrennten Interfaces `intC` und `intS` offene Systeme mit kompatiblen externen Interaktionspunkten. Dies ist möglich, da beide das Interface `intA` importieren, welches eine entsprechende Kanaldefinition enthält. Daraus ergibt sich ein System von Interfaces, das exakt die Typabhängigkeiten des globalen Systems modelliert.

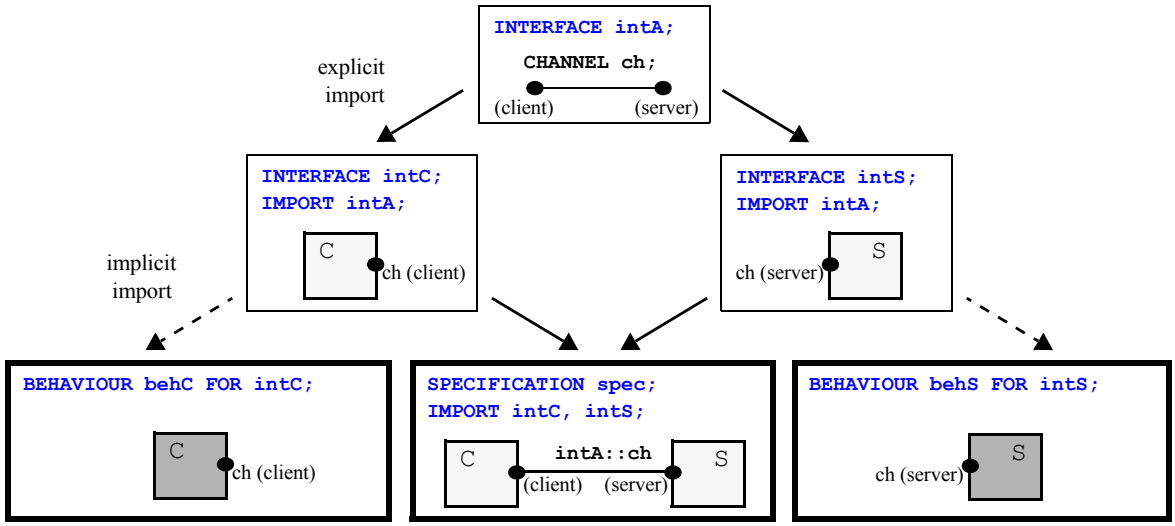


Abbildung 7-13: Separierung von Definitionen durch Interface- und Import-Hierarchien

7.2.4 Modulattributierungsregeln

Die in den Klauseln 5.2.1 und 7.3.6.2 von [ISO97] (siehe auch Anhang C) angegebenen Modulattributierungsregeln sind nicht ohne weiteres für die Anwendung in Open-Estelle geeignet, da sie sich ausschließlich auf die textuelle Verschachtelung von Modulen beziehen. Deshalb wurden für Open-Estelle *erweiterte Modulattributierungsregeln* entwickelt (siehe Anhang C.4), die

(i) beschränkt auf den Sprachumfang von Standard-Estelle vollständig (also *syntaktisch* und *semantisch*) *äquivalent* zu den Originalregeln sind,

(ii) für die Estelle-Erweiterung Open-Estelle *semantisch* (d.h. bezüglich des dynamischen Modulinstanz-Baums) das selbe Attributierungsschema bewirken wie die Originalregeln,

(iii) es ermöglichen, sowohl offene Systeme zu spezifizieren, die eines oder mehrere Subsysteme[32] enthalten (also in unattributierten Umgebungen eingesetzt werden können), als auch offene Systeme, die als Teil anderer Subsysteme eingesetzt werden können und

(iv) es insbesondere auch ermöglichen, offene Systeme zu spezifizieren, die *in beliebig attributierten Umgebungen* eingesetzt werden können.

Die ursprüngliche Definition der Modulattributierungsregeln in [ISO97] wurde auf heterogene Weise basierend auf direkter und indirekter textueller Modulverschachtelung definiert (siehe Anhang C). Diese Methode ist in Standard-Estelle anwendbar, da der Kontext[33] der Anwendung eines Moduls immer mit dem Kontext seiner Definition identisch war.

In Open-Estelle ist diese Vorbedingung nicht länger gültig, da es hier nunmehr drei verschiedene Kontexte für ein offenes System gibt:

- Das *externe Interface* (und damit auch die Attributierung) eines offenen Systems wird durch eine MODULE-BODY-DECLARATION in einer Interface-Definition spezifiziert.
- Das *Verhalten* eines offenen Systems wird durch eine MODULE-BODY-DEFINITION in einer Behaviour-Definition spezifiziert.
- Die *Anwendung* eines offenen Systems (also seine Instanziierung) wird innerhalb einer Modulrumpf-Definition oder einer Spezifikation durch Import des jeweiligen Interfaces und ein entsprechendes INIT-STATEMENT mit Bezug auf das offene System spezifiziert.

Offensichtlich sind die auf Basis strikter textueller Verschachtelung definierten Modulattributierungsregeln von Standard-Estelle nicht direkt auf Open-Estelle übertragbar. Zur Lösung dieses Problems wurde in mehreren Schritten ein neuer Satz von Modulattributierungsregeln formuliert, welche für die Integration der Open-Estelle-Erweiterung geeignet und (unter Beschränkung auf den Sprachumfang von Standard-Estelle) zu den bisherigen Regeln vollständig äquivalent sind.

Dazu wurden zunächst die bisherigen Regeln auf konsistente Art und Weise allein als Relationen zwischen einem Modul und seinem *Kontext* (dem unmittelbar übergeordneten Modul, „*closest containing module*“) definiert (siehe Tabelle 7.29). Dabei wird insbesondere auch die Nichtexistenz eines solchen übergeordneten Moduls im Falle des Spezifikationsmoduls mitberücksichtigt.

---

32. Mit dem Ausdruck „*Subsystem*“ bezeichnen wir Instanzen von Systemmodulen.

33. In diesem Zusammenhang bezeichnet der Ausdruck „*Kontext*“ die „*closest containing module-definition*“.

Tabelle 7.29: Modulattributierungsregeln von Standard-Estelle (reorganisiert)

Kontext („*closest containing module*“)	**Resultierende mögliche Modulattributierungen**				
	„system-activity“	„activity“	„system-process“	„process“	*nicht attributiert*
„systemactivity“		•			
„activity“		•			
„systemprocess“		•		•	
„process“		•		•	
nicht attributiert	•		•		•
existiert nicht[a]	•		•		•

a. Die Zeile „*existiert nicht*“ beschreibt die möglichen Attribute für Top-Level-Module, welche kein „*containing module*“ besitzen. Dies sind bei Standard-Estelle nur die Spezifikationsmodule, bei Open-Estelle kommen noch die Interface- und Behaviour-Definitionen dazu.

Im nächsten Schritt wurde ein neuer Satz von Modulattributierungsregeln formuliert, der diese Relation ausgehend von der Attributierung des jeweils eingeschachtelten Moduls formuliert.

Ausgehend von diesen Regeln wurden anschließend die syntaktischen Erweiterungen von Open-Estelle derart integriert, dass innerhalb einer INTERFACE-DEFINITION bzw. einer BEHAVIOUR-DEFINITION beliebig attributierte offene Systeme definiert werden können. Es ergeben sich damit die Regeln 1 bis 4 von Def. 7.11 auf Seite 332 (siehe auch Anhang C.4).

Nachdem wir bisher lediglich die statische (syntaktische) Verschachtelung von Modulattributierungen geregelt haben, müssen als nächstes Regeln zum Import unterschiedlich attributierter offener Systeme in unterschiedlich attributierte Umgebungen festgelegt werden.

Ziel der gesamten Attributierungsregeln in Standard-Estelle und Open-Estelle ist letztlich ja nicht die Durchsetzung eines Verschachtelungsschemas der Attribute von (statischen) Moduldefinitionen, sondern die Einhaltung eines solchen Schemas auf der (dynamischen) Hierarchie der Modulinstanzen.

Das Ziel der *erweiterten Modulattributierungsregeln* ist entsprechend, das von Standard-Estelle vorgegebene Attributierungsschema auf diesen Modulinstanzen, auf dem ja auch ganz wesentlich die Estelle-Semantik basiert, auch für Instanzen von Open-Estelle-Spezifikationen sicherzustellen. Dies bedeutet, dass durch den Import eines offenen Systems (genauer: den Import der INTERFACE-DEFINITION) und die Instanziierung des offenen Systems durch ein INIT-STATEMENT keine Hierarchie von Modulinstanzen entstehen darf, die bezüglich ihrer Attributierungen nicht auch in Standard-Estelle zulässig (bzw. möglich) gewesen wäre.

Ein denkbarer Ansatz zur Erreichung dieses Ziels wäre es, den Import einer Interface-Definition nur dann zu erlauben, wenn alle darin deklarierten offenen Systeme bezüglich ihrer Modulattributierungen auch in dem importierenden Kontext lokal hätten definiert werden können. Dies bedeutet zum Beispiel, dass in ein attributiertes Modul (als Kontext) kein Interface importiert werden kann, in welchem ein unattributiertes offenes System deklariert wird.

Dieser Ansatz führt zwar zum gewünschten Ziel, er bedeutet jedoch offensichtlich auch eine erhebliche Einschränkung bei der Zusammenstellung unterschiedlich attributierter offener Systeme in ein gemeinsames Interface. Daher wurde zur Erhöhung der Flexibilität beim Import offener Systeme in einem konkreten Kontext die oben genannte Anforderung auf die tatsächlich genutzten offenen Systeme beschränkt, indem nur solche Deklarationen offener Systeme aus ei-

nem importierten Interface die oben genannten Attributierungs-Anforderungen erfüllen müssen, die auch in dem importierenden Kontext in einem INIT-STATEMENT referenziert werden (siehe Regel 6 in Def. 7.11).

Zuletzt beinhalten die erweiterten Modulattributierungsregeln von Open-Estelle noch eine zusätzliche Regel, die es ermöglichen soll, offene Systeme zu definieren, die in *beliebig attributierten Umgebungen* importiert und instanziiert werden können. Die bisher vorgestellten Regeln machen dies unmöglich, da in einem unattributierten Kontext nur unattributierte oder SYSTEM-attributierte offene Systeme eingesetzt werden können, diese jedoch in keinem attributierten Kontext nutzbar sind. Die Grundidee dieser letzten Erweiterung besteht nun darin, die Unterscheidung zwischen SYSTEM-attributierten und nicht SYSTEM-attributierten offenen Systemen aufzuheben, indem letztere bei Bedarf (also beim Import in einem nicht attributierten Kontext) zu SYSTEM-attributierten offenen Systemen *umgedeutet* werden (siehe Regel 5 von Def. 7.11).

Es ergeben sich damit die in Def. 7.11 angegebenen Attributierungsregeln. In Anhang C.4 sind als Teil der formalen Definition von Open-Estelle noch weitere Erläuterungen zu den Regeln zu finden.

Definition 7.11: Modulattributierungsregeln von Open-Estelle (siehe auch Anhang C.4)

The following module attribution and nesting rules shall replace the rules given in Clauses 5.2.1 and 7.3.6.2 of [ISO97]:

1. Each active module shall be attributed.
2. A module[34] that is not attributed or attributed „**SYSTEMACTIVITY**“ or „**SYSTEMPROCESS**“ shall be a SPECIFICATION or be directly contained in an INTERFACE-DEFINITION, a BEHAVIOUR-DEFINITION or a not attributed module.
3. A module that is attributed „**ACTIVITY**“ shall be directly contained inside an INTERFACE-DEFINITION, a BEHAVIOUR-DEFINITION or an attributed module.
4. A module that is attributed „**PROCESS**“ shall be directly contained inside an INTERFACE-DEFINITION, a BEHAVIOUR-DEFINITION or a module that is attributed „**PROCESS**“ or „**SYSTEMPROCESS**“.
5. Within an importing environment that is a not attributed module, any imported MODULE-BODY-DECLARATION that is attributed „**PROCESS**“ or „**ACTIVITY**“ is handled like it was attributed „**SYSTEMPROCESS**“ (in case of „**PROCESS**“) or „**SYSTEMACTIVITY**“ (in case of „**ACTIVITY**“).
6. If the BODY-IDENTIFIER of an INIT-STATEMENT denotes an imported MODULE-BODY-DECLARATION, this MODULE-BODY-DECLARATION has to be attributed in such a manner that the module closest containing the INIT-STATEMENT could contain a child module with this attribution (according to the preceding attribution rules).

(Ende von Definition 7.11)

Diese automatische Umdeutung bedeutet dabei keine wirkliche Abschwächung der syntaktischen Präzision, da die **SYSTEM**-Attributierung auch in Standard-Estelle bereits redundant war und gegebenenfalls aus der Attributierung des Kontextes jederzeit rekonstruiert werden konnte (siehe auch Tabelle 7.29). Der beschriebene Umdeutungsmechanismus hat dabei insbesondere für die internen Definitionen des offenen Systems und seine interne Unterstrukturierung keine weiteren syntaktischen Nebenwirkungen und integriert sich ebenso vollständig in die Semantik des Gesamtsystems, wie wir im nächsten Abschnitt sehen werden.

34. With the term „module“ we refer to a SPECIFICATION or a MODULE-BODY-DEFINITION.

Die Regel ermöglicht es jedoch insbesondere, speziell ein **ACTIVITY**-attributiertes[35] offenes System in jeden beliebig attributierten Kontext[36] zu importieren und dort zu instanziieren (siehe Spalten „*system activity*“ und „*activity*“ in Tabelle 7.30) und bildet damit eine wichtige Grundlage für die flexible Nutzung offener Systeme in Estelle.

Tabelle 7.30: Modulattributierungsregeln von Open-Estelle

Kontext[a] **(„*closest containing module*“)**	**Resultierende mögliche Modulattributierungen**[b]				
	„system-activity“	„activity“	„system-process“	„process“	*nicht attributiert*
„systemactivity“		•			
„activity“		•			
„systemprocess“		•		•	
„process“		•		•	
nicht attributiert	•	←	•	←	•
existiert nicht[c]	•		•		•

a. Interface- und Behaviour-Definitionen sind selbst keine Moduldefinitionen und sind (im Sinne des „closest containing module“) in der „Kontext“-Spalte daher nicht repräsentiert. Sie können aber gemäß Def. 7.11 beliebig attributierte Moduldefinitionen bzw. Moduldeklarationen enthalten und bedingen selbst auch keine Uminterpretationen der Attributierungen.

b. Das Zeichen „←“ zeigt an, dass das entsprechende Modulattribut eines offenen Systems beim Import zulässig ist, dabei jedoch zu dem Attribut der links benachbarten Spalte umgedeutet wird.

c. Die Zeile „*existiert nicht*“ beschreibt die möglichen Attribute für Top-Level-Module (also das Spezifikationsmodul), welche kein „*containing module*“ besitzen.

35. Ein **ACTIVITY**-attributiertes Modul darf bereits unter den Standard-Estelle-Modulattributierungsregeln in einem **ACTIVITY**, **SYSTEMACTIVITY**, **PROCESS** oder **SYSTEMPROCESS** attributierten Kontext enthalten sein. Die beschriebene Umdeutungsregel behandelt das **ACTIVITY**-Attribut im Fall eines unattributierten Kontextes dann als **SYSTEMACTIVITY**-Attribut, was ebenfalls gemäß der Standard-Estelle-Modulattributierungsregeln zu einer zulässigen Modulhierarchie führt.

36. Ein „Import“ eines (wie auch immer attributierten) offenen Systems in eine nicht-existierende Umgebung (also die Umdeutung als Spezifikationsmodul) ist syntaktisch nicht vorgesehen und wäre auch von geringem Nutzen, da eine Instanziierung in einem ansonsten leeren und nichtattributierten Spezifikationsmodul letztlich den gleichen Effekt hätte. Entsprechend ist die unterste Zeile von Tabelle 7.30 für (nicht triviale) offene Systeme ohne Bedeutung.

7.3. Semantik von Open-Estelle

Als formale Beschreibungstechnik besitzt Estelle sowohl eine formale Syntax als auch eine *formale Semantik*. Entsprechend deckt die vorgestellte Spracherweiterung Open-Estelle ebenfalls beide Aspekte ab, um die Qualifizierung als formale Beschreibungstechnik zu erhalten.

Wir werden im Folgenden zunächst die bei der Entwicklung von Open-Estelle intendierte Kernsemantik der Anwendung eines offenen Systems in einem Estelle-Kontext diskutieren, wobei das resultierende Gesamtsystem offen oder geschlossen sein kann (siehe mittleres bzw. rechtes System in Abb. 7-14). Im letzteren Fall lässt sich die Semantik des Gesamtsystems sogar vollständig in die Standard-Estelle-Semantik abbilden, so dass keine semantischen Erweiterungen im eigentlichen Sinne erforderlich sind (siehe Abschnitt 7.3.1).

Eine Erweiterung der Estelle-Semantik ist dagegen für die Darstellung der Semantik eines offenen Systems für sich allein (also unabhängig von seiner Integration in eine konkrete Umgebung) erforderlich, da hier im Gegensatz zur Standard-Estelle-Semantik, bei der das betrachtete Gesamtsystem ja immer geschlossen ist, nun auch *Interaktionen mit einer nicht näher spezifizierten potentiellen Umwelt* mit in Betracht gezogen werden müssen. Wir werden die dazu erforderlichen kompatiblen Erweiterungen der Standard-Estelle-Semantik in Abschnitt 7.3.2 vorstellen.

Schließlich werden wir in Abschnitt 7.3.3 noch mögliche alternative Semantiken für Standard-Estelle und Open-Estelle diskutieren, die eine bessere Abstraktion der Schnittstelle und der Interaktionen zwischen offenen Systemen und möglichen Umgebungen liefern. Diese Semantiken erfordern jedoch eine grundlegende Neustrukturierung der Estelle-Semantik, die, wie wir sehen werden, nur sehr schwer kompatibel zur Standard-Estelle-Semantik gehalten werden kann.

7.3.1 Semantik des Imports offener Systeme

Ein zentrales Designziel von Open-Estelle war eine weitestmögliche Integration in das zu Grunde liegende (Standard) Estelle. Konzeptionell wurde dies mittels der Modellierung offener Systeme durch die bereits in Standard-Estelle existierenden Module realisiert.

Auf syntaktischer Ebene besteht die wesentliche Erweiterung von Open-Estelle in der Möglichkeit zur Definition solcher offener Systeme in separaten Interface- und Behaviour-Definitionen, wobei diesen externen Definitionen eine formale Syntax (d.h. eine eindeutige syntaktische Interpretation) zugeordnet wird, die insbesondere vollständig unabhängig vom späteren Import der Definitionen in andere Kontexte ist (siehe Abschnitt 7.2).

Durch diese Art der syntaktischen Fundierung lässt sich die Anwendung eines offenen Systems innerhalb eines importierenden Kontexts vollständig auf die in Standard-Estelle bereits vorgesehenen Abstraktionen zurückführen: Die Anwendung eines offenen Systems unterscheidet sich von seiner Instanziierung bis zu seiner Termination syntaktisch in nichts von der Anwendung eines einfachen Estelle-Moduls in Standard-Estelle.

Im Falle eines *geschlossenen Gesamtsystems* (siehe rechtes System in Abb. 7-14) ermöglicht die konzeptionelle Ununterscheidbarkeit importierter offener Systeme und textuell eingebetteter Moduldefinitionen, die Semantik des resultierenden Gesamtsystems auf die Standard-Estelle-Semantik abzubilden. Wir werden insbesondere später in Abschnitt 7.4 sehen, wie ein unter

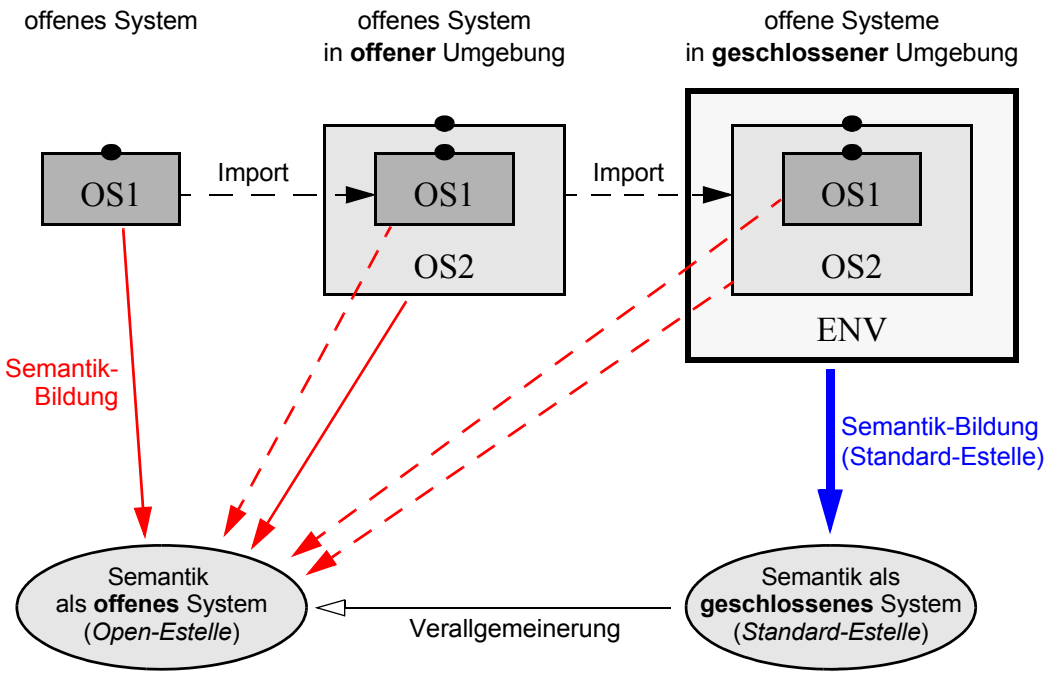


Abbildung 7-14: Offene Systeme in offenen oder geschlossenen Umgebungen

Einsatz von Open-Estelle-Erweiterungen spezifiziertes, geschlossenes Gesamtsystem durch eine syntaktische Transformation in eine äquivalente Standard-Estelle-Spezifikation umgewandelt werden kann.

Die Frage der Semantik des Imports eines offenen Systems in eine *offene Umgebung*[37] (siehe mittleres System in Abb. 7-14) lässt sich entsprechend ebenfalls vollständig auf syntaktischer Ebene auf die Frage nach der Semantik eines monolithisch (also ohne Import) spezifizierten Gesamtsystems zurückführen, wobei dieses nunmehr selbst offen ist (siehe linkes System in Abb. 7-14). Die Frage nach der Semantik eines solchen offenen Systems für sich (also ohne konkrete Umgebung) kann daher im nächsten Abschnitt unabhängig von der Frage seiner Unterstrukturierung in importierte Teilsysteme diskutiert werden. Wir werden später in Abschnitt 7.3.3 auf die Frage des Zusammenwirkens der Semantiken offener Systeme für sich alleine und importiert in eine konkrete Umgebung nochmals zurückkommen.

7.3.2 Teilsystemsemantik von Open-Estelle

In diesem Abschnitt stellen wir die *semantische* Erweiterung von Open-Estelle vor, mit deren Hilfe einem offenen System für sich allein, also unabhängig von einer konkreten Umgebung, eine Semantik zugeordnet werden kann. Diese Semantik wurde kompatibel zur Standard-Estelle-Semantik entwickelt und basiert auf der Idee, die Semantik eines offenen Systems zunächst analog zu der eines in Standard-Estelle spezifizierten (d.h. geschlossenen) Systems zu definieren, um darauf aufbauend alle möglichen Interaktionen über die durch Open-Estelle hinzugefügten externen Schnittstellen zu berücksichtigen. Diese Interaktionen bestehen gemäß der ver-

37. also eine Umgebung, die selbst ein offenes System ist

fügbaren Schnittstellenkomponenten offener Systeme aus dem Versenden bzw. Empfang von asynchronen Nachrichten und der Manipulation von exportierten Variablen durch das offene System oder seine Umgebung.

Die Berücksichtigung aller möglichen Interaktionen durch die externen Schnittstellen des offenen Systems modelliert dabei die Integration in alle möglichen Umgebungen und deren Verhalten.[38] Dieses *potentielle Verhalten* wird dann reduziert, sobald das offene System in eine *konkrete Umgebung* integriert wird.

Diese Reduktion manifestiert sich aufgrund der Definition der Estelle-Semantik (und damit auch der hier definierten Open-Estelle-Semantik) in Form einer *Menge zulässiger Berechnungen* durch eine *Teilmengenbeziehung* zwischen

(i) den möglichen Berechnungen des offenen Systems für sich alleine und

(ii) im Zusammenwirken mit einer konkreten Umgebung.

Diese Systemsicht ist konzeptionell analog zu der Semantik des Teilsystems als (syntaktischer) Teil eines Standard-Estelle-Systems, wie es z. B. durch den oben beschriebenen formalen Import[39] in eine konkrete Estelle-Umgebung entstehen kann. Betrachtet man in der Semantik des resultierenden Gesamtsystems (also in seinen Berechnungen) lediglich den Anteil des[40] vom offenen System gebildeten Modulinstanz-Teilbaums, so erhält man gerade die Menge der Berechnungen des Teilsystems im Zusammenwirken mit dieser konkreten Umgebung.[41]

Diese Sichtweise liefert auf einfache Weise einen semantisch fundierten Korrektheitsbegriff für die hier vorgestellte Semantik offener Systeme in Relation zur oben vorgestellten syntaktisch fundierten: Die Semantik eines offenen Systems ergibt sich aus der Vereinigung aller Semantiken (Berechnungen) der Integration des offenen Systems in konkrete Umgebungen. Dabei ist es letztlich unerheblich, ob das betrachtete Gesamtsystem, auf dessen Semantik sich die hier vorgestellte Semantikdefinition letztlich stützt, selbst offen oder geschlossen ist: Da die Verschachtelung des Imports offener Systeme endliche Tiefe[42] hat, kann über Strukturinduktion die semantische Fundierung des Korrektheitsbegriffs gezeigt werden. Insbesondere kann letztlich ein geschlossenes System immer als Spezialfall eines offenen Systems betrachtet werden, und wir werden sehen, dass in diesem Spezialfall die vorgestellte Semantik offener Systeme mit der Standard-Estelle-Semantik zusammenfällt. Die Betrachtung geschlossener Systeme ist an dieser Stelle jedoch für die Fundierung der Semantik offener Systeme unerlässlich.

---

38. Insbesondere ist es möglich, zu einem konkreten Interface eine Umgebung zu spezifizieren, die potentiell jede durch diese Schnittstelle mögliche Interaktion auch potentiell durchführen kann. Dies entspricht insbesondere der Idee des „*Universal Test Drivers Generator*“ [LaLeMa91].

39. bzw. aus Systemsicht durch die Instanziierung der importierten Systembeschreibung

40. genauer: *einen* der Modulinstanz-Teilbäume, da ja das offene System beim syntaktischen Import wie ein gewöhnliches Modul auch mehrmals instanziiert werden kann

41. Die oben zur Flexibilisierung der Beschreibungstechnik eingeführte Umdeutung der Attributierung eines offenen Systems abhängig von der importierenden Umgebung beeinträchtigt diesen Übergang. Alternativ kann man jedoch stattdessen die aus der Anwendung bzw. Nichtanwendung dieser Umdeutung resultierenden Systeme als verschiedene (offene) Spezifikationen deuten.

42. was bei endlichen Beschreibungen durch das Verbot direkt oder indirekt rekursiver Importe effektiv sichergestellt wird (siehe Bemerkungen in Anhang C.3.2)

7.3.2.1 Die Standard-Estelle-Semantik

Der Estelle-Standard [ISO97] definiert die Semantik einer (geschlossenen) Spezifikation *SP* formal durch eine Menge von „*Berechnungen*“ („*computations*“). Berechnungen sind Sequenzen $<sit_0, sit_1,....>$ so genannter „*globaler Situationen*“ („*global situations*“), wobei sit_0 „*initial*“ ist und für alle $j>0$, sit_j eine mögliche „*nächste globale Situation*“ („*next global situation*“) von sit_{j-1} bezüglich der „*next-state*“ Relation ist. Im Wesentlichen beinhaltet eine globale Situation $sit = (gid_{SP}; A_1,...,A_n)$ die lokalen Zustände aller Modulinstanzen (gid_{SP}: Zustand der Eingangswarteschlangen, Werte lokaler Variablen, Modulinstanzstruktur, Verbindungsstruktur) und die Mengen $A_1,...,A_n$ der jeweils in den Subsystemen $S_1,...,S_n$ zum Feuern ausgewählten Transitionen. Somit definiert eine Estelle-Spezifikation ein *Transitionssystem* (S, δ), wobei S und δ sich auf die Menge globaler Situationen bzw. die next-state Relation beziehen. Nebenläufigkeit wird durch Interleaving modelliert.

Wir werden nun diese Semantik ergänzen, um die erweiterte Ausdrucksfähigkeit von Open-Estelle, offene Systeme beschreiben zu können, abzudecken. Die resultierende Semantik von Open-Estelle wird dabei schließlich die Standard-Estelle-Semantik als Spezialfall für die Systeme beinhalten, die keine externen Interaktionspunkte oder exportierte Variablen besitzen.[43]

Dazu betrachten wir zunächst die Definition der Relation „*next global situation*“ und der darauf aufbauend definierten Berechnungen („computations“) für eine Spezifikation *SP* gemäß der Standard-Estelle-Semantik:

Definition 7.12: (*next global situations*, *computation*; Clause 5.3.4, [ISO97]):

Given a global situation, $sit = (gid_{SP}; A_1,...,A_n)$, the set of next global situations is described as follows:

(a) For every $i = 1,...,n$: if $A_i = \varnothing$, then for every $AS(gid_{SP}/S_i) \in AS^*(gid_{SP}/S_i)$, $(gid_{SP}; A_1,...,AS(gid_{SP}/S_i),...,A_n)$ is a next global situation of sit.

(b) For every $i = 1,...,n$: if $A_i \neq \varnothing$, then for every $t \in A_i$, $(t(gid_{SP}); A_1,...,A_i\backslash\{t\},...,A_n)$ is a next global situation of sit.

NOTE — There are as many next global situations of the situation sit as there are possible choices of next transition t (and its results) in case (b), and different empty sets A_i in sit, for the case (a). In addition, in case (a), all possible choices of the set *AS* resulting from non-determinism of each component process in the system rooted at S_i must be taken into account.

NOTE — [...]

(Ende von Definition 7.12)

Eine Folge $sit_0, sit_1, ..., sit_j, ...$ von globalen Situationen von *SP* ist also genau dann eine Berechnung von *SP*, wenn sit_0 initial ist und für alle $j>0$ gilt, dass sit_j eine nächste globale Situation von sit_{j-1} ist, die sich aus einer der obigen Regeln (a) oder (b) ergibt.

---

43. Wir gehen bei der Definition der Semantik davon aus, dass zur Fundierung ein oder mehrere Subsysteme existieren. Gehen diese nicht aus entsprechenden Systemmodulen im offenen System (oder aus dem offenen System selbst) hervor, so kann es alternativ durch Umdeutung der Attributierung des offenen Systems oder durch Annahme der Bereitstellung von mindestens einem Subsystem durch eine (genauer: jede bzgl. der Attributierung kompatible) Umgebung erfolgen. Im letzteren Fall betrachten wir ausschließlich die Aktivitäten des offenen Systems in seinem Subsystem.

Dabei wird für jedes Subsystem S_i (repräsentiert durch ein Systemmodul) entweder (a) eine neue Menge schaltbarer Transitionen A_i ausgewählt[44] oder (b) eine bereits ausgewählte Transition $t \in A_i$ geschaltet[45]. Wenn eine Transition t einer Modulinstanz P geschaltet wird, werden als globale Schaltwirkung die Ausgaben von t in die Zielwarteschlangen übertragen (definiert in $transmission_P(gid'_{SP})$ mit $gid'_{SP} \in [t]_P(gid_{SP})$; siehe Klausel 9.5.4, [ISO97]). Wenn für eine Ausgabe keine Zielwarteschlange definiert ist, wird die Nachricht verworfen. Es ist dabei zu beachten, dass alle Zielwarteschlangen in gid'_{SP} repräsentiert sind und daher alle globalen Effekte direkt anwendbar sind. Als weitere (lokale) Schaltwirkung können in $t(gid_{SP})$ auch exportierte Variablen des betroffenen Moduls modifiziert werden.

7.3.2.2 Erweiterung für offene Systeme

Um nun diese Definition der *„next global situations“* und der darauf aufbauenden Berechnungen („*computations*“) für Open-Estelle-Spezifikationen verallgemeinern zu können, müssen wir nun die Interaktion eines offenen Systems mit seiner Umgebung modellieren. Diese Interaktionen bestehen aus dem Empfang von Nachrichten von der Umgebung, dem Verschicken von Nachrichten an diese und der Modifikation der exportierten Variablen durch die Umgebung. Da jedoch bei der hier definierten Teilsystemsemantik die Umgebung und damit ihr Verhalten noch nicht festgelegt sind, müssen *alle möglichen Umgebungen* mit allen möglichen empfangenen bzw. versendeten Nachrichten und alle möglichen Modifikationen exportierter Variablen berücksichtigt werden.[46]

Zunächst betrachten wir nun die Erweiterung der Relation „*next global situation*“ aus Def. 7.12 um den Empfang von *Nachrichten aus der* und die Modifikation von *exportierten Variablen durch die Umgebung*. Mit der Repräsentation des Versendens von Nachrichten an die Umgebung werden wir uns später befassen.

Alle Interaktionen mit der Umgebung geschehen durch externe Interaktionspunkte und exportierte Variablen. Da die Interaktionspunkte und die exportierten Variablen getypt sind, ist die Menge der Interaktionen, die von einer Modulinstanz P von ihrer Umgebung durch einen externen Interaktionspunkt ip empfangen werden können ($receive_P(ip)$, siehe Klauseln 9.3.1 und 9.4.3, [ISO97]), sowie die Menge der Werte, die einer exportierten Variablen e durch die Umgebung zugewiesen werden können, determiniert.[47] Um nun alle möglichen Umgebungen in Betracht zu ziehen, modellieren wir alle möglichen Empfangsereignisse und Zuweisungen

---

44. Dazu wird die leere Menge A_i (Menge ausgewählter Transitionen des Subsystems S_i) durch das Ergebnis einer Transitionsauswahl in diesem Subsystem $AS(gid_{SP}/S_i)$ ersetzt.

45. Dazu wird eine Transition t aus der Menge A_i entnommen und der globale Systemzustand gid_{SP} durch die Wirkung der Transition $t(gid_{SP})$ ersetzt.

46. Es ist zu beachten, dass die Auswirkungen der Anwendung der Statements „**terminate**“ oder „**release**“ auf ein offenes System keine Auswirkungen auf die Menge seiner Berechnungen hat. Wir berücksichtigen an dieser Stelle zur Vereinfachung auch nicht die Folgen der Anwendung von „**attach**“ oder „**detach**“ Statements durch die Umgebung mit (direkter oder indirekter) Auswirkung auf externe Interaktionspunkte offener Systeme, da die daraus resultierenden möglichen Operationen auf den beteiligten Warteschlangen sehr spezifisch für Standard-Estelle mit seiner monolithischen Semantikdefinition sind und aus konzeptioneller Sicht bzgl. eines offenen Systems kaum als sinnvolle Schnittstellenoperation zu motivieren sind. Analog betrachten wir an dieser Stelle zur Vereinfachung auch nicht die Auswirkungen einer Verbindung zweier externer Interaktionspunkte der selben Instanz eines offenen Systems durch eine Umgebung (s. u.).

durch eine Erweiterung der *„next global situation“*-Relation, wie sie in Def. 7.13 dargestellt ist. Weiterhin berücksichtigen wir dabei globale Situationen bezogen auf beliebige Modulinstanzen *P* anstatt nur auf Spezifikationsinstanzen *SP*.

Definition 7.13: (*next global situations*, ***potential*** *computation*; Open-Estelle):

Given a global situation, $sit_P = (gid_P; A_1,...,A_n)$, of a module instance P, the set of next global situations is described as follows:

(a) For every $i = 1,...,n$: if $A_i = \varnothing$, then for every $AS(gid_P/S_i) \in AS^*(gid_P/S_i)$: $(gid_P; A_1,...,AS(gid_P/S_i),...,A_n)$ is a next global situation of sit_P.

(b) For every $i = 1,...,n$: if $A_i \neq \varnothing$, then for every $t \in A_i$: $(t(gid_P); A_1,...,A_i\backslash\{t\},...,A_n)$ is a next global situation of sit_P.

(c) *For every $gid_P' \in env\_mod^+(gid_P)$: $(gid_P'; A_1,...,A_n)$ is a next global situation of sit_P. env_mod is defined as follows[48]:*

(c1) *For every $ip \in EIP_P$: for every $<m,v_1,...,v_k> \in receive_P(ip)$: $received_P(gid_P, <m,v_1,...,v_k>) \in env\_mod(gid_P)$.*

(c2) *For every $e \in EV\text{-}id_M$, where $P \in INST(M,B,E)$, and e is of type T: for every $v \in E(T)$: $assign_P(gid_P, e, v) \in env\_mod(gid_P)$.*

NOTE — $sit_P = (gid_P; A_1,...,A_n)$ is the global situation of module instance *P*, where gid_P is defined as usual, and each A_i is a set of transitions of the component instances rooted at (α) *P*, if *P* is attributed, or (β) S_i, if *P* is not attributed, and S_i are system modules. This generalizes the definition of global situations to arbitrary module instances. If *P* = *SP*, the definition of sit_P is identical to that of *sit* in the Estelle standard.

NOTE — There are as many next global situations of the situation sit_P as there are possible choices of (a) different empty sets A_i in sit_P and possible choices of the set *AS* resulting from non-determinism of each component process in the system rooted at S_i, (b) next transitions *t* (and their results), (c) inputs from the environment and assignments to exported variables of *P* by the environment.

NOTE — For closed systems (i.e. module instances that have no external interaction points and no exported variables) Clause (c) of the definition above has no effects. Consequently, the ‘*next global situations*’ relation given above reduces to the one of Def. 7.12.

NOTE — EIP_P is the set of external interaction points of instance *P* (Clause 9.4.3, [ISO97]). If *P* = *SP*, then EIP_P is empty, and no inputs will be received from the environment. $EV\text{-}id_M$ is the set of exported variables of *P* as declared in its module header *M* (see Clause 9.4.1, [ISO97]). If *P* = *SP*, then no exported variables are defined. If EIP_P and $EV\text{-}id_M$ are both empty, Def. 7.13 is equivalent to Def. 7.12 of standard Estelle.

---

47. Zusammen mit der *Containertyp-Erweiterung* (siehe Abschnitt 5.2.2) können eingebettet in einen Containertyp beliebige, aber wohldefinierte Datentypen übergeben werden (die enthaltene Typstruktur ergibt sich eindeutig aus der jeweiligen Konstruktion der Daten). Es ist jedoch zu beachten, dass dabei auch rekursive Datenstrukturen aufgebaut werden können. Aufgrund der Endlichkeit der Gesamtspezifikation ist die Anzahl der dort jeweils definierbaren Strukturtypen jedoch endlich. Kritisch ist hier somit lediglich die in endlicher Zeit (also jeweils bis zu einem gegebenen Zeitpunkt der Berechnung) erreichbare Strukturgröße, die unbegrenzt, aber endlich sein muss. Das Estelle-Zeitmodell liefert hier keine hinreichenden Aussagen, jedoch erscheint diese Annahme sinnvoll.

48. Die Funktion $env\_mod: EIP_P \rightarrow \wp(EIP_P)$ definiert eine Relation $env\_mod_R = \{(x,y) \in EIP_P \times EIP_P \mid y \in env\_mod(x)\}$.

    Sei $env\_mod_R^+$ der transitive Abschluss von env_mod_R. Dann definiert die Funktion $\boldsymbol{env\_mod}^+: EIP_P \rightarrow \wp(EIP_P),\ x \rightarrow \{y \in EIP_P \mid (x,y) \in env\_mod_R^+\}$ einen *transitiven Abschluss* der Funktion *env_mod*, d.h. das Ergebnis von einer oder mehrerer aufeinanderfolgender Ereignisse gemäß (c1) und/oder (c2).

NOTE — $receive_P(ip)$ is the subset of Interactions (Clause 9.3.1, [ISO97]) that the instance P can receive through interaction point ip (Clause 9.4.3, [ISO97]). $received_P(gid_P)$ is obtained from gid_P by replacing *s'.ie(ip')* by *append(<ip',ip,m,*v_1*, ..., *v_k*>, s'.ie(ip'))*, where *downattach(ip) = ip', ip'* in $EIP_{P'}$, and s' is the local state of *P'* (see Clause 9.5.4, [ISO97]). This includes the special case that ip is not attached, i.e. *downattach(ip) = ip*.

NOTE — $assign_P(gid_P,e, v)$ is a new gid of *P* where the difference with gid_P is expressed by $s.Loc(alloc_B(e)) := v$, where s is the local state of *P* (see Clause 9.5.4, [ISO97]).

(Ende von Definition 7.13)

Eine Folge <sit_0, sit_1, ..., sit_j, ...> von globalen Situationen von *P* heißt genau dann eine „*potentielle Berechnung*“ („*potential computation*“) von *P*, wenn für jedes $j>0$ gilt, dass sit_j eine nächste globale Situation von sit_{j-1} gemäß (a), (b) oder (c) von Def. 7.13 ist.

Im Vergleich zur Standard-Estelle-Semantik ergeben sich zwei wesentliche Unterschiede: Der erste Unterschied betrifft die Definition globaler Situationen sit_P für beliebige Modulinstanzen *P*, anstatt nur für die spezielle Modulinstanz *SP* wie in Def. 7.12. Um die Ausführung eines offenen Systems zu modellieren, verallgemeinern wir die Notation der globalen Situation durch die Kombination von gid_P mit Mengen von zur Ausführung ausgewählten Transitionen.

Der zweite Unterschied betrifft den *Empfang* von Nachrichten von der Umgebung und die Zuweisung von Werten an exportierte Variablen der Modulinstanz *P*. Diese werden durch den transitiven Abschluss env_mod^+ der Funktion *env_mod* in (c) von Def. 7.13 modelliert. Nachrichten können (ausschließlich) durch externe Interaktionspunkte von *P* ($ip \in EIP_P$) empfangen werden, wie in (c1) ausgedrückt. Es ist dabei zu beachten, dass env_mod über alle $ip \in EIP_P$ und alle $<m,v_1,...,v_k> \in receive_P(ip)$ geht. Somit werden alle möglichen Umgebungen berücksichtigt. Wenn ein Attachment[49] zu einem externen Interaktionspunkt von P besteht, wird jede Nachricht an die Zielwarteschlange am Ende der Attach-Kette geleitet.

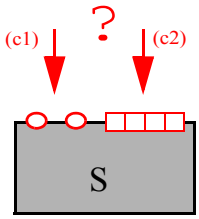
Zuweisungen zu exportierten Variablen von *P* durch die Umgebung werden durch (c2) ausgedrückt. Dabei geht *env_mod* über alle $e \in EV\text{-}id_M$ und alle $v \in E(T)$, für *e* vom Typ *T*. Damit werden auch hier alle möglichen Umgebungen berücksichtigt.

Da das in der Definition der nächsten globalen Situation beschriebene Verhalten allgemein nur auftritt, wenn das offene System in eine (konkrete) Umgebung integriert ist, benutzen wir den Ausdruck „*potentielle Berechnung*“ („*potential computation*“) von *P* für eine Folge von globalen Situationen, die die o. g. Bedingungen erfüllen. Der Grund dafür ist, dass das in (c) ausgedrückte Verhalten der Umgebung noch nicht existiert. Erst sobald die Umgebung vollständig vorliegt, wird die Menge der *potentiellen Berechnungen* reduziert auf eine Menge von *Berechnungen* im Sinne von Def. 7.12 [ISO97]. Dies ist im Falle eines geschlossenen Systems (also eines offenen Systems mit leerer externer Schnittstelle) unmittelbar der Fall. Hier werden die nächsten globalen Situationen vollständig durch (a) und (b) beschrieben, was letztlich der ursprünglichen Semantikdefinition von Standard-Estelle entspricht (s. o.) und somit Standard-Estelle insbesondere durch die Darstellung eines geschlossenen Systems als Spezifikation (dem Start-Nichtterminal der Standard-Estelle-Syntax) syntaktisch wie auch semantisch zu einem Spezialfall von Open-Estelle macht.

Nachdem wir uns mit dem Empfang von Nachrichten aus der Umgebung und der Modifikation exportierter Variablen beschäftigt haben, untersuchen wir nun das Verschicken von Nachrichten aus dem offenen System an die Umgebung. In der Standard-Estelle-Semantik werden alle

49. eine Verknüpfung zwischen Interaktionspunkten, die durch Ausführung eines `attach`-Statements entsteht

Ausgaben einer Transition *t* von *P* als Ergebnis der lokalen Wirkung (definiert durch $[t]_P(s)$, siehe Clause 9.6.6.5, [ISO97]) in der lokalen Zustandskomponente *s.out* gesammelt. Die Funktion $transmission_P$ definiert den globalen Effekt durch reihenfolgeerhaltendes Anhängen der Elemente aus der Folge *s.out* in die jeweiligen Zielwarteschlangen (Klausel 9.5.4, [ISO97]). Falls es keine solche Zielwarteschlange für eine Nachricht gibt (also $linked(ip,gid_{SP})$ nicht erfüllt ist), wird die Nachricht verworfen (Clauses 9.5.3 und 9.5.4, [ISO97]).



Definition (*next global situations*, *potential computation*; Open-Estelle):

Given a global situation, $sit_P = (gid_P; A_1,...,A_n)$, of a module instance *P*, the *set of next global situations* is described as follows:

select transitions

(a) For every $i = 1,...,n$: if $A_i = \varnothing$, then for every $AS(gid_P/S_i) \in AS^*(gid_P/S_i)$: $(gid_P; A_1,...,AS(gid_P/S_i),...,A_n)$ is a next global situation of sit_P.

fire transition

(b) For every $i = 1,...,n$: if $A_i \neq \varnothing$, then for every $t \in A_i$: $(t(gid_P); A_1,...,A_i\backslash\{t\},...,A_n)$ is a next global situation of sit_P.

external interaction

(c) For every $gid_P' \in$ ***env_mod***$^+(gid_P)$: $(gid_P'; A_1,...,A_n)$ is a next global situation of sit_P. ***env_mod*** is defined as follows:

(c1) For every $ip \in EIP_P$: for every $<m,v_1,...,v_k> \in receive_P(ip)$: $received_P(gid_P, <m,v_1,...,v_k>) \in env\_mod(gid_P)$.

(c2) For every $e \in EV\text{-}id_M$, where $P \in INST(M,B,E)$, and *e* is of type *T*: for *every* $v \in E(T)$: $assign_P(gid_P, e, v) \in env\_mod(gid_P)$.

Abbildung 7-15: Erweiterung der Estelle-Semantik für offene Teilsysteme (S)

Die Behandlung von Übertragungen in Open-Estelle folgt direkt der Standard-Estelle-Semantik. Wenn die Zielwarteschlange einer Nachricht zum offenen System gehört, ist die Semantik der Übertragung identisch. Wenn jedoch die Zielwarteschlange außerhalb des offenen Systems liegt, also eine Ausgabe (direkt oder indirekt[50]) über einen externen Interaktionspunkt von P erfolgt, verlässt die Nachricht das offene System.[51] Dies bedeutet, dass Ausgaben an die Umgebung keine Zustandsraummanipulation im offenen System bewirken. Übertragungen an die Umgebung werden entsprechend in der hier vorgestellten Teilsystemsemantik von Open-Estel-

50. d.h. der externe Interaktionspunkt ist Teil einer Attach-Kette, über die die Übertragung erfolgt

51. Wie oben bereits erwähnt betrachten wir hier zur Vereinfachung nicht die Effekte von zwei durch die Umgebung verbundenen externen Interaktionspunkten des selben offenen Systems. Diese Konstellation führt letztlich dazu, dass die Zielwarteschlange für eine darüber übertragene Nachricht wiederum innerhalb des offenen Systems liegt und entsprechend berücksichtigt werden muss.

    Dies ist in Def. 7.13 nur über eine Kombination der Fälle (b) (d.h. das offene System emittiert eine Nachricht nach außen) und (c) (d.h. das offene System empfängt eine beliebige Nachricht, konkret hier identisch zur zuvor emittierten und von der Umgebung zurückgeleiteten) möglich. Die geforderte Schaltwirkung ist somit zwar in ihrem transitiven Abschluss, nicht jedoch „*next global situation*“ selbst enthalten.

    Es wäre an dieser Stelle jedoch denkbar, zusätzlich alle bzgl. ihrer Kompatibilität zulässigen Verbindungen externer Interaktionspunktpaare des offenen Systems in „*next global situation*“ zu berücksichtigen, z. B. durch eine Erweiterung der Transitionswirkung $t(gid_P)$.

le modelliert, indem die resultierenden Nachrichten schlicht verworfen werden. Da dies durch die Definitionen in [ISO97] (siehe Clauses 9.5.3 an 9.5.4) bereits entsprechend behandelt wird, ist an dieser Stelle keine Anpassung der Semantik erforderlich[52].

7.3.3 Ausblick: Alternative Semantikansätze

Wir haben in den vorangegangenen Abschnitten jeweils zur Standard-Estelle-Semantik kompatible Semantiken offener Systeme diskutiert, die

- Teil eines Gesamtsystems sind (Abschnitt 7.3.1) oder
- ein offenes Teilsystem mit einer (noch) nicht konkretisierten Umgebung sind (Abschnitt 7.3.2).

Die strenge Kompatibilität dieser Ansätze zur Standard-Estelle-Semantik ist eine wichtige Grundlage zum nahtlosen Übergang von Estelle zur vorgestellten Erweiterung Open-Estelle.

Es zeigen sich an dieser Stelle jedoch auch Nachteile der Übertragung der Standard-Estelle-Semantik auf Open-Estelle. So basiert die Semantik eines Teilsystems (im Sinne eines Modulinstanz-Teilbaums) innerhalb eines Estelle-Systems immer auf dem Zustandsraum aller rekursiv enthaltenen Teilsysteme.

Dies ist letztlich auch die Ursache dafür, dass

(i) es zwar wie in Abschnitt 7.3.2 gezeigt möglich ist, die Semantik eines („*nach oben*“)[53] *offenen Teilsystems* unabhängig von einer konkreten Umgebung anzugeben,

(ii) eine Semantik einer entsprechenden *Umgebung* unabhängig von einem konkreten (z. B. per Open-Estelle importierten) offenen Teilsystem nicht, bzw. nur sehr schwer angegeben werden kann, da sie bzgl. der Modulinstanzhierarchie an dieser Stelle „*nach unten*“ offen ist und somit einen unvollständigen[54] Modulinstanz-Teilbaum hat, also in gid_P kein der Estelle-Semantik zu Grunde liegender Systemzustand dieses Teilbaums angegeben werden kann.

Diese Asymmetrie ist umso bedauerlicher, als die Syntax von Open-Estelle durch die Separierung der Interface Definition eines offenen Systems eine geradezu *symmetrische* Trennung der Definitionen offener Systeme (genauer: Behaviour Definitionen) und importierender Umgebungen bietet (siehe Abb. 7-16). Dabei bildet die Interface Definition den einzigen (syntaktischen) Berührungspunkt beider Welten. Entsprechend sind nicht nur die möglichen Umgebun-

---

52. Genauer müssen die Funktionen $sent_P$, $received_P$, und $transmission_P$ auf gid_P (anstatt gid_{SP}) angewandt werden, wobei *P'* die Wurzelmodulinstanz des offenen Systems ist (siehe Clause 9.5.4, [ISO97]).

53. Lage der Schnittstelle bzgl. ihrer Ausrichtung in der Modulinstanzhierarchie nach „*oben*“ (also zur Vatermodulinstanz) oder nach „*unten*“ (also zu einer Kindmodulinstanz bzw. dem davon ausgehenden Modulinstanz-Teilbaum)

54. Aus Sicht des „nach unten“ offenen Systems ist nur bekannt, dass dieser Modulinstanz-Teilbaum existiert, da das (durch ein Interface deklarierte aber noch nicht definierte) offene System von seinem Vatermodul instanziiert wurde. Somit kann Zugriff auf seine Schnittstellenkomponenten genommen werden. Über sein Verhalten können jedoch im Rahmen der durch seine äußere Schnittstelle festgelegten Möglichkeiten keine Aussagen gemacht werden. Dies ist letztlich analog zur Situation eines „nach oben“ offenen Systems (siehe Abschnitt 7.1.4).

gen zu einem konkreten offenen System austauschbar (*c1* oder *c2*), sondern auch die offenen Systeme zu einer konkreten Umgebung (*b1* oder *b2*). Die einzige syntaktische Gemeinsamkeit bleibt lediglich das externe Interface (*a*).

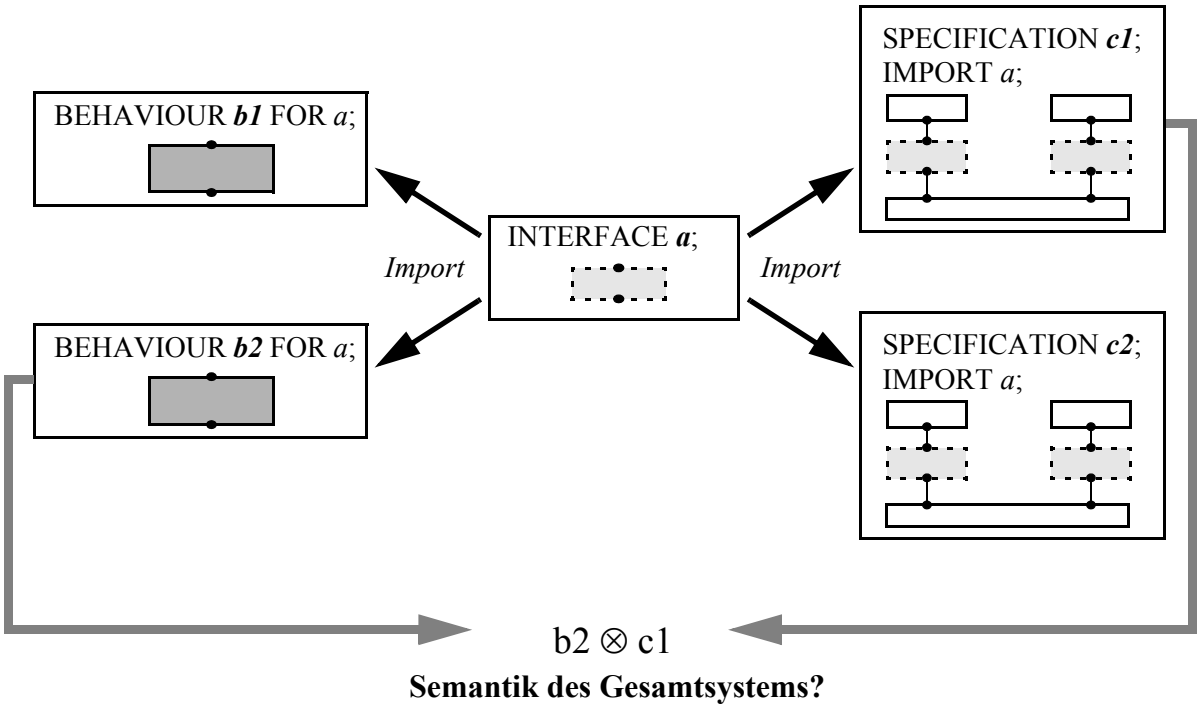


Abbildung 7-16: Symmetrie zwischen offenem System und importierender Umgebung

Eine Semantik, die diese symmetrische Austauschbarkeit angemessen repräsentiert, sollte dabei folgende Anforderungen erfüllen:

- Ableitung der Semantik eines offenen Teilsystems (*unabhängig von einer konkreten Umgebung*),
- Ableitung der Semantik einer importierenden Umgebung (*unabhängig von einem konkreten importierten offenen System*),
- Ableitung der Semantik eines Gesamtsystems[55] (*allein aus den jeweiligen Semantiken der beteiligten Teilsysteme*).

Gerade der letzte Punkt bildet dabei eine wesentliche Grundlage für die Entwicklung von effektiv nutzbaren Semantiken offener Systeme, da nur so auf konstruktive Weise die Semantik des Gesamtsystems auf den Semantiken der Komponenten beruht und somit eine Parallelentwicklung zweier (womöglich auch noch nicht voll kompatibler) Semantiken für ein offenes System für sich alleine und integriert in eine konkrete Umgebung vermieden werden kann.

55. z. B. aus einem *konkreten* offenen System und einer *konkreten* importierenden Umgebung (siehe $b2 \otimes c1$ in Abb. 7-16).

7.3.3.1 Integration von Teilsystemsemantiken

In diesem Abschnitt untersuchen wir die Möglichkeiten zur Integration von Teilsystemsemantiken in die Semantik eines Gesamtsystems zunächst unabhängig von einer konkreten FDT. Wir gehen dazu von einer Funktion $sem(x)$ aus, die jedem syntaktisch wohlgeformten Teil- oder Gesamtsystem eine noch nicht näher spezifizierte Semantik zuordnet, die jedoch einen semantischen Kompositionsbegriff unterstützt, welcher aus der Semantik zweier schnittstellenkompatibler offener Systeme eine Gesamtsystemsemantik liefert.

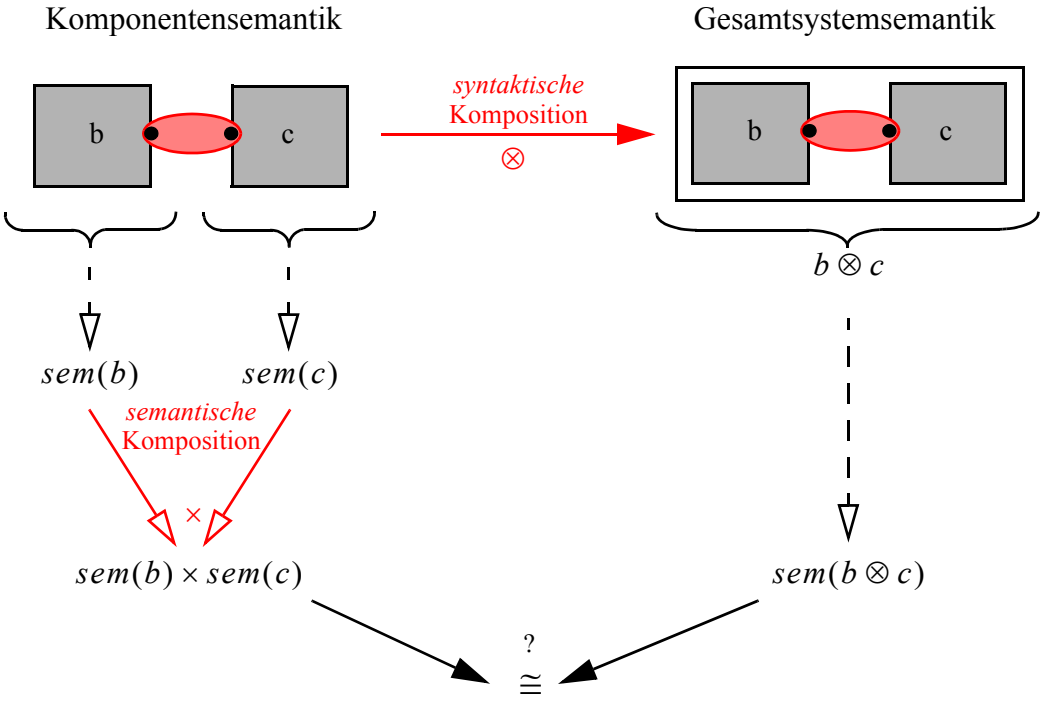


Abbildung 7-17: Komponenten- und Gesamtsystemsemantik

Betrachten wir nun das Szenario in Abb. 7-17, in dem auf der linken Seite die Semantiken $sem(b)$ und $sem(c)$ zweier offener Systeme (b und c) durch einen auf den Teilsemantiken definierten *semantischen Kompositionsoperator*[56] („×“) zur Systemsemantik $sem(b) \times sem(c)$ verschmolzen wird. Die rechte Seite zeigt, wie aus den beiden offenen Systemen durch eine *syntaktische Komposition* („⊗“) zunächst das (potentiell geschlossene) *Gesamtsystem* $b \otimes c$ gebildet wird, um dann daraus eine Semantik $sem(b \otimes c)$ des resultierenden Systems zu bilden.[57]

---

56. Über die beteiligten Komponenten hinaus sind u. U. noch weitere semantikrelevante Faktoren an der Komposition beteiligt. Dies schließt in Estelle z. B. die Verbindungsstruktur ein. Solche im Wesentlichen nur zur Kopplung der Hauptkomponenten hinzugefügten Faktoren bezeichnet man auch als „*Glue*“. Diese zusätzlichen Glue-Faktoren müssen natürlich auch als zusätzliche Parameter G der Kompositionsoperatoren modelliert werden, und es ergeben sich somit die parametrierten Operatoren „$\times_G$“ bzw. „$\otimes_G$“. Alternativ kann der Glue selbst auch als ein (nach unten) offenes System betrachtet werden, das eine eigene Semantik $sem(G)$ besitzt und durch eine Verallgemeinerung auf n-stellige syntaktische und semantische Kompositionsoperatoren berücksichtigt werden kann. Wir verzichten in der nachfolgenden Darstellung jedoch auf die explizite Nennung dieses Zusatzparameters.

57. Es ist zu beachten, dass die beiden Kompositionsbegriffe auf unterschiedlichen Domänen (nämlich der Syntax der Beschreibung bzw. ihrem semantischen Modell) operieren.

Offensichtlich ist es wünschenswert, dass beide Kompositionsbegriffe letztlich zu einer *äquivalenten Semantik* führen, *sem* also ein *Homomorphismus* zwischen dem *syntaktischen* Kompositionsbegriff auf den Systembeschreibungen und dem *semantischen* Kompositionsbegriff auf den (Teil- oder Gesamtsystem-) Semantiken ist.

Ergeben sich Differenzen zwischen beiden Semantiken, stellt dies nicht nur den Nutzen einer Teilsystemsemantik in Frage,[58] sondern es gefährdet sogar die Qualifizierung als formale Beschreibungstechnik.[59]

Ein effektiver Weg zur Vermeidung einer solchen Divergenz ist die Fundierung der Gesamtsystemsemantik auf die Teilsemantiken der (potentiell) komponierten Teilsysteme. Hierdurch erfolgt die Entwicklung einer Gesamtsystemsemantik letztlich durch eine rekursive Dekomposition der unmittelbaren Teilsysteme und eine anschließende semantische Rekomposition der gewonnenen Teilsemantiken. Es ergibt sich damit der in Abb. 7-18 dargestellte Weg zur Gewinnung der Semantik $sem(b \otimes c) = sem(b) \times sem(c)$.

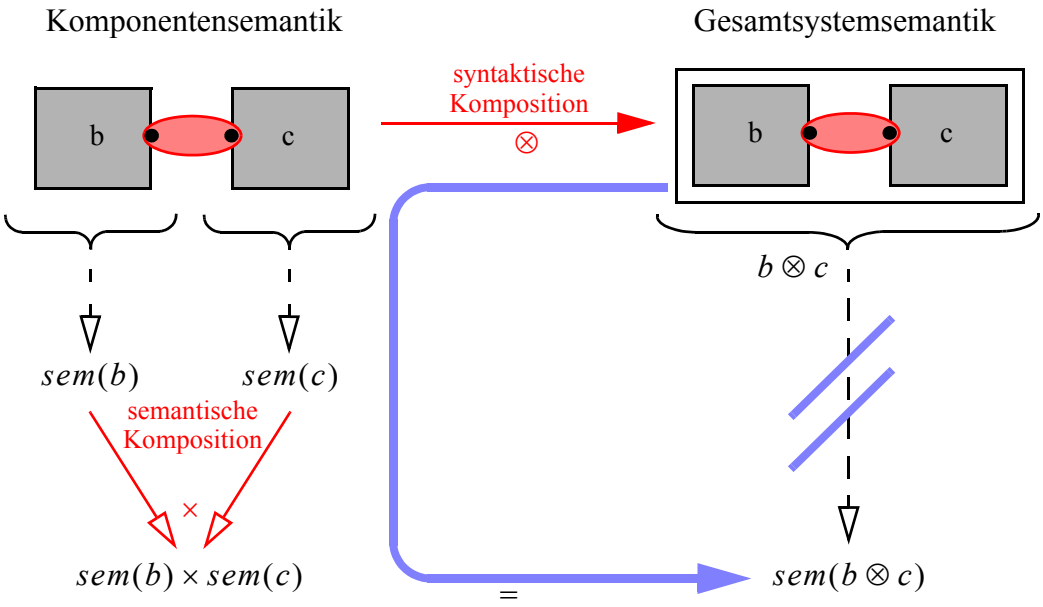


Abbildung 7-18: Auf Komponentensemantik basierende Gesamtsystemsemantik

Diese Art der semantischen Fundierung ist dabei wohldefiniert, wenn neben der Teilsystemsemantik auch die syntaktische Dekomposition (als Umkehrung der syntaktischen Komposition) in ihrem transitiven Abschluss eindeutig ist.[60] Dadurch ist sichergestellt, dass alle denkbaren Partitionierungen des Gesamtsystems in Komponenten als mögliche Ausgangssituation der syntaktischen Komposition berücksichtigt sind.

58. Letztlich gäbe es dann keine kompatible Anwendung der Teilsystemsemantik.

59. Voraussetzung für die Qualifikation als formale Beschreibungstechnik ist eine formale (also im mathematischen Sinne *eindeutige*) Semantik.

60. Dies ist keine notwendige, aber eine hinreichende Bedingung. Bei einer mehrdeutigen Dekomposition müssten jedoch alle möglichen Dekompositionen betrachtet werden, und deren Semantiken müssten bei der semantischen Rekomposition in jedem Fall zueinender äquivalente Gesamtsystemsemantiken liefern.

7.3.3.2 Übertragung auf den Modulinstanzhierarchiebegriff von Estelle

Estelle erfüllt die o. g. Forderung nach eindeutiger Dekomponierbarkeit, wenn man dabei das System entlang seiner Modulstruktur rekursiv vollständig zerlegt. Dies entspricht insbesondere auch genau der Granularität der potentiellen Aufteilung in offene Systeme unter Open-Estelle. Es entsteht dabei ein System von Modulen, die strikt hierarchisch angeordnet sind.

Als nächstes benötigen wir eine Teilsystemsemantik eines Moduls, die sich ggf. auf die Teilsystemsemantik der Kindmodule stützt. An dieser Stelle zeigt sich, dass die Struktur der bisherigen Estelle- und Open-Estelle-Semantik ungeeignet ist, da die „*next global situation*"-Relation (siehe Def. 7.12 bzw. Def. 7.13) genau wie auch das zur Repräsentation des Systemzustands genutzte Tupel sit_P (insbesondere mit seiner Komponente gid_P) die globale Systemstruktur enthält.

Diese globale Sicht setzt sich in praktisch allen zu Grunde liegenden Zustandsraumprädikaten und -Übergangsrelationen fort. So wird z. B. beim Transport einer durch das Feuern einer Transition *t* generierten Nachricht durch eine global wirkende Zustandstransformation ($transmission_P(gid'_{SP})$ mit $gid'_{SP} \in [t]_P(gid_{SP})$; siehe Clause 9.5.4, [ISO97]) direkt auf die Zielwarteschlange zugegriffen, die typischerweise in einem bzgl. der Modulhierarchie völlig unabhängigen Modul liegt. Ähnliches gilt für die Auswirkungen z. B. von `detach`-Statements auf Warteschlangen an den Endpunkten der Attach-Kette.

Die geforderte rekursive Teilsystemsemantik müsste dagegen direkt auf Basis der Semantik der unmittelbaren Kindmodule operieren. Dazu zerlegt man das Modul (bzw. die Modulinstanz, s. u.) rekursiv in

(i) seine Kindmodule und

(ii) seine übrigen, unmittelbar enthaltenen semantikrelevanten Komponenten.

In *(ii)* sind dann die Zustandsraumkomponenten (Kontrollzustände, Variablen, Warteschlangen, etc.), und die Transitionen enthalten.[61] Anschließend definiert man auf Basis einer lokalen Modulinstanzsemantik, die keinen direkten Bezug auf die Semantiken von Kindmodulinstanzen nimmt, durch Bezugnahme auf die Teilsystemsemantiken der jeweiligen unmittelbaren Kindmodule die geforderte rekursiv definierte Teilsystemsemantik für das betrachtete (offene oder geschlossene) Modul. Im transitiven Abschluss ergibt sich so für eine komplette Spezifikation eine zur Teilsystemsemantik voll kompatible Gesamtsystemsemantik.

Doch wie könnten nun eine solche Semantik und der verwendete semantische Kompositionsbegriff aufgebaut sein, um die oben geforderten Eigenschaften zu erreichen? Ein interessanter Ansatz beruht auf der Vorstellung, die Modulinstanzhierarchie als Träger der semantischen Struktur aufzugeben und letztlich eine Semantik auf Basis der (im obigen Sinne lokalen) Semantiken *aller* im (Gesamt- oder Teil-) System vorhandenen Modulinstanzen zu bilden. Die Modulhierarchie dient in dieser Semantik nur noch zur Synchronisation und zur Auswertung direkt hierarchiebezogener Operationen (wie z. B. die Termination von Kindmodulinstanzen aufgrund der Termination einer Vatermodulinstanz).

Es ergibt sich so eine (nicht hierarchisch strukturierte) Menge von Modulinstanzen, deren externe *Schnittstellen* teilweise fest (die bisherige Schnittstelle, in der Modulhierarchie nach „oben") und teilweise (potentiell) *variabel* sind (die „untere" Schnittstelle zur variablen Menge

61. Die sonstigen Definitionen dienen letztlich nur zur Beschreibung der Zustandsraumkomponenten (z. B. Typdefinitionen) oder der Transitionen (z. B. Funktionen, die direkt oder indirekt von Transitionen aufgerufen werden).

der Kindmodulinstanzen als Summe von deren „oberen“ Schnittstellen). Die Zuordnung der unmittelbaren Kindmodule erfolgt dabei nicht über Aggregation, sondern über eine Referenzierung (Assoziation) der Kindmodulinstanzen.

Wir illustrieren im folgenden Beispiel diese Idee anhand der bereits in Abb. 2-1 auf Seite 6 dargestellten Modulinstanz- und Verbindungshierarchie:

Beispiel 7.63: Semantisches Modell zur Modulhierarchisierung

Ausgehend von der in Abb. 7-19-oben angegebenen Modulinstanzhierarchie abstrahieren wir zunächst von der Hierarchie der Modulinstanzen und betrachten diese forthin im Wesentlichen[62] nur noch als unstrukturierte Menge von (offenen) Systemen.

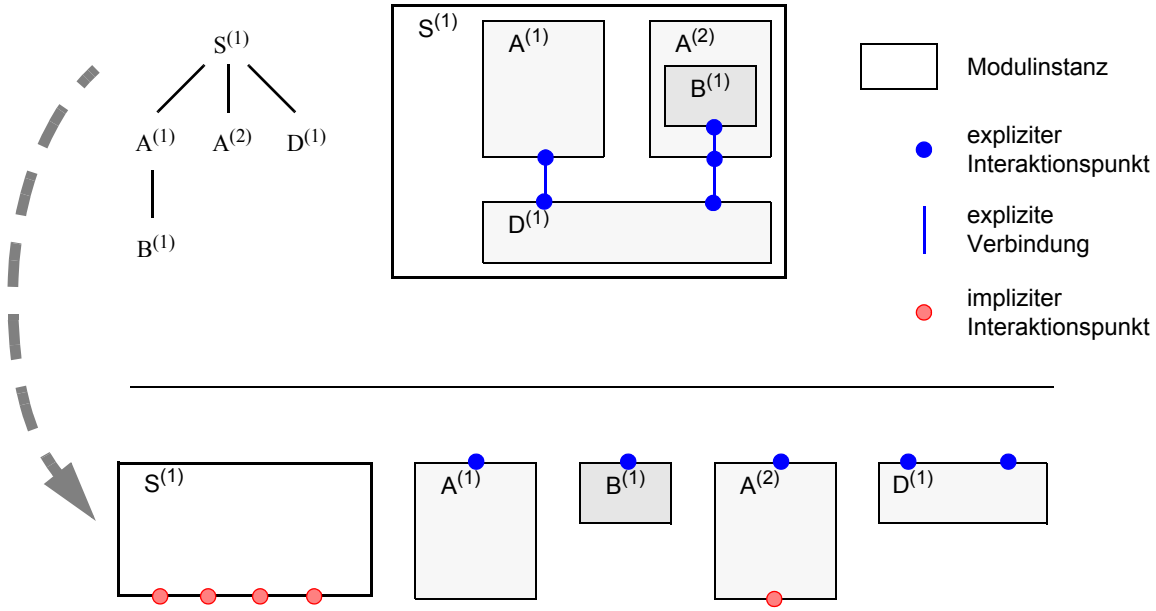


Abbildung 7-19: Semantische Transformation der Modulinstanzhierarchie

Bei dieser Transformation erhalten die offenen Systeme neben ihren bisherigen explizit definierten externen Schnittstellen (obere Reihe von Interaktionspunkten in Abb. 7-19-unten) zusätzlich noch die externen Schnittstellen ihrer Kindmodulinstanzen (untere Reihe von Interaktionspunkten in Abb. 7-19-unten). Diese dienen aufgrund der Abtrennung der Kindmodulinstanzen als Ersatz für die ehemals gemeinsamen Schnittstellen.

Diese implizit von den Kindmodulinstanzen „geerbten“ Schnittstellen werden anschließend mit den zugehörigen externen Schnittstellen der jeweiligen Kindmodule verbunden (Glue).[63] Es ergibt sich damit die in Abb. 7-20 dargestellte Konstellation.

62. Die Vater-Sohn-Beziehung bleibt zur Definition bestimmter semantischer Aspekte erhalten, sie verliert jedoch ihre Eigenschaft als prägende Struktur des Systems.

63. Dies entspricht bei Interaktionspunkten in erster Näherung der Wirkung eines entsprechenden **ATTACH**-Statements. Die semantische Transformation einer Kommunikation über gemeinsame Variablen erfordert eine getrennte Betrachtung.

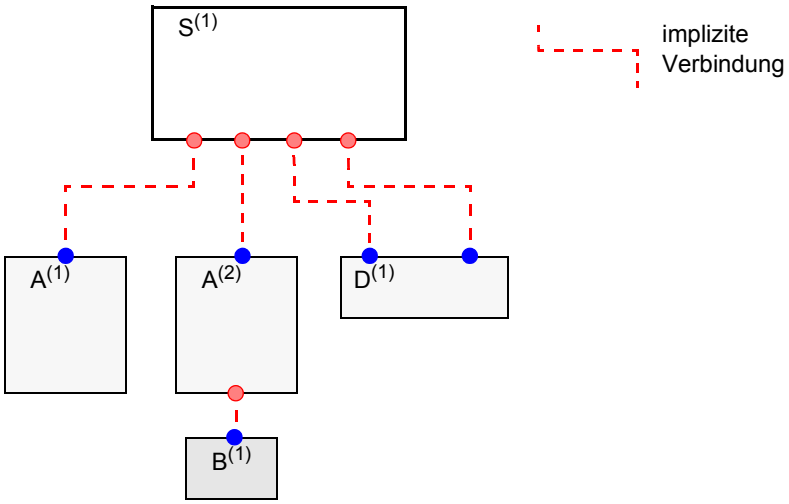


Abbildung 7-20: Implizite Verbindung von (früher gemeinsamen) Interaktionspunkten

Zuletzt erfolgt die Abbildung der (explizit erzeugten) Verbindungsstruktur des Ausgangssystems. Es ergibt sich aus der Konstruktion des Modells, dass derartige Verbindungen[64] ausschließlich jeweils innerhalb einer der erweiterten Modulinstanzen (siehe Abb. 7-21) auftreten.

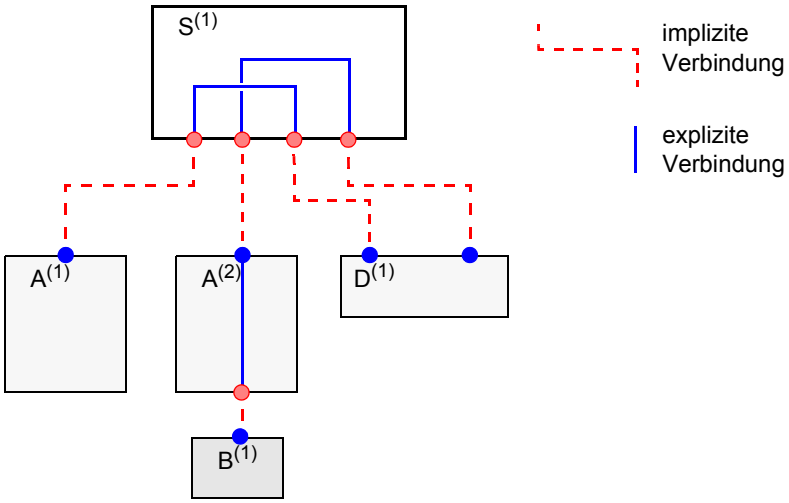


Abbildung 7-21: Explizite und implizite Verbindungsstrukturen

(Ende von Beispiel 7.63)

Offensichtlich liefert die beschriebene semantische Modellbildung unter Beibehaltung der Nachrichtenkommunikationsstrukturen eine Menge *strukturell gleichberechtigter offener Systeme*. Die ursprüngliche Hierarchie der Modulinstanzen wird lediglich zur Steuerung bestimmter, zunächst nachrangiger Aspekte (z. B. Synchronisation) benötigt.

Damit sind wir dem Ziel einer einheitlichen Kompositionssemantik für „*nach oben*“ und „*nach unten*“ offene Systeme deutlich näher gekommen: Es existiert lediglich noch eine Menge gleichberechtigter Teilsysteme, die miteinander kommunizieren. Die Gesamtsystemsemantik ergibt sich in einem solchen Modell entsprechend idealerweise durch die *semantische Kompo-*

64. Der Begriff „Verbindung“ schließt hier sowohl durch **CONNECT** als auch durch **ATTACH** erzeugte Beziehungen zwischen Interaktionspunkten mit ein.

sition der Einzelsemantiken, wobei ggf. erwünschte hierarchieorientierte Synchronisationsanforderungen sowohl im Rahmen der Kompositionssemantik, als auch explizit durch die Einzelsemantiken umgesetzt werden können. In der letzteren Variante wäre dies durch den impliziten Austausch von Synchronisationsnachrichten (über zusätzliche Kommunikationspfade) möglich, wie es z. B. in Petrinetzen [Pet62] in Form von Marken geschieht. Solche Synchronisationsnachrichten könnten z. B. das Recht zur Auswahl einer Transition anzeigen und zur Einhaltung der Vater-Sohn-Priorität von Estelle von einem (Vater-) Modul nach erfolgloser lokaler Transitionsauswahl an eines der ursprünglichen Kindmodule (bei `ACTIVITY`-Attributierung) oder alle Kindmodule (bei `PROCESS`-Attributierung) weitergegeben werden.

Auch andere Operationen, die in der bisherigen Estelle-Semantik eine globale Systemsicht erfordern, könnten in der neuen Semantik allein über unmittelbare Vater-Kind-Beziehungen realisiert werden. So könnte z. B. der globale Transport einer Nachricht über eine Verbindung (und die möglicherweise vor- und nachgeschalteten Attach-Ketten) leicht *Schritt für Schritt* zwischen den beteiligten Komponenten erfolgen, indem z. B. rekursive Aktivierungen entsprechender Schnittstellenmethoden der in diesem Sinne unmittelbar benachbarten Modulinstanzen (also ausschließlich entlang der Modulinstanzhierarchie) erfolgen.

Offensichtlich lassen sich durch diesen Ansatz hierarchieorientierte semantische Spezialfälle, wie sie in der Estelle-Semantik zu finden sind, durchaus realisieren. Insbesondere werden keinerlei Annahmen über die interne Funktionsweise oder Strukturierung von Teilsystemen gemacht; lediglich die zur Interaktion mit der Umgebung erforderlichen Schnittstellen werden vorausgesetzt. Da somit die Semantik eines Teilsystems tatsächlich auf der Basis externen Verhaltens semantisch erfasst wird und keine konkreten internen Strukturen vorausgesetzt werden, bildet dies eine wesentliche Voraussetzung für eine semantisch abgesicherte Interoperabilität mit anderen Beschreibungstechniken bis hin zu konkreten Implementierungen.

Darüberhinaus erlaubt die Aufgabe der Aggregationshierarchie in der semantischen Darstellung der Teilsysteme aber unter Umständen auch die Abstraktion von der Art der Offenheit eines Systems: Ein offenes System im bisherigen Sinne ist ein vollständiger Teilbaum einer Modulinstanzhierarchie mit einer nicht leeren externen Schnittstelle (nach „oben“ offen); andererseits ist auch eine Umgebung für ein solches System für sich ebenfalls ein offenes System, deren Offenheit nun jedoch im Fehlen der internen Beschreibung eines Teilbaums der Modulhierarchie besteht (nach „unten“ offen). In der oben eingeführten Abstraktion unterscheiden sich beide Sichtweisen nur unwesentlich, da die Hierarchie wegfällt und damit beide offenen Systeme nur anhand der externen Schnittstellen als solche erkennbar sind. Insbesondere ist bei einem geeigneten Kompositionsbegriff auch das Zusammenfügen beider Beschreibungen denkbar. Im folgenden Abschnitt betrachten wir abschließend diesen Aspekt nochmals genauer.

7.3.3.3 Semantische Repräsentation und Komposition

Die vorgestellte semantische Repräsentation der Modulinstanzhierarchie ist prinzipiell mit verschiedenen Semantikrepräsentationen für ein offenes Teilsystem bzw. für die oben genannte lokale Semantik einer einzelnen Modulinstanz (z. B. der obersten eines Instanzteilbaumes) verträglich, solange der geforderte Kompositionsbegriff der daraus semantisch komponierten Teilsysteme sinnvoll anwendbar ist.

Betrachten wir dazu als Beispiel eine Semantik offener Teilsysteme auf Basis endlicher Automaten, die jedem offenen oder geschlossenen Teilsystem S_i als Semantik das Tupel $A_i = (I, O, Q, \delta, q_0)$ zuordnet mit

- den Ein- und Ausgabealphabeten I und O,

- der Zustandsmenge Q,
- der Zustandsübergangsrelation $\delta : Q \times I \rightarrow 2^{Q \times O}$ und
- dem Anfangszustand $q_0 \in Q$.

Die Elemente der Ein- und Ausgabealphabete I und O sind dabei jeweils Paare (ip, msg). Dabei identifiziert $ip \in IP$ den jeweiligen Interaktionspunkt[65], über den die Interaktion vom offenen System empfangen bzw. über die sie vom offenen System emittiert wird, wohingegen *msg* die eigentliche Nachricht enthält.

Auf dieser Basis kann z. B. ein *Kompositionsbegriff* definiert werden, der die oben beschriebene implizite Verbindung von Interaktionspunkten zwischen den Modulinstanzen als partielle[66] bijektive Abbildung $con : IP \times IP$ realisiert. Das Gesamtsystem erhält wiederum eine Automatensemantik nach obigem Schema. Wird nun von einem Teilsystem eine Ausgabe (ip, msg) gemacht, so wird diese als Eingabe $(ip, con(msg))$ an das Empfängersystem weitergeleitet,[67] sofern *con* an dieser Stelle definiert ist. Ansonsten handelt es sich um eine Ausgabe an eine externe Schnittstelle des offenen Gesamtsystems, die wiederum als Ausgabe des entstehenden Automaten gehandhabt werden muss.

Wir verzichten an dieser Stelle auf eine umfassende Formalisierung der skizzierten Semantik und betrachten stattdessen einige der wesentlichen Designparameter, die bei der Formalisierung zu berücksichtigen sind, und ihren Einfluss auf die semantische Kompatibilität zu Standard-Estelle:

- *Modulsynchronisation*:

  Standard-Estelle definiert eine relativ enge Synchronisationsbeziehung zwischen benachbarten Modulinstanzen (u. a. Vater-Sohn-Priorität und Geschwistermodulsynchronisation durch `PROCESS`-Attribut; siehe auch Abschnitt 7.2.4). Diese Synchronisation kann auch bei einer kompositionsverträglichen Semantik offener Teilsysteme nachgebildet werden. Die wesentlichen Ansätze dazu sind die bereits oben andiskutierten expliziten Synchronisationsnachrichten zwischen den Teilsystemen oder ein entsprechend attributierter Kompositionsmechanismus. Alternativ wäre auch eine Reduktion der Synchronisation denkbar, wie sie in Abschnitt 4.2.2.4 anhand der *Independent-Module-Erweiterung* diskutiert wurde.

- *Empfangswarteschlangen*:

  Diese bleiben wie auch schon in der Standard-Estelle-Semantik vorteilhafterweise Teil des empfangenden Interaktionspunkts. Daraus ergibt sich zusammen mit der unbegrenzten Kapazität der Warteschlangen, dass die offenen Teilsysteme jede bzgl. der Signatur (d.h. syntaktisch prüfbar) passende Nachricht auch annehmen. Dies ist insbesondere bei einer alternativen Semantikdefinition auf Basis *akzeptierter Sprachen* ein wichtiger Aspekt.

---

65. Da die Module im o. g. Modell aufgrund der Möglichkeit zur dynamischen Erzeugung von Kindmodulinstanzen eine variable und damit unbegrenzte externe Schnittstelle haben, ist die Trägermenge *IP* der Interaktionspunktmarken nicht endlich.

66. Sollte das aggregierte System selbst nicht geschlossen sein, so sind die der externen Schnittstelle zugeordneten Interaktionspunkte nicht im o. g. Sinne implizit verbunden.

67. Wir gehen hier davon aus, dass das empfangende System jede Nachricht auch annimmt (s. u.).

- *Transport von Nachrichten über Attach-Ketten*:

  Durch Attach-Ketten (siehe z. B. Abb. 2-6 auf Seite 20) können in Estelle direkte Verbindungen über mehrere Stufen der Modulinstanzhierarchie geschaffen werden. In der Standard-Estelle-Semantik erfolgt der Transport über diese Ketten durch direkten (globalen) Zugriff innerhalb der Modulhierarchie. Dies ist mit der rein lokalen Kommunikation der benachbarten (also im obigen Sinne implizit verbundenen) Module nicht ohne weiteres verträglich. Ein schrittweiser Transport über die Attach-Kette wahrt dagegen die Geheimnis- und Lokalitätsprinzipien des Semantikentwurfs, ist aber zunächst[68] inkompatibel zu der bisherigen Semantik.

- *Gemeinsame Variablen*:

  Da gemeinsame Variablen Zustandsraumkomponenten mehrerer offener Teilsysteme zugleich sind, entstehen hier sehr enge Einflussnahmen und Synchronisationen zwischen den beteiligten Modulinstanzen, die zunächst nicht ohne weiteres auf das semantische Modell kommunizierender endlicher Automaten abgebildet werden können. Ein Ansatz ist die Modellierung des Zugriffs auf gemeinsame Variablen über Nachrichten, wie sie u. a. in der SDL-Semantik [ITU94] zu finden ist. Dies ist jedoch aufgrund der groben Granularität atomarer Abschnitte in Estelle nicht unproblematisch. Die Vater-Sohn-Priorität[69] von Standard-Estelle erlaubt hier jedoch u. U. eine wesentliche Vereinfachung, indem der aktuelle Wert der gemeinsamen Variablen zusammen mit der oben andiskutierten Markierung zur Synchronisation zwischen Teilsystemen übertragen werden könnte.

Es gibt noch einige weitere Probleme, die bei der Definition einer vollständig zur Standard-Estelle-Semantik kompatiblen *kompositionsverträglichen* Semantik entstehen. Sie beruhen häufig darauf, dass die Estelle-Semantik über weite Strecken auf spezifischen Eigenschaften der Estelle-Komponenten und des Gesamtsystemzustands beruht und nicht für die Komposition offener Systeme geplant war.

Generell scheint hier eine radikale Neudefinition der Estelle-Semantik sinnvoller als eine aufwändige Nachbildung von Semantikartefakten, die häufig vom Spezifizierer nicht einmal gewünscht werden. Die starke Synchronisation ist hier ein typisches Beispiel (siehe auch „Asynchronous-Process“ [BrGo94]). Ein höheres Abstraktionsniveau und mehr Unabhängigkeit zwischen Teilsystem wären hier sicherlich wünschenswert, insbesondere bzgl. der Fragestellungen nach der effektiven Nutzbarkeit der formalen Semantik eines offenen Systems. Als Beispiel sei hier der Beweis von Eigenschaften z. B. einer Protokollmaschine auf Ebene ihres externen Verhaltens genannt.

Wir schließen damit den Exkurs zu alternativen Semantiken und definieren im nächsten Abschnitt mit der textuellen Verschmelzung eine der wesentlichen Grundlagen zur Fundierung der bereits in Abschnitt 7.3 angegebenen Semantik von Open-Estelle auf Basis der Standard-Estelle-Semantik.

---

68. Durch zusätzliche Synchronisationsmaßnahmen kann dieser schrittweise Transport der Interaktionen über Attach-Ketten jedoch „versteckt“ werden, indem diese Transportmaßnahmen Priorität vor anderen Mechanismen erhalten.

69. Gemeinsame Variablen sind in Estelle nur zwischen unmittelbaren Vater- und Sohnmodulinstanzen möglich, sodass hier mit relativ grober zeitlicher Granularität (Modulauswahl- bzw. Schaltzyklus für das ganze Subsystem) ein gegenseitiger Ausschluss beim Zugriff auf gemeinsame Variablen besteht.

7.4. Textuelle Verschmelzung

Eines der primären Ziele bei der Entwicklung von Open-Estelle war größtmögliche Kompatibilität mit der zu Grunde liegenden Spezifikationstechnik Estelle. Einer der wesentlichen Ansätze dazu war die syntaktische Fundierung des *formalen Imports* offener Systeme mittels rein syntaktischer Bezugnahme auf ein externes (und von der importierenden Umgebung unabhängig definiertes) Interface bzw. die darüber zugänglichen offenen Systeme. Im Falle eines resultierenden geschlossenen Systems mündet diese Vorgehensweise direkt in ein System von Spezifikationen, die als Gesamtsystem unmittelbar in das *zu Grunde liegende syntaktische Modell* von Standard-Estelle abgebildet werden können (siehe Abschnitt 7.3.1).

In diesem Abschnitt untersuchen wir nun, inwieweit eine Rückführung eines solchen abgeschlossenen Gesamtsystems offener Teilsysteme auf eine monolithische Spezifikation möglich ist. Dazu definieren wir eine textuelle Transformation dieser Teilsysteme in eine geschlossene Standard-Estelle-Spezifikation.

7.4.1 Grundkonzepte

Die Grundidee der Transformation beruht naheliegenderweise darauf, den (expliziten[70] oder impliziten[71]) formalen Import einer externen Schnittstelle (INTERFACE-DEFINITION) in eine importierende Umgebung durch eine geeignete textuelle Ergänzung der importierenden Umgebung um die jeweils importierten Definitionen abzubilden.

Wie wir bereits in der Motivation zu Open-Estelle gesehen haben, kann diese textuelle Transformation nicht durch *schlichte textuelle Einbettung* der importierten Interface-Definitionen erfolgen, da dabei eine syntaktische Beeinträchtigung der beteiligten Teilsystemdefinitionen nicht ausgeschlossen werden kann (siehe Abschnitt 7.1.2). Entsprechend müssen zuerst mögliche Namenskollisionen und -Redefinitionen durch geeignete Umbenennungen eliminiert werden. Dazu werden allen Namen, die in den Ausgangs-Teilspezifikationen definiert werden, durch einen Namenspräfix ergänzt, der den Kontext ihrer Definition eindeutig beschreibt. Dieses Präfix basiert typischerweise auf dem Namen der jeweiligen textuellen Einheit[72], die die Deklaration enthält (siehe auch Abschnitt 7.1.3). So könnten z. B. die Definitionen aus dem in Abschnitt 7.2.1 angegebenen Beispiel einer Interface-Definition zu folgendem Textfragment transformiert werden:

Beispiel 7.64: Namenserweiterung der Interface-Definition „`binaryService`“
in Abb. 7-8 auf Seite 325

```
TYPE binaryService_tOperand = RECORD x1, x2: REAL; END;
     binaryService_tResult = REAL;

CHANNEL binaryService_binaryServiceChannel (user, provider);
     BY user: request (x: binaryService_tOperand);
     BY provider: respond (y: binaryService_tResult);
```

70. durch ein IMPORT-STATEMENT

71. durch die Referenz eines Interfaces im Kopf einer BEHAVIOUR-DEFINITION

72. also der Name der INTERFACE-DEFINITION, BEHAVIOUR-DEFINITION oder der Spezifikation (siehe auch Abschnitt 7.2.1)

```
MODULE binaryService_binaryOperatorHeader ACTIVITY;
     IP toUser: binaryService_binaryServiceChannel (provider) COMMON QUEUE;
     END;

BODY binaryService_binaryOperator FOR binaryService_binaryOperatorHeader;
     EXTERNAL;
```

(Ende von Beispiel 7.64)

Wie bereits im Beispiel dargestellt, müssen lediglich die direkt referenzierbaren Namen transformiert werden (also z. B. nicht die Record-Komponenten). Weiterhin werden gleichzeitig mit den Deklarationen der Namen natürlich auch global alle Namensreferenzen entsprechend transformiert. Dies schließt auch alle externen Referenzen in den übrigen textuellen Objekten ein. Die QUALIFIED-IDENTIFIER werden dabei von ihrer Open-Estelle-Qualifizierung befreit.

Diese Transformation zusammen mit der Eindeutigkeit der Namen offener Systeme (siehe Abschnitt 7.2.3) ermöglicht es, Namenskollisionen zwischen importierten Deklarationen zu vermeiden. Es bleiben noch die bereits in Abschnitt 7.1.2 diskutierten Namenskollisionen mit dem importierenden Kontext und mit vordefinierten Namen zu unterbinden. Ersteres betrifft nur Namen, die nicht aus importierten Interface-Definitionen stammen, da alle anderen Namen bereits qualifiziert wurden. Die Namenspräfixe müssen hier lediglich so gewählt werden, dass sie von allen anderen Präfixen importierter Systeme verschieden sind. Zuletzt muss das Präfix-Namensschema so gewählt werden, dass in keinem Fall ein vordefinierter Name entstehen kann.[73]

Als nächster Schritt werden die MODULE-BODY-DECLARATIONS (also die mit „EXTERNAL“ gekennzeichneten Moduldeklarationen in den Interfaces) durch die entsprechenden Modulrümpfe der zugeordneten BEHAVIOUR-DEFINITION ersetzt. Dabei werden dann auch gegebenenfalls die in Abschnitt 7.2.4 eingeführten erweiterten Modulattributierungs- und -Reinterpretationsregeln umgesetzt.

Ein interessanter Sonderfall ergibt sich, wenn keine derartige Zuordnung eines offenen Systems möglich ist.[74] In diesem Fall war das zu transformierende Ausgangssystem in dieser Hinsicht bereits unvollständig definiert.[75] Hier wäre es denkbar, die „EXTERNAL“-Deklaration in die transformierte Standard-Estelle-Spezifikation zu übernehmen, wodurch die resultierende Spezifikation zwar syntaktisch korrekt, jedoch semantisch undefiniert bliebe. Dieser Übergang verdeutlicht nochmals den Unterschied zwischen der MODULE-BODY-DECLARARATION in einer Open-Estelle-Interface-Definition und der textuell identischen, aber semantisch nicht unterstützten Unvollständigkeit des entsprechenden Konstrukts in Standard-Estelle (siehe Abschnitt 7.2.1).

---

73. Dies ist für die im obigen Beispiel bereits durch das im Präfix enthaltenen Zeichen „_“ gewährleistet, das in keinem der „*Required Procedures and Functions*“ (siehe Abschnitt 6.6.4, ff. von Annex C [ISO97]) oder „*Required Simple-Types*“ (siehe Abschnitt 6.4.2.2 von Annex C [ISO97]) vorkommt. Werden weitere vordefinierte Namen eingeführt, so müssen die Präfixe ggf. durch zusätzliche Erweiterungen eindeutig gemacht werden.

74. also die notwendige BEHAVIOUR-DEFINITION nicht vorliegt (siehe Abschnitt 6.2.3)

75. Im Sinne der in Abschnitt 7.3.3 diskutierten Semantikansätze könnte das Gesamtsystem alternativ auch als (in diesem Fall „nach unten“) offenes System interpretiert werden. Wir gehen an dieser Stelle bei der vorgestellten Transformation nach Standard-Estelle jedoch naheliegenderweise von einem abgeschlossenen Gesamtsystem aus.

7.4.2 Replikation von Definitionen

Auf den ersten Blick scheint die textuelle Einbettung der so transformierten Interfaces in importierenden Umgebungen bereits die Grundkonzepte von Estelle nach Open-Estelle ausreichend abzudecken, und tatsächlich führt sie in einfachen Szenarien zum Erfolg. Sobald jedoch das selbe offene System an mehreren Stellen der resultierenden Modulhierarchie importiert wird, ergeben sich konzeptionelle Brüche.

Wie wir in Abschnitt 7.2.3 gesehen haben, liefert der Import der externen Schnittstelle eines offenen Systems syntaktisch lediglich Zugriff auf die dort deklarierten syntaktischen Objekte. Erfolgt der Import an mehreren Stellen der Modulhierarchie, so beziehen sich die importierten Namen immer noch auf die selbe Definition und sind damit (z. B. im Fall von Typdefinitionen) zueinander kompatibel. Kopiert man diese Definitionen jedoch in verschiedene Knoten der Modulhierarchie, so entstehen u. U. identisch definierte, aber unterschiedliche und damit inkompatible Definitionen (Widerspruch zum Konzept des „*single point of definition*“).

Glücklicherweise kann diese Problematik durch eine differenzierte Handhabung der importierten Definitionen gelöst werden. Dazu werden bestimmte Definitionen im Modulhierarchiebaum so weit „nach oben“ verlagert, bis alle Referenzen aus Knoten stammen, die nicht oberhalb liegen. Im einfachsten Fall ist dies schlicht immer das Spezifikationsmodul als Wurzel der Hierarchie. Es ist zu beachten, dass diese Verschiebung aufgrund der vorangegangenen Namenstransformationen unkritisch bzgl. Namenskollisionen ist.

Bei den zu verschiebenden Definitionen handelt es sich um Konstanten-, Typen- und Kanaldefinitionen, die aus Interfaces stammen. Diese müssen zur Erhaltung der Typkompatibilität zentral definiert werden. Da diese Definitionen in Standard-Estelle in der Modulhierarchie auch in allen Kindmodulen sichtbar sind, ist so die geforderte einheitliche Definition möglich.

Leider sind jedoch die noch verbliebenen Modulheader- und Modul-Body-Definitionen nicht über Modulgrenzen hinweg sichtbar, so dass eine derartige Verschiebung „nach oben“ in der Modulhierarchie hier nicht möglich ist. Glücklicherweise ist bei diesen Definitionen jedoch auch eine Replikation der Definitionen in verschiedenen Knoten der Modulhierarchie möglich, da von diesen Definitionen Typkompatibilitäten über Modulgrenzen hinweg definiert werden.[76]

Ein Nachteil dieser Vorgehensweise ist, dass die Replikation der Modul-Body-Definitionen, die ja selbst ganze Modulhierarchien enthalten können, den Umfang der resultierenden Gesamtspezifikation exponentiell steigern kann. Jedoch wäre dieser Code-Overhead bei einer „bottom-up“ Entwicklung einer Standard-Estelle-Spezifikation ebenfalls nicht zu vermeiden, wenn tatsächlich die Notwendigkeit für das Auftreten einer Modul-Body-Definition an verschiedenen Knoten des Modulhierarchiebaums besteht. Sie ist im Gegenteil ein Indiz für die Effizienz von Open-Estelle bei der Wiederverwendung von Spezifikationsteilen.

76. Genauer: Von Modulheader-Definitionen hängen nur Modul-Body-Definitionen und Modulvariablen ab und von letzteren gibt es keine Typabhängigkeiten.

7.4.3 Heterogene Timescale-Einheiten

Ein anderes Problem ergibt sich durch die Möglichkeit unterschiedlicher *Timescales* in den verschiedenen Behaviour-Definitionen. Da nach der Verschmelzung in der resultierenden Standard-Estelle-Spezifikation nur noch eine einheitliche Timescale erlaubt ist, müssen die verschiedenen Zeitmodelle bei der Verschmelzung vereinheitlicht werden.

Leider ist dieser Vorgang aufgrund der mangelnden semantischen Definition von Zeit und der Bedeutung der verschiedenen Timescales nicht automatisierbar: In Abschnitt 5.3.5 von [ISO97] wird die Bedeutung der Timescale folgendermaßen beschrieben: „*[...] a timescale, optionally given in the specification, is treated as semantically irrelevant; it is merely an indication of the designer's intentions.*“ Entsprechend ist hier bei der Verschmelzung von Systemen mit unterschiedlichen Timescales eine manuelle Nachbearbeitung der Teilsysteme durch den Spezifizierer erforderlich. Mehr noch ist selbst bei namentlich identischen Timescales in verschiedenen Teilspezifikationen sicherzustellen, dass die Intuition bei der Spezifikation der unterschiedlichen Teilsysteme in dieser Hinsicht kompatibel ist.

Auf Implementierungsebene können dagegen kompatible – d.h. auf dem selben (Real-) Zeitmodell basierende – Timescales (z. B. Sekunden und Millisekunden) problemlos kombiniert werden (siehe Abschnitt 7.5.2.2).

7.4.4 Implementierung

Das beschriebene Transformationsverfahren wurde von Dirk Barthel als Kommandozeilenwerkzeug „`petresolve`“ als Ergänzung des im XEC-Toolkit enthaltenen Estelle-Parsers „`PET`“ implementiert (siehe auch Abschnitt 7.5.1). Die Transformationen erfolgen dabei auf Basis der PET-internen Objektstruktur und setzen zur Gewinnung der geschlossenen Spezifikation die Restore-Funktionalität der PET-Klassenbibliothek ein. Ergebnis der Transformation ist dabei ein Standard-Estelle Quelltext, der von allen Estelle-Werkzeugen weiterverarbeitet werden kann (siehe auch Abb. 7-22). Somit bildet „`petresolve`“ auch für Estelle-Werkzeuge, die die Estelle-Erweiterung Open-Estelle nicht unterstützen, eine Möglichkeit zur Implementierung, Analyse oder Weiterverarbeitung von Open-Estelle-Spezifikationen. Eine Anwendung des Werkzeugs ist in Anhang D.2 dargestellt.

Im nächsten Abschnitt werden wir anhand von XEC zeigen, dass als Alternative zur vorgestellten Transformation nach Standard-Estelle die *direkte Implementierung* von Open-Estelle wesentliche Vorteile bietet.

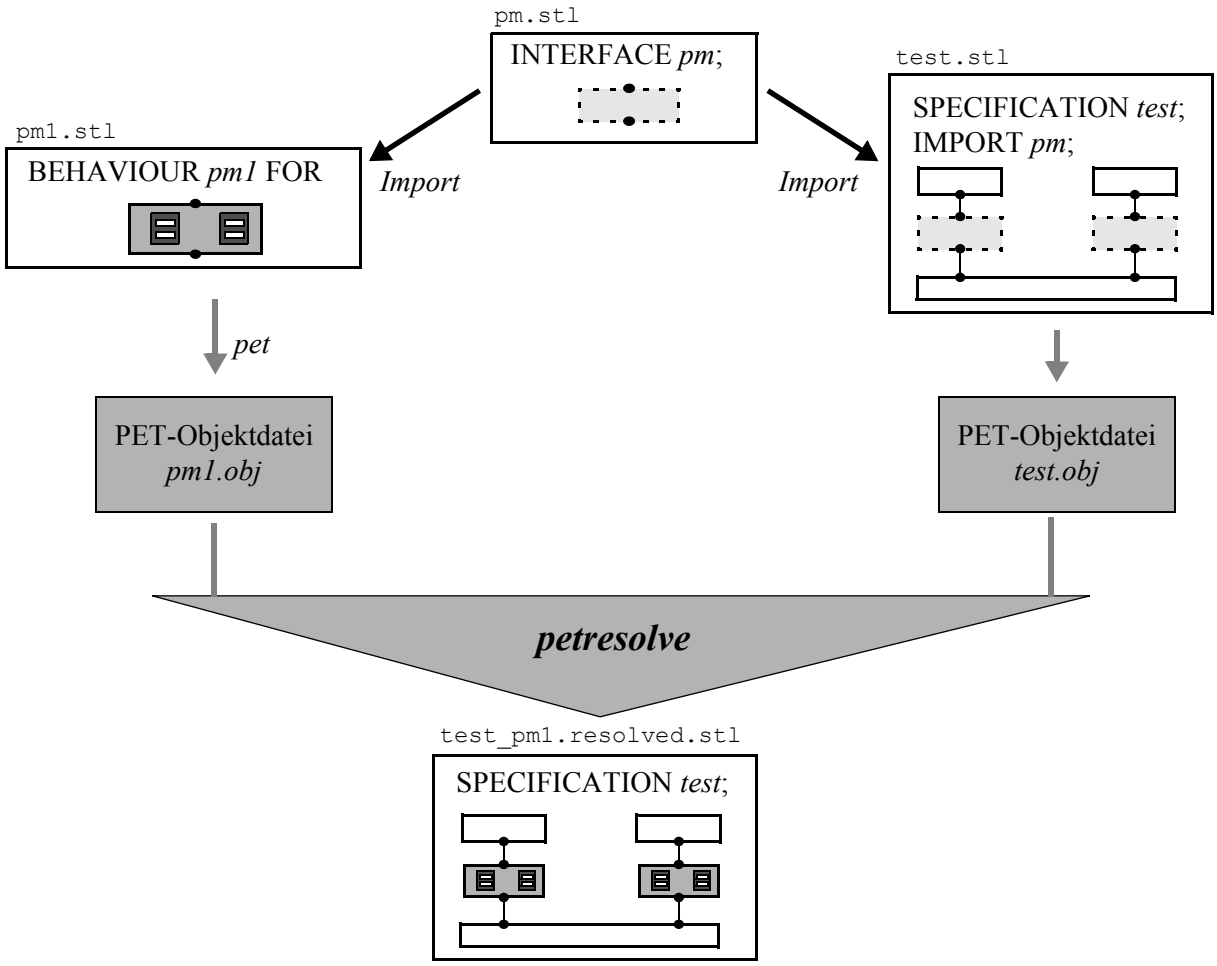


Abbildung 7-22: Verschmelzung offener Spezifikationsteile durch `petresolve`

7.5. Implementierungsaspekte und Toolsupport

Die direkte Implementierung von Open-Estelle im XEC-Toolkit basiert auf drei Komponenten:

- dem Compiler-Frontend PET und dem erzeugten Objektformat,
- dem Implementierungsgenerator XEC und
- der Laufzeitumgebung von XEC.

Wir beginnen mit einem Überblick über die Grundkonzepte der Open-Estelle-Implementierung.

7.5.1 Verarbeitung von Open-Estelle-Spezifikationen mit PET

Wir haben in den vorangegangenen Abschnitten ausgeführt, wie Systeme in Open-Estelle in mehrere Komponenten aufgeteilt und schließlich zu einem Gesamtsystem aggregiert werden können. Einzige gemeinsame Schnittstelle zwischen offenen Systemen und den möglichen importierenden Umgebungen ist dabei das Interface der offenen Systeme (siehe auch Abb. 7-16 auf Seite 343). Eine wesentliche Zielsetzung der Entwicklung von Open-Estelle war die isolierte Prüf- und Verarbeitbarkeit dieser Einzelkomponenten, wodurch eine getrennte Entwicklung der Komponenten und ihre Austauschbarkeit gesichert werden. Diese Trennung ist Grundlage der syntaktischen Konstruktion von Open-Estelle und wird vom XEC-Toolkit beginnend mit dem (dazu entsprechend erweiterten) Compiler-Frontend *PET* weitergeführt.

7.5.1.1 Übersetzung von Open-Estelle-Spezifikionen

Betrachten wir das in Abb. 7-23 dargestellte Szenario einer als offenes System spezifizierten Protokollmaschine (Interface-Definition `pm` und der Behaviour-Definition `pm1`) und der importierenden Spezifikation `test` als Umgebung für diese Protokollmaschine.

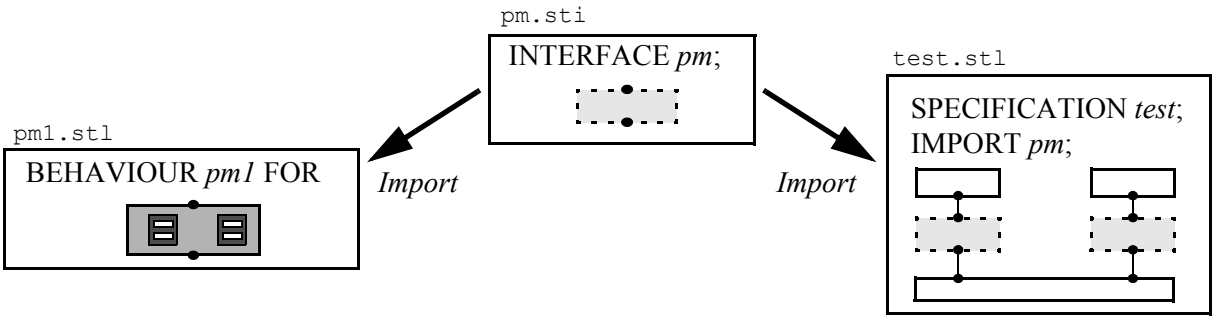


Abbildung 7-23: Beispielszenario offenes System und importierende Umgebung

Die Teilspezifikationen mögen dabei in den entsprechenden Dateien `pm.sti`, `pm1.stl` und `test.stl` abgelegt sein, wobei wir als Konvention die Namenserweiterung „`.stl`“ für Estelle-Spezifikationen und Behaviour-Definitionen einsetzen und für Interface-Definitionen die Namenserweiterung „`.sti`“ (Estelle-Interface) benutzen.[77]

Eine Grundlage der syntaktischen Definition offener Systeme ist die kontextunabhängige Interpretation der Interface-Definition (Abschnitt 7.2.1). Entsprechend kann die Übersetzung der Interface-Definition in die PET-Objektdatei `pm.obi` ebenfalls unabhängig von anderen Spezifikationsteilen[78] erfolgen (siehe auch Anhang D.1).

Die generierte Interface-Objektdatei wird dann bei der Übersetzung anderer Übersetzungseinheiten[79], die das Interface importieren, von PET gelesen und in die syntaktische Struktur integriert. Dabei wird ebenfalls für jede Übersetzungseinheit eine separate Objektdatei (hier: `pm1.obj` und `test.obj`, siehe Abb. 7-24) erstellt, wobei wir analog zu den Quelldateien als Konvention die Namenserweiterung „`.obj`“ für Objektdateien zu Estelle-Spezifikationen und Behaviour-Definitionen einsetzen und bei Interface-Definitionen die Namenserweiterung „`.obi`“ benutzen.

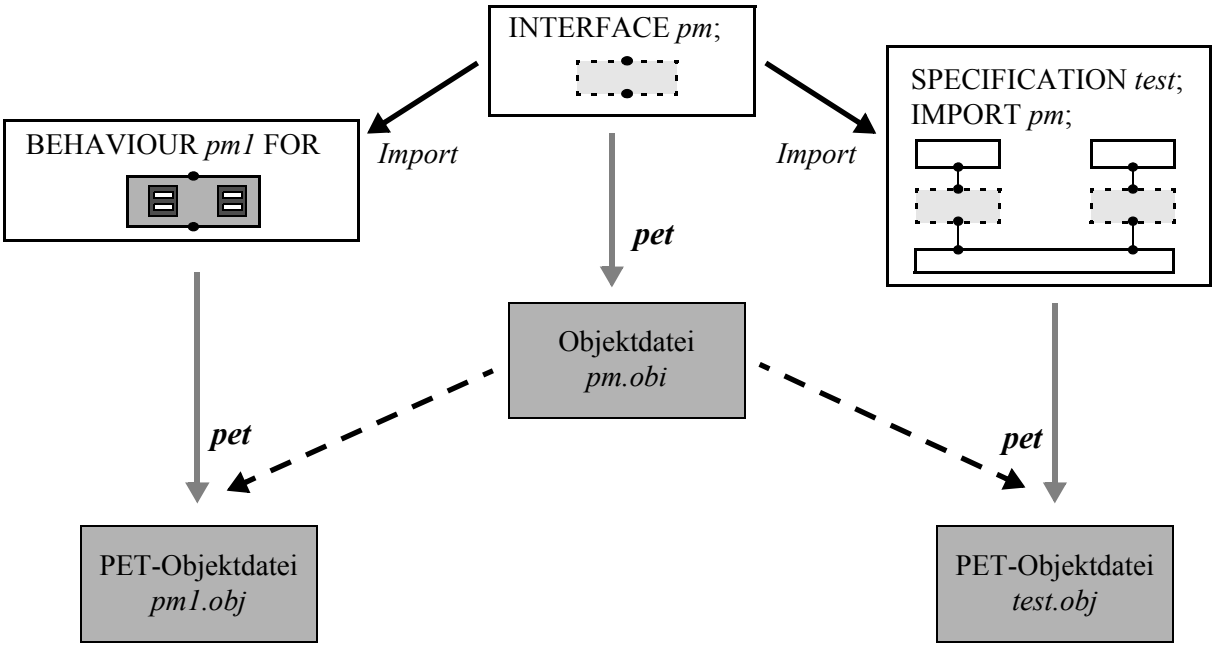


Abbildung 7-24: Getrennte Übersetzung der offenen Spezifikationsteile durch PET

Zur Aktivierung der Open-Estelle-Erweiterung muss PET jeweils mit der Kommandozeilen-Option „`-xopen`“[80] (siehe auch Anhang A.1) aufgerufen werden, durch die u. a. die zusätzlichen Schlüsselwörter von Open-Estelle aktiviert werden.[81] Weiterhin kann gegebenenfalls über die Option „`-I dir`“ ein Suchpfad angegeben werden, in dem (zusätzlich zum aktuellen Verzeichnis) nach Interface-Objektdateien gesucht wird.

77. Die Behaviour- und die Interface-Definitionen können durchaus auch den selben Namen (z. B. pm) tragen. Durch die beschriebene Namenskonvention kommt es auch auf Dateiebene nicht zu Namenskollisionen.

78. Interface-Definitionen können jedoch andere Interface-Definitionen importieren, die dann zuvor mit PET übersetzt worden sein müssen, damit die Objektdateien importiert werden können (s. u.). Da jede Import-Hierarchie fundiert sein muss, ist immer eine entsprechende Folge von Übersetzungen realisierbar (siehe Anhang C.3.2).

79. also Interface-Definitionen, Behaviour-Definitionen oder Spezifikationen

80. Zur Unterstützung eines *Rapid-Prototypings* bei der Dekomposition von Systemen wurde noch eine alternative Option „`-xopen_ur`“ eingeführt, durch die importierte Namen (siehe Abschnitt 7.2.3) zusätzlich auch *unqualifiziert* sichtbar sind, sofern sie nicht von lokalen Definitionen überdeckt werden. Dadurch ist eine schnelle Portierung geschlossener Spezifikationen nach Open-Estelle möglich.

Bei der Generierung der Objektdateien und bei der Suche nach Interface-Objektdateien zu importierten Interfaces werden die o. g. Namenskonventionen automatisch umgesetzt. Die folgenden Aufrufe von PET übersetzen also die drei o. g. Quelldateien:

```
pet -xopen pm.sti
  >> translating input file pm.sti
  >> writing object file pm.obi

pet -xopen pm.stl
  >> translating input file pm.stl
  >> import interface "pm" from "./pm.obi"
  >> writing object file pm.obj

pet -xopen test.stl
  >> translating input file test.stl
  >> import interface "pm" from "./pm.obi"
  >> writing object file test.obi
```

Ein umfassenderes Beispiel ist in Anhang D zu finden.

7.5.1.2 Implementierungskonzepte in PET

Die Implementierung der Open-Estelle-Erweiterung in PET betrifft folgende Aspekte:

- Prüfung der kontextfreien Grammatik (Scanner und Parser)
- Prüfung der nicht kontextfreien Grammatik (Parser und Prüfungen der PET-Bibliothek)
- PET-Objektformat

Die Anpassung des Scanners besteht im Wesentlichen in der dynamisch steuerbaren Aktivierung der neuen Schlüsselwörter und konnte durch die vollständige Neuimplementierung des Scanners mit *flex*[82] (siehe Abschnitt 3.1) sehr kompakt realisiert werden.

Der Parser und die für die zusätzlichen syntaktischen Prüfungen der PET-Bibliothek zuständigen Komponenten mussten neben den offensichtlichen Erweiterungen der kontextfreien Semantik zusätzlich noch bzgl. der komplexeren Sichtbarkeits- und Modulattributierungsregeln von Open-Estelle angepasst werden.

Die von PET aus einer Spezifikation erzeugte *Objektstruktur* musste dagegen nur an zwei Stellen geringfügig erweitert werden. Zunächst wurden die Modulrümpfe um eine Liste der (implizit oder explizit) *importierten Interfaces* ergänzt. Die referenzierten Definitionen der importierten Interfaces integrieren sich dabei vollständig in die PET-Objektstruktur und bedürfen keiner Sonderbehandlung. Insbesondere stellen sich auf dieser Ebene Referenzen auf lokale und importierte Definitionen identisch dar.

---

81. Dadurch können Standard-Estelle-Spezifikationen beim Aufruf von PET ohne diese Option ohne Aktivierung der zusätzlichen Schlüsselwörter und damit ohne Beschränkung bei der Wahl von IDENTIFIERN übersetzt werden. Wurde umgekehrt die Option bei der Übersetzung einer Open-Estelle-Spezifikation irrtümlich weggelassen, so wird dies erkannt und eine entsprechende Warnung ausgegeben.

82. „*fast lexical analyzer generator*“, eine Erweiterung des UNIX-Tools *lex*

Die zweite Erweiterung betrifft die Einführung von INTERFACE-DEFINITION und BEHAVIOUR-DEFINITION als neue *Start-Nichtterminale* von Open-Estelle. Diese sind aus ihrer syntaktischen Struktur heraus eng mit dem dritten Start-Nichtterminal SPEZIFICATION verwandt und werden technisch auch entsprechend im Objektformat realisiert. Lediglich zwei Aspekte mussten hier gesondert behandelt werden:

(i) Unterscheidung zwischen SPEZIFIKATION, INTERFACE-DEFINITION und BEHAVIOUR-DEFINITION

(ii) Referenzierung des Interfaces zu einer BEHAVIOUR-DEFINITION

Der erste Punkt konnte aufgrund der nicht vorhandenen Attributierungsmöglichkeiten für Interface-Definitionen und Behaviour-Definitionen durch zwei zusätzliche Attributwerte im Aufzählungstyp „`enum ModClass`“ ohne Strukturänderung des Objektformats integriert werden. Das im Kopf der BEHAVIOUR-DEFINITION referenzierte Interface[83] (*ii*) wird als erstes Listenelement der Importliste der BEHAVIOUR-DEFINITION übertragen und erfordert somit ebenfalls keine separate Repräsentation. Dies bietet zudem den Vorteil, dass bei einer Weiterverarbeitung der implizite Import des referenzierten Interfaces bereits ohne Sonderbehandlung abgehandelt ist.

Insgesamt wurde die Struktur des Objektformats also lediglich zur Repräsentation der Importlisten erweitert.

Die für die Erzeugung und Weiterverarbeitung des Objektformats verantwortliche PET-Klassenbibliothek musste dagegen umfangreicher angepasst werden, um die mit Open-Estelle verbundenen Erweiterungen der kontextfreien und kontextorientierten Grammatiken zu realisieren. Hier seien die Auflösung qualifizierter Namen, die erweiterten Sichtbarkeitsregeln für importierte Objekte und die erweiterten Modulattributierungsregeln als Beispiele genannt. Die Erweiterungen sind anhand der bedingten Kompilierung („`#ifdef OPEN_EXT`“) leicht erkennbar.[84]

Die Weiterverarbeitung der im obigen Beispiel erzeugten PET-Objektdateien des offenen Systems und der importierenden Umgebung kann getrennt erfolgen. Neben der bereits erwähnten Verschmelzung zu einer geschlossenen Standard-Estelle-Spezifikation (siehe Abb. 7-22, Abschnitt 7.4.4 bzw. Anhang D.2) bietet sich die direkte Weiterverarbeitung als offene Komponenten an, die wir im nächsten Abschnitt anhand von XEC vorstellen.

7.5.2 Getrennte Implementierung der Systemkomponenten mit XEC

Die Implementierung der durch die jeweiligen PET-Objektdateien beschriebenen Systemkomponenten erfolgt bei XEC völlig separat bis hin zur Erzeugung der Maschinen-Objektdateien (siehe Abb. 7-25). Zur Bildung eines ausführbaren Systems muss daher erst zum Zeitpunkt des Zusammenbindens der Teilkomponenten mit dem Linker eine konkrete Kombination von offenen Systemen und Umgebungen ausgewählt werden. Wir werden später sehen, dass dies eine erhebliche Beschleunigung der Übersetzungszyklen erlaubt.

---

83. z. B. `pm` in „`BEHAVIOUR pm1 FOR pm;`“

84. Die Definitionen in den Flex- und Bison-Quellen konnten aufgrund syntaktischer Beschränkungen dieser Werkzeuge nicht vollständig eingeklammert werden. Die entsprechenden Erweiterungen werden jedoch bei Deaktivierung von OPEN_EXT indirekt deaktiviert.

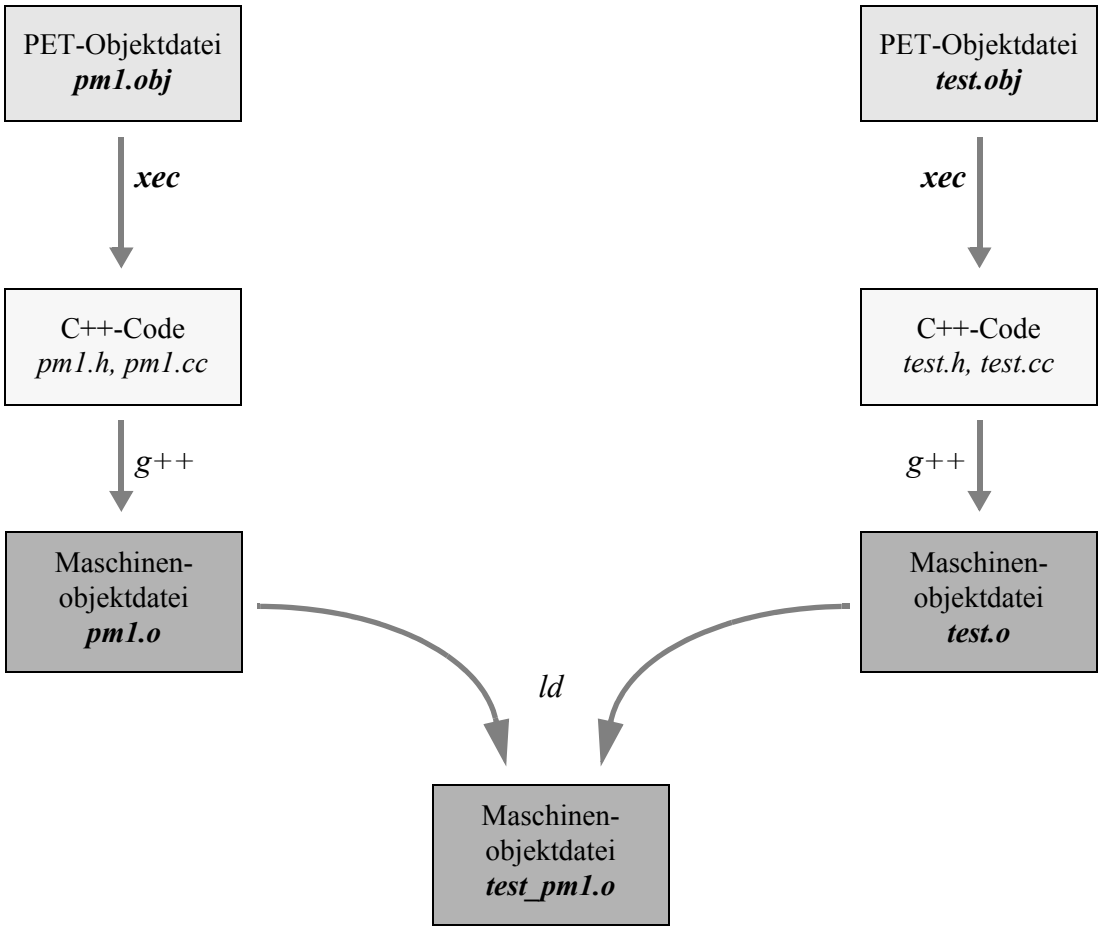


Abbildung 7-25: Getrennte Implementierung der offenen Teilsysteme durch XEC

Um diese getrennte Übersetzbarkeit realisieren zu können, muss bei der Generierung des C++-Codes durch XEC sichergestellt sein, dass die erzeugten Quelltexte alle zur späteren Übersetzung durch den C++-Compiler (hier g++) erforderlichen Deklarationen beinhalten. Dies schließt auch sämtliche aus Interfaces importierten Deklarationen ein, da diese von den zu implementierenden Systemteilen referenziert und entsprechend auch zur Grundlage des generierten C++-Codes werden.

Umgekehrt müssen bei der Übersetzung der von den entsprechenden Behaviour-Definitionen exportierten Definitionen analoge Deklarationen bereitstehen, da nur so am Ende zueinander passende und vom Linker zusammenfügbare Implementierungsteile entstehen.

An dieser Stelle greift das bereits in Kapitel 3 eingeführte *deterministische Kodierungsschema* von XEC. Zusammen mit der Open-Estelle zu Grunde liegenden syntaktischen Eindeutigkeit einer Interface-Definition (unabhängig von einer importierenden Umgebung, s. Abschnitt 7.2.1) ermöglicht sie, aus der selben Interface-Definition unabhängig vom Kontext immer die selben C++-Deklarationen zu erzeugen. Dies schließt die in den PET-Objektdateien enthaltenen importierten Interface-Definitionen ein.[85]

Dazu werden bei der Codegenerierung aus einer PET-Objektdatei zu allen importierten Interface-Definitionen in den „`.h`“-Dateien vollständige C++-Klassendeklarationen generiert. Diese dienen als Signatur des offenen Systems auf Implementierungsebene zur Auflösung aller Referenzen auf das offene System. Die notwendigen Klassendefinitionen zu diesen Deklarationen werden nur bei der Codegenerierung des entsprechenden offenen Systems erzeugt.[86]

Besondere Bedeutung kommt hier der Instanziierung eines importierten offenen Systems innerhalb einer importierenden Umgebung zu. Dabei muss auf Implementierungsebene in der importierenden Umgebung eine Klasseninstanz erzeugt werden, deren Deklaration nur in einem anderen Implementierungszweig (dem der Behaviour-Definition des importierten Interfaces) bekannt ist.

Ursache dafür ist, dass aus Sicht der importierenden Umgebung von dem offenen System nur die externe Schnittstelle (d.h. der Name und mit dem Modulheader die Signatur) bekannt ist. Wie wir in Abschnitt 3.3.2 gesehen haben, wird das offene System (also der Modul-Body) als eine davon abgeleitete Klasse implementiert, welche die Deklarationen und Definitionen aus dem Inneren des Bodies als Komponenten und lokale Definitionen enthält.

Zum Zeitpunkt der Übersetzung der importierenden Umgebung sind jedoch diese internen Aspekte des offenen Systems noch nicht bekannt. Mehr noch ist nicht einmal der Name der Behaviour-Definition (und damit der implementierenden C++-Klasse) bekannt, da diese erst beim Zusammenfügen der Teilsysteme festgelegt wird.[87] Entsprechend ist eine direkte Instanziierung des offenen Systems (z. B. per „`new`“-Operator) hier nicht möglich.

Die Lösung besteht darin, für jedes offene System eine statische Konstruktor-Methode in der Interface-Klasse zu deklarieren, die dann von dem importierenden System aufgerufen werden kann. Diese liefert eine dynamisch erzeugte Instanz des offenen Systems in Form eines Zeigers auf die Modulheader-Klasse, welche die Signatur des offenen Systems bildet (siehe Abschnitt 3.3.2). Dieses Konstrukt, das letztlich einem *„Factory Method Pattern“* [Gam95] entspricht, bietet die geforderte Abstraktion und stellt so das Bindeglied zwischen der Deklaration und Nutzung eines offenen Systems und einer dazu passenden Definition dar.

Wir illustrieren dies anhand des in Abschnitt 7.2 (bzw. Anhang D) eingeführten Interfaces „`binaryService`“ mit dem offenen System „`binaryOperator`“ (siehe Abb. 7-8 auf Seite 325) und der zughörigen Behaviour-Definition „`binaryAdder`“, die eine konkrete Definition zum offenen System „`binaryOperator`“ liefert (siehe Abb. 7-10 auf Seite 327). Hier wird als Schnittstelle zwischen dem offenen System und der importierenden Umgebung die statische Methode `INTERFACE_binaryservice::BODY_binaryoperator_NEW` definiert:

Beispiel 7.65: Statische Konstruktor-Methode für offenes System

Deklaration in „`binaryAdder.h`“ und „`test.h`“:

```
12: class INTERFACE_binaryservice {
    // ...
80:         static HEADER_binaryoperatorheader*
                    BODY_binaryoperator_NEW(Module* pParent);
81: };
```

85. Es ist zu beachten, dass die Interface-Objektdateien nach der Erzeugung der importierenden Spezifikations- bzw. Behaviour-Objektdateien dort integriert werden und nicht weiter benötigt werden (siehe Abb. 7-24). Entsprechend gibt es zwischen den einzelnen Implementierungszweigen ausgehend von jeweils einer einzelnen PET-Objektdatei keine Querbezüge zu anderen Implementierungszweigen (siehe Abb. 7-25).

86. Dies entspricht dem typischen Bibliotheks-Header-Konzept von C und C++, mit dem Unterschied, dass die Headerdefinitionen in mehreren „`.h`“-Dateien (jedoch identisch) vorliegen. Diese Replikation ist bei vollständig getrennten Implementierungszweigen unumgänglich.

87. zur Erinnerung: Zu einer Interface-Definition kann es diverse verschiedene Behaviour-Definitionen geben, die lediglich Bezug auf die Interface-Definition nehmen (siehe Abschnitt 7.2.2).

Definition mit Zuordnung des offenen Systems in „`binaryAdder.cc`“:

```
73: INTERFACE_binaryservice::HEADER_binaryoperatorheader*
        INTERFACE_binaryservice::BODY_binaryoperator_NEW
            (Module* pParent)
        {
74:         return BEHAVIOUR_binaryadder::BODY_binaryoperator::
                                staticData.newInstance(pParent);
75:     }
```

(Ende von Beispiel 7.65)

Der Aufruf der Methode „`staticData.newInstance`“ ersetzt den direkten `new`-Operator zugunsten des Zugriffs auf den Modulinstanzcache, welcher die geforderte Instanz neu erzeugt oder eine frühere Instanz wiederverwendet (siehe Abschnitt 3.5.4.1).

Auf Basis der dargestellten Generierungstechnik waren lediglich noch einige kleinere Anpassungen von XEC zur Implementierung der Open-Estelle-Erweiterung erforderlich. Diese sind wie auch bei den o. g. PET-Erweiterungen anhand der bedingten Kompilierung („`#ifdef OPEN_EXT`“) leicht erkennbar. Im Wesentlichen betreffen sie die besonderen Sichtbarkeits- und Attributierungsregeln von Open-Estelle. Letztere erfordern eine dynamische Bestimmung des effektiven System-Attributs einer Modulbody-Klasse, da u. U. die selbe Klassendefinition in unattributierten und attributierten Umgebungen instanziiert werden kann und dabei ein Systemattribut erhält oder nicht (siehe Abschnitt 7.2.4). Dies erfolgt zum Instanziierungszeitpunkt durch Abfrage der Attributierung des Vatermoduls.

7.5.2.1 Dynamische Bindung

Eine nahe liegende Erweiterung des Implementierungsmodells besteht in der Realisierung der offenen Teilsysteme als dynamisch ladbare Bibliotheken. Dies könnte es ermöglichen, erst zum Zeitpunkt des Systemstarts die Teilkomponenten festzulegen, die dann dynamisch geladen werden.

In diesem Fall müsste die jeweilige Auswahl einer (aus einer Behaviour-Definition erzeugten) dynamischen Bibliothek (die ein offenes System darstellt) und eines (aus einer Specification erzeugten) Programms als oberste importierende Umgebung geeignet parametriert werden.[88] Bei einer dynamisch vom Programm gesteuerten Bindung der Teilsysteme[89] könnte die Zuordnung sogar erst zur Laufzeit erfolgen. Eine Dynamisierung des oben beschriebenen Interface-Behaviour-Zuordnungsmechanismus (siehe Beispiel 7.65) könnte sogar dazu genutzt werden, gleichzeitig verschiedene Behaviour-Definitionen zum selben Interface einzusetzen.

Eine derartige dynamische Bindung wird in einer späteren Version von XEC realisiert werden.

88. Sie erfolgt bisher beim manuellen Link-Vorgang.

89. z. B. per „`dlopen(...)`“ unter Solaris und Linux

7.5.2.2 Heterogene Timescale-Einheiten

Zuletzt kommen wir noch auf die bereits in Abschnitt 7.4.3 angesprochene Möglichkeit unterschiedlicher Timescales bei der Kombination separat entwickelter Teilsysteme zurück. Bei der oben vorgestellten Implementierung durch XEC wird diese Problematik dadurch entschärft, dass von Hause aus nur bestimmte *realzeitorientierte* Timescale-Einheiten (z. B. `Seconds`, `Milliseconds`, `Microseconds`) vom XEC-Laufzeitsystem unterstützt werden (siehe auch Abschnitt 3.3.7.7). Die mit diesen Einheiten angegebenen Zeitspannen werden intern in ein einheitliches Zeitformat (Mikrosekunden) umgerechnet, so dass prinzipiell sogar jede `DELAY`-Klausel eine eigene Zeitskala festlegen könnte. Somit ist eine konsistente Bewertung der Zeitangaben und ein konsistentes globales Zeitmodell sichergestellt.

7.5.3 Effizienz des Implementierungsvorgangs

Zu Beginn unserer Betrachtungen über effiziente Implementierung (Kapitel 4) haben wir verschiedene Optimierungsziele im Zusammenhang mit automatisch generierten Implementierungen identifiziert. Eines dieser Ziele war die Effizienz des Implementierungsvorganges selbst. Ein wichtiger Faktor ist dabei die *Turn-Around-Zeit* bei der Generierung von Implementierungen, also die Zeit von einer Änderung in der Ausgangsspezifikation bis zum Bereitstehen einer neuen Implementierung.

Prinzipbedingt hat die Estelle-Erweiterung Open-Estelle – anders als die meisten anderen in diesem Text vorgestellten Spezifikations- und Implementierungstechniken – keinen direkten Einfluss auf die Performance der erzeugten Implementierung. Wir werden jedoch sehen, dass Open-Estelle einen sehr starken Einfluss auf die Effizienz des Implementierungsvorganges selbst haben kann (siehe auch [ThGo98b]).

Betrachten wir dazu die in Abb. 7-26 dargestellte getrennte Implementierung der XTP-Protokollmaschine (`xtp.stl`, `xtp.sti`) und einem Testszenario (`test.stl`). Wir vergleichen nun die zur Implementierung der Einzelkomponenten erforderlichen Zeiten mit der Implementierung einer äquivalenten geschlossenen Spezifikation des Szenarios. Wir schlüsseln den Zeitbedarf[90] gemäß den einzelnen erforderlichen Operationen auf:

Operation	Standard-Estelle	Open-Estelle	
		xtp-pm	xtp-test
1. pet (Interface)	–	0,02 s	
2. pet	0,41 s	0,43 s	0,03 s
3. xec	0,40 s	0,40 s	0,01 s
4. g++ (Übersetzung)	12,84 s	12,39 s	0,54 s
5. g++ (Binden)	0,32 s	0,33 s	

Tabelle 7.31: Zeitbedarf für Übersetzungsschritte Standard-/Open-Estelle

90. Plattform: Intel P4, 1x 2,524 GHz CPU, 1 GByte RAM, Debian Linux, Kernel 2.4.26, gcc 2.95.4, pet 2.01, xec 1.3.3(+) mit „`-m optimize`“

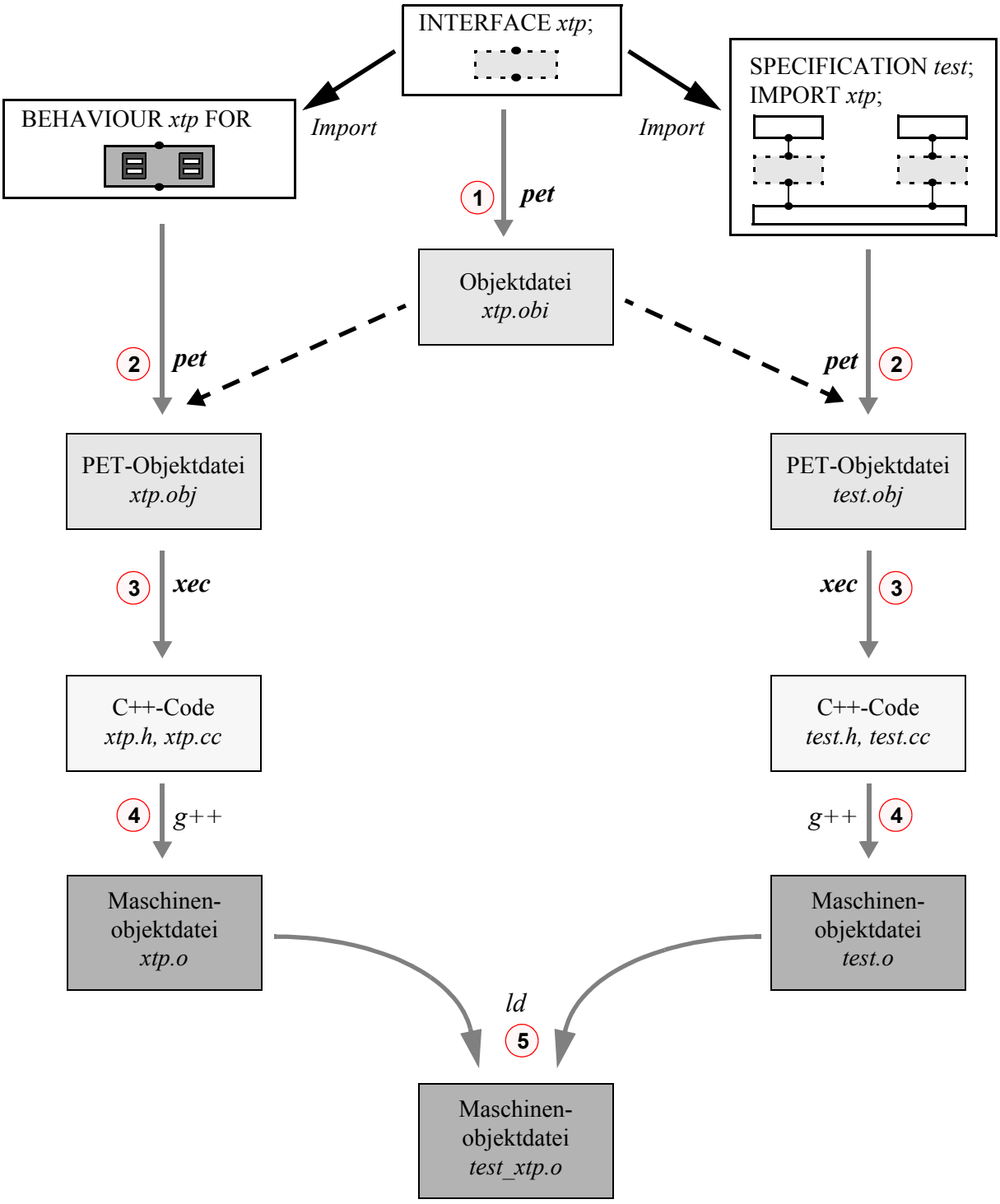


Abbildung 7-26: Performancevergleich beim Implementierungsvorgang

Die im Entwicklungsprozess effektiv auftretenden Turn-Around-Zeiten hängen bei dem in Open-Estelle spezifizierten System von den Änderungen ab, die zur Neuübersetzung führen. Wir unterscheiden hier

- Komplette Neuübersetzung (z. B. nach Änderungen an der Interface-Definition)
- Übersetzung nach einer Änderung von `xtp.stl`
- Übersetzung nach einer Änderung von `test.stl`

Während bei der geschlossenen Spezifikation in allen Fällen die selben Übersetzungszeiten anfallen (schließlich muss immer die gesamte Spezifikation von Grund auf neu übersetzt werden), variieren die Zeiten bei dem mit Open-Estelle spezifizierten System erheblich:

Zeitbedarf[a] für ...	Standard-Estelle	Open-Estelle	Speedup
Komplettübersetzung	13,97 s	14,15 s	0,99
Übersetzung nach Änderung an `xtp.stl`	13,97 s	13,57 s	1,03
Übersetzung nach Änderung an `test.stl`	13,97 s	0,93 s	15,02

a. Alle Zeitangaben haben eine Genauigkeit von ± (0,03 s + 1%)

Im direkten Vergleich wird deutlich, dass bei Änderungen am Testszenario der xtp-Protokollmaschine (`test.stl`) aufgrund der Tatsache, dass die komplexe Protokollmaschine nicht erneut implementiert werden muss, erhebliche Zeiteinsparungen durch Open-Estelle möglich sind. Während die übrigen Fälle zu praktisch identischen Turn-Around-Zeiten führen, wird hier ein *Speedup* von ca. 15 erreicht. Dies erweist sich besonders bei Experimenten mit Anwendungsbeispielen von xtp als vorteilhaft, da hier normalerweise keine Änderungen an der Protokollmaschine oder gar ihrem externen Interface erforderlich sind, zum Evaluieren verschiedener Protokollfunktionalitäten aber gerade bei so flexiblen Protokollen wie XTP oft diverse Testszenarien durchgespielt werden müssen.

In komplexeren Szenarien, in denen z. B. auch verschiedene Protokollmaschinen beteiligt sind, kann der Gewinn sogar noch weitaus höher ausfallen. Aber auch bei Entwicklungsarbeiten an einer einzelnen großen Protokollmaschine können durch eine Untergliederung der Protokollmaschine in mehrere offene Teilsysteme die Entwicklungszeiten verkürzt werden.

Diese Grundidee lässt sich noch wirksamer umsetzen, wenn Spezifikationsteile (z. B. ein kompletter Dienst oder eine Protokollmaschine wie XTP) ähnlich zu *Programmierbibliotheken* bei imperativen Programmiersprachen als fertig übersetzte Komponenten im System vorliegen und praktisch gar nicht mehr individuell übersetzt werden müssen. Die Fragestellung der dazu erforderlichen *Generizität von Protokollkomponenten* untersuchen wir im nächsten Abschnitt.

7.6. Wiederverwendung und Generizität

Zum Abschluss unserer Betrachtungen über Open-Estelle wenden wir uns nochmals einer der Hauptmotivationen der Entwicklung von Open-Estelle zu, nämlich der *Wiederverwendung* von Spezifikationsteilen bzw. der Entwicklung *generischer* Protokollkomponenten.

Wie wir bereits in Kapitel 5 gesehen haben, führt die strenge Typsicherheit von Standard-Estelle zu einer extrem starken Prägung der äußeren Schnittstelle von Dienst- und Protokollmaschinenspezifikationen auf konkrete Nutzdatentypen bzw. (interne) PDU-Typen. Sie beeinträchtigt die Austauschbarkeit von solchen Protokollkomponenten erheblich.

War diese Beeinträchtigung in einfachen (geschlossenen) Spezifikationen auf Grund der meist vorzufindenden Spezialisierung des gesamten Szenarios auf eine konkrete Kombination von Protokollkomponenten möglicherweise zunächst nicht offensichtlich, so erwächst beim Versuch einer *anwendungsunabhängigen Spezifikation von Protokollkomponenten* aus dieser starken Typprägung ein mit den Mitteln von Standard-Estelle unlösbares Problem. So kann zum Beispiel in Standard-Estelle kein abstrakter Datentransportdienst spezifiziert werden, der jeden beliebigen Nutzdatentyp von einem Interaktionspunkt zu einem anderen transportiert (siehe auch Beispiel 5.32 auf Seite 189).

In Kapitel 5 haben wir zur Lösung dieses Problems die Containertyp-Erweiterung am Beispiel von Estelle eingeführt. Die aus ihr resultierende Möglichkeit zur Spezifikation und Kopplung von Nutzdaten- und PDU-typunabhängigen Protokollkomponenten (insbesondere Modulen) löst dieses Problem auf elegante und semantisch präzise Weise. Offensichtlich tritt der Nutzen der Containertyp-Erweiterung jedoch erst bei der Spezifikation von wiederverwendbaren offenen Systemen voll zu Tage, und in der Tat ging die Motivation zur Entwicklung der Containertyp-Erweiterung aus unseren Fallstudien mit der Open-Estelle-Erweiterung hervor.

Beide Erweiterungen, die Containertyp-Erweiterung und die Open-Estelle-Erweiterung, wurden so konzipiert, dass sie problemlos miteinander *kombiniert* werden können. So würde man zum Beispiel zur Spezifikation eines abstrakten Datentransportdienstes in Form eines offenen Systems einfach als zu transportierenden Nutzdatentyp den Containertyp einsetzen. Der resultierende Dienst, der im Beispiel 5.37 auf Seite 201 bereits als Teil einer geschlossenen Spezifikation skizziert wurde, kann leicht in die folgende Interface- und Behaviour-Spezifikation dekomponiert werden.

Zunächst wird in den folgenden Beispielen der Kanal `simple_send_receive_ch` als Typ der Dienstzugangspunkte des späteren abstrakten Datentransportdienstes definiert. Diese Definition erfolgt in einer separaten Interface-Definition gleichen Namens (Beispiel 7.66-a), da wir später von der Kanaldefinition noch an anderer Stelle Gebrauch machen werden (s. u.).

Beispiel 7.66-a: Interface-Definition „`simple_send_recv_ch.sti`“

```
INTERFACE simple_send_receive_ch; (* ssr *)
        TYPE SDU = ANY TYPE;
        CHANNEL ssr_ch(User,Provider);
                BY User:        D_Send(Data: SDU);
                BY Provider:    D_Recv(Data: SDU);
END.
```

(Ende von Beispiel 7.66-a)

Das externe Interface des eigentlichen offenen Systems (Beispiel 7.66-b) importiert diese Kanaldefinition und deklariert darauf aufbauend den Datentransportdienst in Form des Bodies `ats_bod`.

Beispiel 7.66-b: Interface-Definition „`abstract_transport_service.sti`“

```
INTERFACE abstract_transport_service; (* ats *)
        IMPORT simple_send_receive_ch;

        MODULE ats_mod ACTIVITY;
                IP IpToUser: ARRAY [1..2]
                        OF simple_send_receive_ch::ssr_ch(Provider)
                            COMMON QUEUE;
                END;

        BODY ats_bod FOR ats_mod;
                EXTERNAL;
END.
```

(Ende von Beispiel 7.66-b)

Die interne Definition des Bodies erfolgt in der entsprechenden Behaviour-Definition (Beispiel 7.66-c).

Beispiel 7.66-c: Behaviour-Definition „`abstract_transport_service.stl`“

```
BEHAVIOUR abstract_transport_service FOR abstract_transport_service;
        BODY ats_bod FOR abstract_transport_service::ats_mod;
                TRANS
                        ANY i: 1..2 DO
                        WHEN IpToUser[i].D_Send(Data)
                                BEGIN
                                OUTPUT IpToUser[3-i].D_Recv(Data);
                                END;
        END; (* BODY *)
END.
```

(Ende von Beispiel 7.66-c)

Ein Dienstnutzer, der auf diesem abstrakten Transportdienst aufsetzt, muss an seiner „unteren“ Schnittstelle einen kompatiblen Interaktionspunkt besitzen und dementsprechend die Kanaldefinitionen der Schnittstelle importieren. Um die Abhängigkeiten der aufeinander aufbauenden importierten Definitionen präziser modellieren zu können, haben wir im vorangegangenen Beispiel diese Kanaldefinition in ein separates Interface verlegt. Dadurch können wir im nächsten Beispiel einer Protokollmaschine, die den angebotenen Dienst des abstrakten Transportdienstes verfeinert,[91] ohne Bezug auf diesen Dienst selbst spezifizieren. Lediglich die Kanaldefinition als gemeinsame Schnittstelle wird importiert (siehe auch Abb. 7-13 auf Seite 329).

91. Mit Verfeinerung einer Dienstschnittstelle ist hier ein Aufsetzen einer Protokollschicht (hier also zweier Protokollmaschineninstanzen) gemeint, die syntaktisch nach oben die selbe (oder eine vererbte, s. u.) Schnittstelle wie der darunter liegende Dienst bietet. Die semantische Verfeinerung stellt dabei typischerweise eine „Verbesserung“ des gebotenen Dienstes dar, z. B. die Eliminierung von verfälschten Paketen oder die Kompensation von Paketverlusten durch den zu Grunde liegenden, in dieser Hinsicht „schlechteren“ Dienst.

Beispiel 7.66-d: Interface-Definition „simple_send_receive_pm.sti“

```
INTERFACE simple_send_receive_pm; (* ssr *)
        IMPORT simple_send_receive_ch;

        MODULE ssr_mod ACTIVITY;
                IP IpToUser: simple_send_receive_ch::ssr_ch(Provider)
                                COMMON QUEUE;
                   IpToProvider: simple_send_receive_ch::ssr_ch(User)
                                COMMON QUEUE;
                END;

        BODY ssr_bod FOR ssr_mod;
                EXTERNAL;
END.
```

(Ende von Beispiel 7.66-d)

Der PDU-Typ dieser Protokollmaschine wird erst in ihrer Behaviour-Definition (bzw. möglicherweise diversen Behaviour-Definitionen) spezifiziert und bleibt somit ein rein interner Aspekt, der nach außen nicht sichtbar wird:

Beispiel 7.66-e: Behaviour-Definition „example_pm.stl“ (Auszug)

```
BEHAVIOUR example_pm FOR simple_send_receive_pm;
        BODY ssr_bod FOR simple_send_receive_pm::ssr_mod;
                TYPE PDU = (* ... *);
                (* ... *)
        END; (* BODY *)
END.
```

(Ende von Beispiel 7.66-e)

Bei der Spezifikation solcher Protokollmaschinen erweist sich diese Fähigkeit der Containertyp-Erweiterung als sehr nützlich, den PDU-Typ des spezifizierten Protokolls als rein internen Aspekt des Protokollmaschinenmoduls und damit auch des offenen Systems spezifizieren zu können. Dies macht die äußere Schnittstelle der Protokollmaschine als offenes System vollständig unabhängig von derartigen internen Aspekten und erlaubt so nicht zuletzt den Einsatz verschiedener Protokolle und Protokollmaschinen im selben Szenario, ohne an anderen Komponenten Änderungen vornehmen zu müssen.

So könnten die im Folgenden skizzierten Protokollmaschinen zum Beispiel einmal das Alternating-Bit-Protokoll (AB-Protokoll, Beispiel 7.66-f) und ein anderes Mal ein Sliding-Window-Protokoll (Sliding-Window-Protokoll, Beispiel 7.66-g) implementieren. Zwar benutzen beide Protokollmaschinen unterschiedliche PDU-Typen, jedoch werden diese an der (identischen) äußeren Schnittstelle nicht sichtbar. Entsprechend können die beiden Protokollmaschinen beliebig ausgetauscht werden, solange nur die jeweils miteinander kommunizierenden Protokollmaschineninstanzen in dieser Hinsicht kompatibel bleiben. Dieser Austausch kann, wie wir in Abschnitt 6.5 gezeigt haben, sogar *einfach durch neues Binden* der ansonsten bereits fertig übersetzten Teilkomponenten des Kommunikationsszenarios zu einem neuen ausführbaren Programm erfolgen.

Beispiel 7.66-f: Behaviour-Definition „ab_protocol_pm.stl“ (Auszug)

```
BEHAVIOUR ab_protocol_pm FOR simple_send_receive_pm;
        BODY ssr_bod FOR simple_send_receive_pm::ssr_mod;
                IMPORT simple_send_receive_ch;

                TYPE PDU = RECORD (* AB-Protokoll-PDU *)
                        ab_flag: (a, b);
                        payload: simple_send_receive_ch::SDU;
                        END; (* RECORD *)
                (* ... *)
        END; (* BODY *)
END.
```

(Ende von Beispiel 7.66-f)

Beispiel 7.66-g: Behaviour-Definition „sliding_window_pm.stl“ (Auszug)

```
BEHAVIOUR sliding_window_pm FOR simple_send_receive_pm;
        BODY ssr_bod FOR simple_send_receive_pm::ssr_mod;
                IMPORT simple_send_receive_ch;

                CONST seq_max = 65635;
                TYPE PDU = RECORD (* Sliding-Window-Protokoll-PDU *)
                        seq: 0..seq_max;
                        payload: simple_send_receive_ch::SDU;
                        END; (* RECORD *)
                (* ... *)
        END; (* BODY *)
END.
```

Eine interessante Besonderheit in den obigen Beispielen ist die Tatsache, dass die obere Schnittstelle der Protokollmaschine und ihre untere Schnittstelle identisch sind. Dies tritt gerade bei derartigen einfachen Dienstschnittstellen, die z. B. lediglich Sende- und Empfangsnachrichten mit einem unspezifischen Nutzdatum austauschen, häufiger auf. So könnte unterhalb der oben definierten Protokollschicht, die per AB- oder Sliding-Window-Protokoll Paketverluste des zu Grunde liegenden Dienstes kompensiert, auch z. B. mit den gleichen externen Schnittstellen eine weitere Protokollschicht eingezogen werden, die mit Hilfe von Prüfsummen Paketverfälschungen eines darunter liegenden Dienstes erkennt und behandelt. Diese könnte mit den gleichen oberen und unteren Schnittstellen versehen werden, und auch hier wäre der PDU-Typ (wie z. B. die Prüfsummenlänge) ein rein interner Aspekt der jeweiligen Behaviour-Definition.

Dieser Aspekt der *Schnittstellengenerizität* gewinnt mit zunehmender Vereinfachung und der damit einhergehenden Vielzahl von Protokollkomponenten und Einzelschnittstellen an Bedeutung. Dabei ist insbesondere im Bereich der *Mikroprotokolle* aufgrund der Forderungen nach flexibler Konfigurierbarkeit, Austauschbarkeit und (architektureller) Offenheit eine individuelle Anpassung zwischen den einzelnen Protokollkomponenten nur bedingt möglich. Hier ist eine weitestmögliche Normierung und Verallgemeinerung der Schnittstellen, wie wir sie durch die Kombination von Containertyp-Erweiterung und Open-Estelle erreicht haben, wünschenswert ([GKS03], [Fli+04]).

Komplexere Dienste, die zum Beispiel Zusatzparameter wie Zieladressen bis hin zu komplexen Quality-of-Service-Parametern beinhalten, machen andererseits u. U. spezifische Schnittstellendefinitionen erforderlich. Hier wäre möglicherweise ein Vererbungsmechanismus bei der Spezifikation von Kommunikationsschnittstellen eine interessante Erweiterungsmöglichkeit für Open-Estelle, die jedoch mit erheblichen konzeptionellen und technischen Anpassungen der FDT Estelle verbunden wäre. Wir verzichten auf eine tiefere Diskussion dieses Ansatzes und beschließen damit die Ausführungen zu Open-Estelle.

7.7. Zusammenfassung

Estelle-Spezifikationen beschreiben „*abgeschlossene Welten*“ in dem Sinne, dass die beschriebenen Systeme keine Möglichkeit zur Kommunikation mit einer Umgebung haben. Dies bedeutet, dass *offene Systeme* (z.B. eine Protokollmaschine) nur *als Teil einer geschlossenen Estelle-Spezifikation* beschrieben werden können. Dadurch ist jedoch sowohl die syntaktische wie auch die semantische Interpretation des offenen Systems nur im Kontext genau dieser Umgebung möglich. Pragmatische Lösungsansätze auf Basis textuellen Imports von Spezifikationsfragmenten besitzen jedoch weder eine formale (also eindeutige) Syntax, noch eine solche Semantik und erweisen sich somit als ungeeignet.

Zur Lösung dieser Problematik haben wir mit *Open-Estelle* eine syntaktische und semantische Erweiterung von (Standard-) Estelle eingeführt. Open-Estelle definiert ein offenes System in zwei getrennten textuellen Einheiten, die seine externe Schnittstelle (*Interface-Definition*) bzw. seine internen Aspekte (*Behaviour-Definition*) festlegen. Die Interface-Definition besitzt dabei eine eindeutige syntaktische Interpretation, die unabhängig von den potentiellen importierenden Umgebungen ist. Der Zugriff auf ein offenes System durch eine *importierende Umgebung* erfolgt über den *formalen Import* der Definitionen aus der Interface-Definition. Die importierten Namen werden dabei qualifiziert sichtbar und die benannten Objekte damit referenzierbar (z. B. „`xtp::PDU_type`“).

Eine Interface-Definition *deklariert* eine Menge von offenen Systemen in Form von unvollständigen Moduldefinitionen („`EXTERNAL`“-attributiert). Diese können von den importierenden Umgebungen wie lokale Moduldefinitionen genutzt werden, u. a. um Instanzen der offenen Systeme (also Modulinstanzen) zu erzeugen. Die Vervollständigung einer solchen importierenden Umgebung erfolgt allein durch das *Zuordnen* einer dazu passenden Behaviour-Definition, die zu jedem der in ihrer Interface-Definition deklarierten offenen System jeweils eine passende Moduldefinition und somit ein *konkretes Verhalten* des offenen Systems spezifiziert. Diese *logische Zuordnung* erfolgt *auf Metaebene* und ermöglicht die freie *Komposition* verschiedener Behaviour-Definitionen und importierender Umgebungen, sofern sie sich beide auf die selbe Interface-Definition beziehen.

Aufgrund der syntaktischen Fundierung besitzt eine importierende Umgebung (sofern sie eine Spezifikation ist) bereits zusammen mit dem importierten offenen System eine *Standard-Estelle-Semantik*. Zusätzlich wurde eine Semantik für offene Systeme definiert, indem jeder über die externen Schnittstellen mögliche Stimulus berücksichtigt wird. Die Menge der sich daraus ergebenden (potentiellen) Berechnungen wird dann bei der Integration des offenen Systems in eine konkrete Umgebung auf die Wirkung der von dieser tatsächlich erzeugten Stimuli reduziert.

Ein Schwachpunkt dieser Semantik ist ihre mangelnde *Kompositionsverträglichkeit*, da die Semantik eines Gesamtsystems nicht aus der Komposition der Semantiken von Teilsystemen gewonnen werden kann. Insbesondere haben importierende Umgebungen ohne ein konkretes offenes System überhaupt keine (offene) Semantik. Wir haben zur Beseitigung dieses Defizits einige Ansätze zur Gewinnung einer geeigneten kompositionsverträglichen Semantik diskutiert, wobei jedoch die Frage, ob diese (mit vertretbarem Aufwand) kompatibel zur Standard-Estelle-Semantik realisierbar sind, offen bleibt.

Das Konzept der Komposition von kompatiblen Teilsystemen (die jedes für sich bereits eine formale Syntax besitzen) zu einer vollständigen Spezifikation setzt sich auch bei der Implementierung durch XEC fort: Offene Systeme und importierende Spezifikationen können mit Hilfe

von XEC getrennt voneinander bis auf Maschinen-Objektdateiebene implementiert werden. Erst beim Binden zu einer ausführbaren Anwendung müssen die Teilsysteme zusammengestellt werden.

Weiterhin wurde ein Werkzeug vorgestellt, das gemäß der syntaktischen Fundierung von Open-Estelle offene Systeme und importierende Spezifikationen zu syntaktisch äquivalenten geschlossenen Standard-Estelle-Spezifikationen verschmelzen kann.

Die volle Leistungsfähigkeit des Konzeptes von Open-Estelle erschließt sich jedoch erst in Kombination mit den in Kapitel 5 eingeführten Containertypen, da mit ihrer Hilfe Nutzdatentyp-neutrale und somit generische Kommunikationskomponenten als offene Systeme spezifiziert werden können. Dies ist eine essentielle Grundlage für die Wiederverwendung solcher Komponeten.

Die untersuchte Problemstellung der mangelnden Möglichkeiten zur formalen Definition *offener Systeme* und *ihrer Komposition* stellt sich in ähnlicher Form auch in anderen formalen Beschreibungstechniken. So unterstützt z. B. SDL [ITU94] zwar die syntaktische Spezifikation offener Systeme (Kanäle können mit der Systemgrenze verbunden werden) und diese besitzen auch eine formale Semantik, die jedoch keinen Signalaustausch mit der Umwelt beinhaltet. Auch die Komposition solcher offener Systeme wird weder syntaktisch noch semantisch unterstützt.

Die Übertragung der hier anhand von Estelle demonstrierten Konzepte scheint durchaus realisierbar und könnte die Ausdrucksfähigkeit von SDL um die Fähigkeit zur formalen Spezifikation offener Systeme und ihrer Komposition erweitern. Auch die Kombination mit den ebenfalls auf SDL übertragbaren Containertypen erscheint auf dieser Basis realisierbar und im Sinne eines komponentenbasierten Wiederverwendungsansatzes wünschenswert.

8. Zusammenfassung und Ausblick

Wir fassen nun die Ergebnisse anhand unserer Ausgangsmotivationen, der *Steigerung der Ausdrucksfähigkeit* (Abschnitt 8.1) und der *effizienten Implementierbarkeit* formaler Beschreibungstechniken (Abschnitt 8.2) zusammen und schließen dann mit einem Ausblick auf weitere Forschungsziele (Abschnitt 8.3).

8.1. Ausdrucksfähigkeit

Die formale Spezifikation von Kommunikationssystemen stellt durch die mit ihr verbundene *Abstraktion* und *Präzision* eine wichtige Grundlage für die formale Verifikation von Systemeigenschaften dar. Diese Abstraktion begrenzt jedoch auch die *Ausdrucksfähigkeit* der formalen Beschreibungstechnik und kann somit zu problemunangemessenen Spezifikationen führen.

Wir haben in dieser Arbeit anhand der formalen Beschreibungstechnik Estelle zwei solche Aspekte untersucht. Beide führen speziell in Hinsicht auf die Domäne von Estelle, der Spezifikation von Kommunikationsprotokollen, zu schwerwiegenden Beeinträchtigungen der Ausdrucksfähigkeit, durch die eine problemangemessene Spezifikation teilweise unmöglich wird.

Eines dieser Defizite zeigt sich bei dem Versuch, in Estelle ein *offenes System* wie z. B. eine Protokollmaschine oder einen Kommunikationsdienst zu spezifizieren. Da Estelle-Spezifikationen nur geschlossene Systeme beschreiben können, werden solche Komponenten immer nur als Teil einer fest vorgegebenen Umgebung spezifiziert und besitzen auch nur in dieser eine formale Syntax und Semantik. Überträgt man nun (textuell) das Spezifikationsfragment dieses offenen Systems in eine andere Umgebungsspezifikation, so hat es dort eine neue syntaktische Interpretation, die in keiner festen Beziehung zur vorherigen steht. Solche offenen Systeme haben also keine formale Syntax und somit auch keine formale Semantik. Als Lösung für dieses Problem haben wir in Kapitel 7 die kompatible syntaktische und semantische Estelle-Erweiterung *Open-Estelle* eingeführt, die eine formale Spezifikation solcher offener Systeme und ihres Imports in verschiedene Umgebungen ermöglicht.

Ein anderes Defizit in der Ausdrucksfähigkeit von Estelle ergibt sich aus der strengen Typprüfung. Wir konnten zeigen, dass es in heterogenen, hierarchisch strukturierten Kommunikationssystemen durch die dort auftretenden *horizontalen* und *vertikalen Typkompositionen* zu einer unangemessenen Modellierung von Nutzdatentypen an den Dienstschnittstellen kommt. Dieses Problem erweist sich beim Versuch einer generischen und nutzdatentypunabhängigen Spezifikation eines offenen Systems (z. B. mit Open-Estelle) sogar als fatal. Estelle erlaubt auch hier keine problemangemessene Modellierung solcher Fragestellungen. Deshalb führten wir in Kapitel 5 die kompatible *Containertyp-Erweiterung* ein, durch die eine formale Spezifikation

nutzdatentypunabhängiger und somit *generischer Schnittstellen* von Diensten und Protokollmaschinen ermöglicht wird. Insbesondere ermöglicht es diese Erweiterung, PDU-Typen von Protokollmaschinen als deren internes Geheimnis zu modellieren.

Beide Erweiterungen wurden auf Basis unseres Estelle-Implementierungsgenerators XEC (s. u.) implementiert und anhand von Fallstudien untersucht.

Beide Erweiterungen, *Open-Estelle* und auch die *Containertyp-Erweiterung*, sind konzeptionell unabhängig von einer konkreten Spezifikationstechnik und können (aufgrund der dort ebenfalls vorliegenden Problemstellungen) z. B. auch auf SDL übertragen werden.

8.2. Effiziente Implementierbarkeit

Die Ableitung *korrekter Implementierungen* aus formalen Protokollspezifikationen ist eine wichtige Voraussetzung zur Übertragung der auf formaler Ebene nachgewiesenen Eigenschaften auf die Implementierungsebene. Besondere Attraktivität besitzt dieser Implementierungsschritt, wenn er automatisiert werden kann, da dann der Nachweis der Korrektheit der Implementierung auf den Nachweis der Korrektheit des Implementierungsverfahrens und der verwendeten Werkzeuge zurückgeführt werden kann.

Als Grundlage für unsere Implementierungs- und Optimierungsexperimente haben wir den „*eXperimental Estelle Compiler*“ (XEC) entwickelt, der ein wichtiges Ergebnis dieser Arbeit darstellt (Kapitel 3). Er ermöglicht aufgrund seines Implementierungskonzeptes eine sehr flexible Modellierung des Systemmanagements und ist somit insbesondere für die Realisierung verschiedener Auswahloptimierungen geeignet. XEC ist zudem mit verschiedenen Statistik- und Monitoring-Funktionalitäten ausgestattet, durch die eine effiziente quantitative Analyse der durchgeführten Implementierungsexperimente möglich ist. Neben dem vollständigen Sprachumfang von Estelle unterstützt XEC auch die meisten der hier eingeführten Estelle-Erweiterungen.

Neben der Korrektheit ist die *Effizienz* der gewonnenen Implementierungen eine wichtige Anforderung im praktischen Einsatz. Hier zeigt sich jedoch, dass viele der in formalen Protokollspezifikationen verwendeten Konstrukte nur schwer semantikkonform und zugleich effizient implementiert werden können. Dies gilt insbesondere für die effiziente Übertragung von Nutzdaten zwischen Spezifikationskomponenten unter Einhaltung der Copy-Semantik.

In manuell erstellten Implementierungen wird die geforderte Effizienz häufig durch eine Abweichung von der spezifizierten Systemsemantik erreicht. Dies ist insbesondere dann der Fall, wenn das *ursprünglich intendierte* System nicht in der formalen Beschreibungstechnik ausgedrückt werden konnte. Die Ursachen für solche Abweichungen liegen (neben konzeptionellen Ausdrucksschwächen[1]) meist in dem spezifikationsbedingten Overhead begründet, der zu einer problemunangemessenen Komplexität führen kann.

1. D. h. ein bestimmter *implementierungsrelevanter* Aspekt kann auf Spezifikationsebene nicht modelliert werden und muss deshalb abweichend implementiert werden. Ein typisches Beispiel ist die Implementierung einer Protokollmaschinenspezifikation, die im Sinne eines offenen Systems mit einer realen Umgebung kommunizieren soll, die aber nur als Teil eines geschlossenen Systems und somit in einem anderen semantischen Kontext spezifiziert werden kann (siehe auch *Open-Estelle*).

Bei einer automatischen Generierung von Implementierungen sind solche Abweichungen meist nicht möglich und im Sinne der Korrektheit der Implementierung auch nicht wünschenswert. Entsprechend haben wir anhand des Kontrollflusses und der Handhabung von Nutzdaten untersucht, wie die spezifizierten Operationen effizient implementiert werden können, ohne das Abstraktionsniveau senken zu müssen.

Die Optimierung des Kontrollflusses (Kapitel 4) geschieht dabei ausgehend von der effizienten Realisierung der Basisoperationen der von XEC erzeugten Implementierungen primär anhand der Transitionsauswahl, da diese speziell bei komplexen Spezifikationen einen erheblichen Teil der Ausführungszeit beansprucht. Wir haben dazu verschiedene heuristische Optimierungen der globalen Auswahl und der modullokalen Auswahl entwickelt und sowohl analytisch wie auch experimentell evaluiert. Wesentliche Ansatzpunkte waren dabei verschiedene ereignisgesteuerte Auswahlverfahren auf globaler Ebene und die Reduktion der zu untersuchenden Transitionen auf lokaler Ebene. Einige dieser Verfahren konnten durch den Einsatz bestimmter Spezifikationsstilregeln oder Estelle-Erweiterungen wie die *Independent-Module-Erweiterung* in ihrer Wirksamkeit deutlich verbessert werden. Dabei konnten bzgl. der Anzahl der getesteten Module und Transitionen Leistungssteigerungen um ganze Größenordnungen erreicht werden. Die Überprüfung der Ergebnisse anhand der ausführungszeitbezogenen Leistungsbewertung hat diese Ergebnisse bestätigt.

Hinsichtlich der effizienten Handhabung von Daten (Kapitel 6) haben wir unterschiedliche Ansätze auf verschiedenen Ebenen untersucht, die jedoch in den meisten Fällen eine problemunangemessene Ausrichtung der Spezifikation auf die effiziente Datenübertragung erfordern. Eine überraschend elegante, problemorientierte und effiziente Lösung konnte schließlich auf Basis der *Containertyp-Erweiterung* entwickelt werden, die ursprünglich in Kapitel 5 zur Steigerung des Abstraktionsniveaus eingeführt wurde. Dieses Ergebnis widerlegt die Vorstellung, dass Maßnahmen zur Steigerung der effizienten Implementierbarkeit auch immer durch eine Senkung des Abstraktionsniveaus erkauft werden müssen.

8.3. Ausblick

Optimierungsmaßnahmen können immer nur bestimmte Aspekte eines Systems ansprechen und sind somit gerade im Bereich von operationalen formalen Beschreibungstechniken aufgrund der Vielfältigkeit der zu berücksichtigenden Randbedingungen und Konstellationen *prinzipiell unvollständig*. Dies gilt ganz besonders für heuristische Optimierungen, die auf das Einhalten bestimmter Bedingungen ausgerichtet sind.

Die bisherigen Evaluierungen haben die untersuchten Spezifikationen als mehr oder weniger fest vorgegeben behandelt und nur wenige minimale Änderungen zur Verbesserung der Anwendbarkeit unserer Heuristiken durchgeführt. An einigen Stellen haben wir jedoch bereits Leitlinien für den Spezifizierer entwickelt, durch die die Wirksamkeit der Optimierungsmaßnahmen verbessert werden kann.

Der nächste logische Schritt ist somit die Untersuchung, welchen Einfluss verschiedene Stile oder Lösungsansätze bei der Entwicklung einer Spezifikation tatsächlich auf die Effizienz der Implementierung haben. Auf Basis solcher Untersuchungen könnten dann nicht nur die bereits entwickelten Optimierungsansätze anhand einer jeweils optimal angepassten Spezifikation evaluiert werden, sie könnten auch Hinweise zur Entwicklung neuer Optimierungsheuristiken liefern.

Interessante Optimierungsansätze für die globale Transitionsauswahl ergeben sich auch durch die Fortsetzung der Konzepte der Ereignissteuerung auf gemeinsame Variablen. Hier könnte durch eine präzisere Analyse der lesenden und ändernden Zugriffe die Wirksamkeit der Ereignissteuerung deutlich erhöht werden. Dies gilt besonders für den XTP-Benchmark, der an vielen Stellen gemeinsame Variablen einsetzt.

Diese Analysen könnten auch auf die lokale Transitionsauswahl fortgesetzt werden, um so letztlich auch die Auswahl bzgl. der PROVIDED-Klauseln optimieren zu können. Dies gilt speziell für lokale Variablen, die den Kontrollzustand eines lokalen Teilautomaten nachbilden. Hier werden jedoch sicherlich strikte Spezifikationsstilregeln zur Vereinfachung der Aufgabe erforderlich sein.

Interessante Fragestellungen abseits der effizienten Implementierung ergeben sich bei den Encoding-Verfahren für die Containertyp-Erweiterung vor dem Hintergrund der Steigerung der Interoperabilität von (offenen) Estelle-Implementierungen mit anderen Systemen. Hier bietet speziell eine Anbindung an ASN.1 [ISO88a] einen viel versprechenden Ansatz.

Schließlich können viele der vorgestellten Implementierungs- und Optimierungskonzepte auf andere formale Beschreibungstechniken wie z. B. SDL übertragen werden. Dies gilt auch für die Konzepte hinter den Estelle-Erweiterungen Containertyp und Open-Estelle. In welchem Umfang diese Übertragungen letztlich im Detail sinnvoll und möglich sind, bleibt an dieser Stelle offen.

Anhang A: XEC

Wir ergänzen in diesem Anhang einige technische Zusatzinformationen zu den in Abschnitt 3.1 vorgestellten Komponenten des XEC-Toolkits (siehe auch Abb. 3-1 auf Seite 25).

A.1: Der Compiler-Frontend PET

Der „*Portable Estelle Translator*“ *PET* des XEC-Toolkits (siehe Abschnitt 3.1.1) ist eine erweiterte Version des gleichnamigen Tools aus dem bereits in Abschnitt 2.1.4 genannten Estelle-Implementierungswerkzeug PET/DINGO [SiSt93]. PET liest als *Compiler-Frontend* Estelle-Spezifikationen im Textform, analysiert und überprüft ihre Syntax anhand der kontextfreien und kontextsensitiven Anteile der Estelle-Grammatik und bildet intern eine zur syntaktischen Struktur der Spezifikation äquivalente baumartige Grafendarstellung, deren Knoten Instanzen der *PET-Klassenbibliothek* sind.

Ausgehend von der Urversion von PET aus dem genannten PET/DINGO-Toolkit wurde PET im Rahmen der Entwicklung von XEC in großen Teilen neu kodiert und insbesondere zur Unterstützung der o. g. Estelle-Erweiterungen deutlich erweitert.

```
>> Portable Estelle Translator (PET) 2.02

>> Usage: pet {OPTION | FILE}

>> FILE: Estelle source file ("-" means stdin)

>> OPTIONS:
>>   -o OBJ_FILE   write object format to OBJ_FILE
>>   -l            write listing of source file to FILE.lst
>>   -c[-]         do [not] keep comments in object file (default: -c)
>>   -i            list names of include files
>>   -v            verbose mode
>>   -xasync       support eXtension 'ASYNChronous process'
>>   -xindep       support eXtension 'INDEPendent modules'
>>   -xopen        support eXtension 'OPEN estelle'
>>   -xopen_ur     support eXtension 'OPEN estelle'
>>                         with Unqualified Reference
>>   -I DIR        insert DIR to interface search path
>>                 (only useful with -xopen)
>>   -xat          use eXtension 'Any Type'
>>   -xptria       use eXtension 'PoinTeRs as InterAction parameters'
>>   -x            enable all known eXtensions
>>   -relaxed      rate uncritical errors as warnings
>>                 (e.g. global variables and pure/non-pure problems)
```

A.1.1: Die PET-Klassenbibliothek

Die PET-Klassenbibliothek enthält im Wesentlichen für jedes syntaktische Element der Estelle-Grammatik die Definition einer jeweils spezialisierten Klasse[1], die neben Feldern zur Darstellung der spezifischen Parameter des syntaktischen Elements (z. B. der konkrete Name eines Identifiers oder der Verweis auf die ggf. referenzierte Definition) auch Methoden zum Zugriff auf diese Parameter bereitstellt. Die Äquivalenz dieser Grafendarstellung zur Ausgangsspezifikation lässt sich bereits daran erkennen, dass die genannten Klassen auch Methoden zur vollständigen *textuellen Rückgewinnung* eines zur Ausgangs-Spezifikation analogen[2] Spezifikationstextes beinhalten.

Eine weitere wesentliche Eigenschaft der PET-Klassenbibliothek ist die Fähigkeit, die beschriebene Objektstruktur persistent in einer Binärdatei (*PET-Objektdatei*) abzulegen und zu einem späteren Zeitpunkt wieder in eine äquivalente Objektstruktur restaurieren zu können. Dabei kapselt die PET-Klassenbibliothek die transiente und persistente Repräsentation der Spezifikations-Syntax ein und wird entsprechend von allen Werkzeugen genutzt, die PET-Objektdateien weiterverarbeiten (siehe Abb. A-1). Dies sind neben XEC (s. u.) und `petresolve` (siehe Abschnitt 7.4.4) auch das Werkzeug `petrestore`, das die beschriebene Wiedergewinnung eines Spezifikationstextes aus einer PET-Objektdatei auf Kommandozeilenebene ermöglicht.

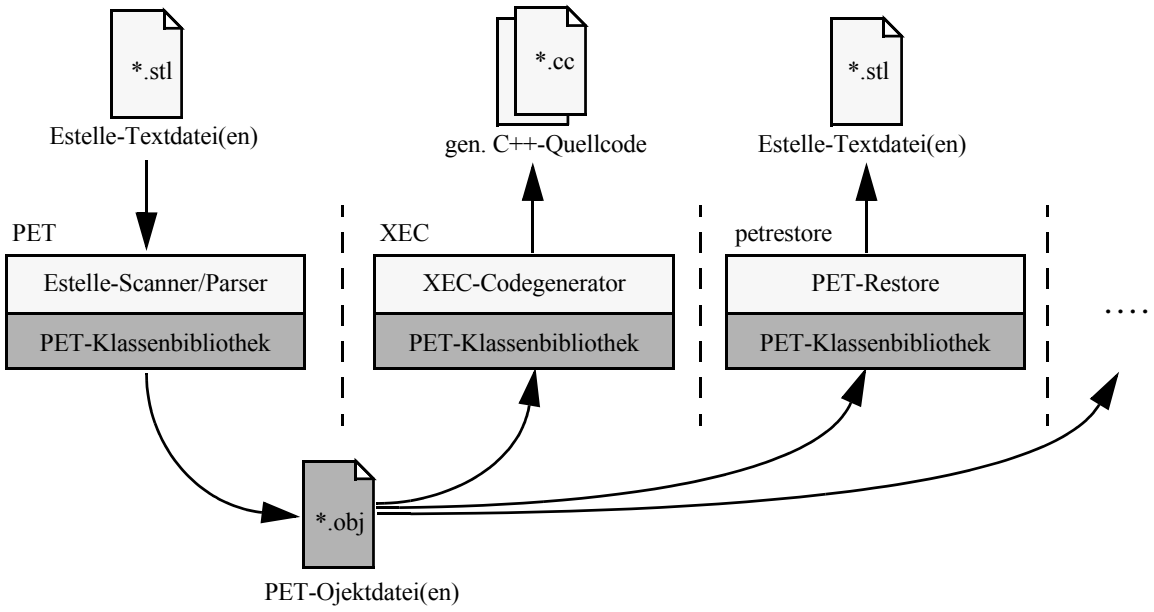


Abbildung A-1: Anwendungsstruktur der PET-Klassenbibliothek und des Objektformates

1. Die PET-Klassenbibliothek enthält allein zur syntaktischen Darstellung der Estelle-Spezifikation über 80 Klassendefinitionen.

2. Die Rückgewinnung des Spezifikationstextes beinhaltet einige syntaktische Modifikationen, die jedoch zu einer semantisch äquivalenten Spezifikation führen.

A.1.2: Textuelle Inklusion

Als Teil der Modernisierung von PET wurde auch der ursprünglich handkodierte Scanner des Compilers durch einen vollständig neu implementierten, auf *flex*[3] basierenden Scanner ersetzt.

Dabei wurden auch mehrere Erweiterungen wie z. B. die Möglichkeit einer verschachtelten textuellen Inklusion von Dateien in den Spezifikationstext durch Einsatz eines Include-Statements („`#include 'filename'`“) unter Erhaltung der präzisen Lokalisierung eines erkannten syntaktischen Fehlers hinzugefügt.[4] Diese Ergänzungen erwiesen sich als besonders nützlich bei der Handhabung von komplexen, tief verschachtelten Spezifikationstexten, wie z. B. bei XTP 4.0 (siehe Abschnitt 2.3.4), welche anhand der Modulhierarchie in insgesamt zehn ineinander verschachtelte inkludierte Fragmente aufgeteilt wurde.

Für die Strukturierung eines derart komplexen und tief verschachtelten Spezifikationstextes in Include-Dateien bietet sich dabei die folgende Struktur an, wobei als Basisname der jeweiligen Dateien der (voll qualifizierte) Modulname benutzt wird:

Beispiel A.67-a: Inhalt der Spezifikations-Datei „`spec.stl`“

```
SPECIFICATION spec
    { lokale Definitionen, Kanäle, Interfaces, ... }
    CHANNEL {....} ;

    { Kindmodule textuell inkludieren: }
    #include 'spec.kindmodul_1.inc'
    #include 'spec.kindmodul_2.inc'
    { .... }

    { Modulrumpf: Variablen, Transitionen, ... }
    TRANS {....} ;
END.
```

(Ende von Beispiel A.67-a)

Die inkludierten Kindmoduldefinitionen selbst enthalten möglicherweise selbst wiederum Kindmodule und können daher analog textuell strukturiert werden:

Beispiel A.67-b: Inhalt der Spezifikations-Datei „`spec.kindmodul_1.stl`“

```
MODULE kindmodul_1
    { lokale Definitionen, Kanäle, Interfaces, ... }
    CHANNEL {....} ;

    { eigene Kindmodule textuell inkludieren: }
    #include 'spec.kindmodul_1.kindmodul_1a.inc'
    #include 'spec.kindmodul_2.kindmodul_1b.inc'
    { .... }
```

3. „*fast lexical analyzer generator*“, eine Erweiterung des UNIX-Tools *lex*

4. Die Integration der textuellen Inklusion in den Scanner hat wesentliche Vorteile gegenüber einem Präprozessorlauf (z. B. durch `cpp`), da neben der angemessenen Handhabung von Estelle-Kommentaren („`(*...*)`“ und „`{...}`“) und Strings („´...´“ statt ´"..."´) bzgl. Include-Statements auch die Lokalisierung von Fehlern in der Quelldatei anstatt in der generierten Zwischendarstellung erfolgen kann.

```
    { Modulrumpf: Variablen, Transitionen, ... }
    TRANS {....} ;
END;
```

(Ende von Beispiel A.67-b)

Durch diese Strukturierungsmethode entsteht für jede Moduldefinition genau eine Textdatei, die gegebenenfalls Include-Anweisungen für die textuelle Einbettung von Kindmoduldefinitionen enthält.

Dies erlaubt es, eine der wesentlichen Schwächen von Estelle bei der Handhabung komplexer und tief verschachtelter Modulhierarchien abzumildern, indem die Kindmoduldefinitionen nicht mehr (rekursiv) als Teil der Definition ihres jeweiligen Vatermodules mitten in der Folge der lokalen Definitionen dieses Vatermoduls eingeschlossen werden und so auch nicht mehr durch die z.T. extreme räumliche Trennung[5] der direkt zum Vatermodul gehörigen Teile sehr unübersichtlich werden. Stattdessen ergibt sich die Modulhierarchie der Spezifikation nach obigem Dateinamensmuster für Include-Dateien bereits aus den Dateinamen, und alle Moduldefinitionen werden (lediglich unterbrochen durch die Include-Anweisungen) als textuelle Einheit angegeben.

A.1.3: Aktivierung von Estelle-Erweiterungen

Weitere Ergänzungen von PET betreffen verschiedene Estelle-Erweiterungen, die im Verlauf dieses Textes detailliert vorgestellt wurden. Zur Aktivierung[6] der Erweiterungen werden jeweils Kommandozeilenoptionen (s. o.) eingesetzt, die eine präzise Abgrenzung der Spracherweiterungen zwischeneinander und zum Estelle-Standard erlauben.

Die unterstützten Estelle-Erweiterungen und die zu ihrer Aktivierung erforderlichen PET-Kommandozeilenoptionen sind in Tabelle 3.4 auf Seite 117 dargestellt. Da alle genannten Erweiterungen voll kompatibel[7] zueinander und insbesondere zu Standard-Estelle sind, können auch alle diese Erweiterungen zugleich aktiviert werden (u. a. durch die Option „`-x`“).

---

5. Die unmittelbar lokalen Definitionen des Spezifikationsmoduls selbst befinden sich am Anfang und Ende des Spezifikationstextes und sind z. B. im Falle der Estelle-Spezifikation von XTP 4.0 durch über 7000 Zeilen tief verschachtelter Kindmoduldefinitionen voneinander getrennt.

6. Der neuimplementierte Scanner erlaubt u. a. auch die dynamische Aktivierung und Deaktivierung der durch optionale Erweiterungen eingeführten neuen Schlüsselwörter, um im Falle der Deaktivierung einer Erweiterung diese nun nicht mehr reservierten Identifier uneingeschränkt für eigene Definitionen nutzen zu können.

7. Lediglich durch die Erweiterungen eingeführte neue Schlüsselwörter (s. o.) können durch ihre Verwendung als Namen in Standard-Estelle-Spezifikationen zu Interferenzen führen. Eine explizite Angabe des benötigten Sprachumfangs durch die genannten Kommandozeilenparameter vermeidet diese Probleme, da der lexikografische Scanner von PET nur die Schlüsselwörter des gerade selektierten Sprachumfangs als solche interpretiert.

A.1.4: Datei-Namenskonventionen und praktische Anwendung

Wurde durch die genannten Optionen ein geeigneter Sprachumfang festgelegt, so ist zur Erzeugung einer PET-Objektdatei im Allgemeinen nur noch die Angabe einer Estelle-Quelldatei erforderlich. Als Namenskonvention werden von den Werkzeugen bzw. in den Beispieldateien zu XEC, wie auch in diesem Text, die in Tabelle A.32 angegebenen Dateinamenserweiterungen eingesetzt, wobei die letzten beiden Einträge nur in Zusammenhang mit der Estelle-Erweiterung „*Open-Estelle*“ (siehe Kapitel 7) von Bedeutung sind.

Dateinamenserweiterung	Dateityp
`.stl`	Estelle-Quelltext
`.inc`	Estelle-Quelltext (Include-Datei)
`.obj`	erzeugte PET-Objektdatei
`.sti`	Estelle-Quelltext (**Open-Estelle**-Interface)
`.obi`	erzeugte PET-Objektdatei (**Open-Estelle**-Interface)

Tabelle A.32: Konventionen für Dateinamenserweiterungen

Somit ergibt sich zur Übersetzung einer einfachen Standard-Estelle-Spezifikation in der Datei „`test.stl`“ im aktuellen Verzeichnis der in Abb. A-2 dargestellte Aufruf. Dabei wird in der Datei „`test.obj`“ die PET-Objektdatei zur Ausgangsdatei erzeugt.

```
# pet test.stl

>> Portable Estelle Translator (PET) 2.01
>> translating input file test.stl
>> writing object file test.obj
```

Abbildung A-2: Einfacher Übersetzungslauf von PET (V2.01)

A.2: Der Code-Generator XEC

A.2.1: Kommandozeilenoptionen

```
>> XEC (V 1.3.3)

syntax: xec infile-name [text-outfile-base] options

infile-name:         PET-obj file for input
text-outfile-base:   basename for *.cc, *.h and *.mk files
                     (default: basename and directory of infile-name)

options: -m opt      create makefile from template "xecmk_<opt>.mk"
                     (default for <opt>: "default")
         -mkembed    embed makefile templates instead of including them
         -l dir      directory of runtime-library-sources
                     (default: ${XEC} or '/local/lib/xec/1.3.3')
         -v          dump statics of compilation
         -V          dump version IDs of source modules

example:  'xec x.obj out/x'
                     will create out/x.mk, out/x.cc, and out/x.h
                     out/x.mk will be prepared to build out/x
```

Siehe auch Anhang A.6 zu den Makefile-Templates.

A.2.2: Quelldateien und Klassen

Quelldatei(en)	Aufgaben	Enthaltene Klassen (u. a.)	Aufgaben
cmain.cc/h	Steuerung der Codegenerierung, Kommandointerface	Compiler	
cmodule.cc/h	Modulstruktur	CEnvironment	
		CState	Kontrollzustände
		CChannel	Kanaldefinition
		CHeader	Modulheader
		CModuleBody	Modulrumpf
		PredefAssocNode	Rückassoziation vordefinierter Typen, Prozeduren, Funktionen
cfunc.cc/h	Prozeduren und Funktionen	CFunction	Prozeduren und Funktionen
ctrans.cc/h	Transitionen und Transitionsklauseln	CAnyClause	ANY-Klausel
		CPriorityClause	PRIORITY-Klausel
		CFromClause	FROM-Klausel
		CToClause	TO-Klausel
		CWhenClause	WHEN-Klausel
		CProvidedClause	PROVIDED-Klausel
		CDelayClause	DELAY-Klausel
		CTransition	Transition
cdecl.cc/h	Typen, Variablen, Expressions, Statements,	DefAssocNode	Rückassoziation von PET-Objekten auf XEC-Objekte
		CDeclDef	Codegenerierung für alle sonstigen syntaktischen Elemente (Basisklasse für fast alle anderen Klassen)
cmisc.cc/h	Hilfsfunktionen	TracePoint	statische Monitoring-Unterstützung

Tabelle A.33: XEC-Quelldateien und Klassen

A.3: Die Laufzeitbibliothek XECRT

Quelldatei(en)	Aufgaben	Enthaltene Klassen	Aufgaben
xecrt_modules.cc/h	Modulstruktur, Transitionen, Interaktionen, Modulvariablen	InterAction-Abstract	Basisklasse f. Interaktionen
		SimpleTrans	Transitionen ohne DELAY und WHEN
		DelayedTrans	DELAY-Transitionen
		DelayTimer	abstrakter Timer für DELAY
		InputTrans	Eingabetransitionen
		IP	Interaktionspunkte
		Module	Modul (-Header)
		Specification	Spezifikation
		StaticModule-DataBase	Modul-Metaklasse
		StaticModuleData	
		Modvar	Modulvariablen
		AllModIterator	ALL <modultyp>
		AllOrdIterator	ALL <ordinaltyp>

Tabelle A.34: XECRT-Quelldateien und Klassen

Quelldatei(en)	Aufgaben	Enthaltene Klassen	Aufgaben
xecrt_estelle.cc/h	Datentypen (Strings, Arrays, Sets, etc.)	Timescale	systemweite Zeit
		Array, ArrayS, ArrayC	Arrays (auch Strings)
		String	String-Konstanten
		AbstractSet, TypedSet, DynamicSet, EmptySet	Sets (getypt, dynamisch, leer)
		UNSPECIFIED_TYPE	„...“-Typ
		Container	Containertyp-Erweiterung (siehe Abschnitt 5.2)
		Abstract_Container	
xecrt_context.cc/h	Tracing, Monitoring, Timer	ReportItem	automatische Ausgabe von Event-Zählern etc.
		Counter	Event-Zähler
		Timer	Echtzeit- / Simulationszeit-Uhr
		MonitorPacketBase	Monitoring von Paketen (PATO)
		Tracing	Event-Trace-Puffer
xecrt_debugger.cc/h	Debugger-Schnittstelle	Debugger	polymorphe Basisklasse für alle (externen) Debugger

Tabelle A.34: XECRT-Quelldateien und Klassen

A.4: Kommandozeilenoptionen der generierten Implementierungen

Die Kommandozeilen-Hilfe der generierten Implementierungen kann mit der Optione „-h“ bzw. „-hh“ aufgerufen werden (hier übersetzt mit mit Monitoring-Support):

```
XEC (V 1.3.3) runtime environment

 -treal [maxsleep_usec]          real time (default)
                                 maxsleep_usec: maximum sleep time:
                                   >= 0:       maximum sleep time in microsec.
                                               (default: no limit)
                                   'SIMSLEEP': clock jumps instead of sleeps
 -tsim [fire_usec [test_usec]]   simulated time
                                 (default: 10.0, 0.5 usec)
 -monitor [level [name [size]]] activate and control monitoring
                                 level of monitoring (default: smtp)
                                   s: specification cycles
                                   m: module cycles
                                   t: transition cycles
                                   p: packet transmission
                                 name of log-file (default: monitor.log)
                                 size of ring-buffer (default: 100'000)
 -o file                         write log-entries to given file
 -v [n]                          verbose level (def.: '-v 2'; '-v' = '-v 10')
                                 each level includes the previous levels
                                   0: -
                                   1: counter statistics at termination
                                   2: timing  statistics at termination
                                   3: module  structure  at termination
                                   4: system sleeps
                                   5: system cycles
                                   6: firing of transitions
                                   7: module instances
                                   8: interactions
                                   9: connections and attaches
                                  10: testing of modules
                                  11: testing of transitions
 -
[no]deadlock                    control deadlock detection (default: deadlock)
 -atbr                            copy ANY-TYPE by reference (default)
 -atbc                            copy ANY-TYPE by copying
 -modpool n                       limit instance pool (per type) to n
                                    (n >= 0, default: 10)
 -localselect t                   set default local selection method to t
                                    (straight, prio, CST, QST)
 -[no]treesleep                   allow/deny tree sleep optimization
                                    (default: -treesleep)
 -[no]imodopt                     allow/deny independent root module opt.
                                    (default: -imodopt)
 -eventdrive n                    set event driven module selection
                                    (n=0:off, n=1:simple, n=2:mult., n=3:rep.)
                                    (default: -eventdrive 3)
 -h | -hh                         print command line help (short or long)
```

A.5: Die XList-Listenverwaltungen der Baselib-Klassenbibliothek

Neben diversen Definitionen[8], Hilfsklassen und -Funktionen stellt die Klassenbibliothek Baselib (baslib.h und baslib.cc) als wichtigste Komponente vier Klassen-Templates[9] zur besonders effizienten und abstrakten Handhabung von (doppelt verketteten) Listen bereit:

- XList <class T>
- XListNode <class T>
- XPtrList <class T>
- XPtrListNode <class T>

Die ersten beiden Klassen-Templates XList und XListNode dienen dabei zur Realisierung einer *exklusiven* Listenzugehörigkeit einer Menge von Objekten (abgeleitet von XListNode) zu einer Liste (Instanz von XList), wohingegen Listen des Typs XPtrList Objekte *nicht-exklusiv* enthalten. Insofern stellen Listen des Typs XPtrList „klassische“ Listen dar, die lediglich Referenzen auf eine Menge von Datenobjekten bereitstellen. Wir werden jedoch sehen, dass solche Referenz-Listen lediglich einen Spezialfall einer allgemeineren Klasse von Listen bilden und entsprechend auch in unserer Implementierung durch Spezialisierung dieser allgemeineren Klassen-Templates XList und XListNode definiert sind. Die besonderen Eigenschaften dieser allgemeineren Klassen-Templates bieten dabei eine besonders günstige Möglichkeit zur Modellierung von weitgehend Typ-homogenen und streng hierarchischen Objektstrukturen, bei denen z. B. die Löschung eines Knotens Automatismen zur Pflege der Objektstruktur nach bestimmten Regeln auslösen soll.

Betrachten wir dazu die klassische Implementierung einer doppelt verketteten Liste zur Aufnahme einer Folge von Nutzlast-Objekten eines gegebenen Typs (siehe Abb. A-3). Die wesentliche Datenstruktur der Liste besteht aus einer Menge von Listen-Knoten, die jeweils eine Referenz (in C++ typischerweise ein Pointer) auf ihre beiden Nachbarn (Vorgänger- und Nachfolger-Knoten bezüglich der Knotenreihenfolge in der Liste) enthält. Gibt es keinen solchen Nachbarn, so ist hier eine spezielle leere Referenz (in C++ typischerweise der NULL-Pointer) hinterlegt. Weiterhin enthält jeder Listenknoten einen Verweis auf die eigentliche Listen-Nutzlast, also das Objekt, das in der Liste verwaltet werden soll. Die Liste wird verankert durch ein Listen-Wurzel-Objekt, das Verweise auf den jeweils ersten und letzten Listen-Knoten enthält.

Durch eine Implementierung mittels zweier Klassen-Templates, die den Nutzlast-Typen als Typ-Parameter[10] erhalten, lässt sich eine solche doppelt verkettete Liste vollständig typsicher implementieren, d. h. es sind zur Kodierung der Listen-Klassen selbst und bei ihrer Anwendung keinerlei explizite oder implizite Typkonvertierungen erforderlich. Wir werden später mit der XPtrList eine solche Implementierung kennen lernen.

---

8. U. a. werden in der Datei baselib.h diverse compilerabhängige Makros definiert, die zur Umgehung von diversen Fehlern verschiedener C++-Compiler (-Versionen) eingesetzt werden.

9. Wir unterscheiden hier begrifflich zwischen *Klassen-Templates* (also der Definition der Templates) und (*Template-*) *Klassen* (also die durch die vollständige Parametrierung der Templates gewonnenen eigentlichen Klassen). Analog werden auch die Begriffe *Funktions-Templates* und *Template-Funktionen* eingesetzt.

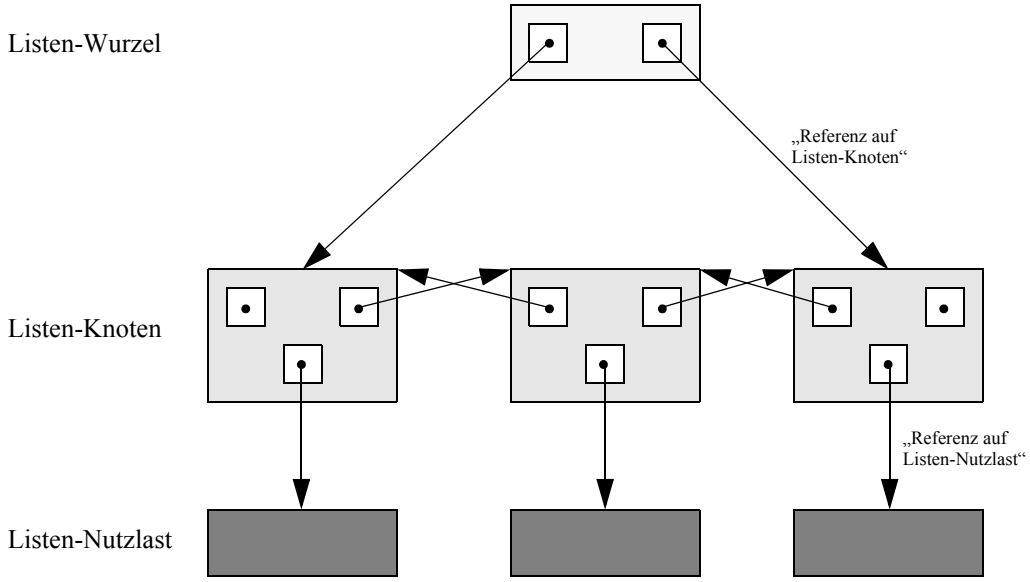


Abbildung A-3: Konventionelle DV-Liste mit separaten Verwaltungsknoten

Der Vorteil einer solchen Listenkonstruktion besteht insbesondere darin, dass lediglich Referenzen ausgehend von der Liste (genauer von den Listen-Knoten) auf die Nutzlast-Objekte existieren und damit keine besonderen Vorkehrungen bei der Klassen-Definition der Nutzlast-Objekte für eine mögliche Aufnahme in eine solche Liste erforderlich sind. Zudem kann ein Nutzlast-Objekt in beliebig vielen Listen zugleich enthalten sein.

Ihr Nachteil besteht jedoch umgekehrt darin, dass es von Seiten der Listen-Nutzlast zunächst keinen expliziten Bezug auf den sie referenzierenden Listen-Knoten gibt. Dies macht es z. B. erforderlich, dass zur Entfernung eines gegebenen Listen-Nutzlast-Objekts aus einer solchen Liste typischerweise die gesamte Liste nach dem referenzierenden Listen-Knoten durchsucht werden muss.

Besonders deutlich wird dieses Problem, wenn ein Nutzlast-Objekt gelöscht (und damit destruiert) wird, da dann alle Referenzen auf dieses Objekt ungültig werden.[11] Dies schließt natürlich auch alle Listen ein, die möglicherweise Referenzen auf das Nutzlast-Objekt beinhalten.

---

10. Die Nutzlast-Objekte müssen zur Wahrung der Typsicherheit in der Listenverwaltung einem gemeinsamen Typ angehören, der dann als Typ-Parameter der Listen-Klassen-Templates eingesetzt wird. Dieser gemeinsame Typ kann natürlich auch im Sinne einer Klassenhierarchie ein gemeinsamer Basistyp sein. Vollständig heterogene Listen dieser Art lassen sich in C++ im Extremfall aber auch durch die Parametrierung mit dem Nutzlast-Typen `void` erreichen, wodurch jedoch Typkonvertierungen beim Zugriff auf die in einer Liste enthaltenen Nutzlast-Objekte auf Anwendungsebene unumgänglich ist, da die Referenzen auf die Nutzlast-Objekte als untypisierte Zeiger (d. h. `void*`) gehandhabt werden.

11. Das sog. „*dangling pointer*“-Problem bei expliziter Speicherfreigabe. Die Problematik der unerwünschten Listenzugehörigkeit auf *semantischer* Ebene besteht jedoch unabhängig vom Speicherverwaltungsmodell.

Allgemein ist das Entfernen eines Nutzlast-Objekts aus einer (bzw. allen) referenzierenden Listen überhaupt nur dann möglich, wenn diese Listen (d. h. die Listen-Wurzel-Objekte) im Kontext dieses Entfernungs-Vorgangs bekannt sind. Das Entfernen eines Listen-Nutzlast-Objektes aus einer (möglicherweise) enthaltenden Liste erfordert also in jedem Fall zusätzliche Maßnahmen bei der Anwendung der Liste.

Dieser zusätzliche Management-Aufwand ist der Preis für die Flexibilität dieser Art von Listen, da die Aufnahme eines Nutzlast-Objekts in die Liste weder spezielle Anforderungen an den Typ der Nutzlast stellt, noch eine exklusive Listenzugehörigkeit erzwungen wird.

In vielen Fällen wird durch die Listenzugehörigkeit jedoch eine Form von starker Aggregation ausgedrückt, indem ein Nutzlast-Objekt immer genau einer (bzw. höchstens einer) Liste angehört. Dies gilt u. a. bei der Nachbildung von Baumstrukturen, wie man sie z. B. in der streng hierarchisch strukturierten Modulstruktur von Estelle findet. So kann man in Kapitel 3 sehen, dass bis auf die Spezifikations-Modulinstanz (als Wurzel des Aggregations-Baumes) tatsächlich alle vom XEC-Laufzeitsystem angelegten Datenobjekte in jeweils genau einem anderen Objekt (stark) aggregiert sind.

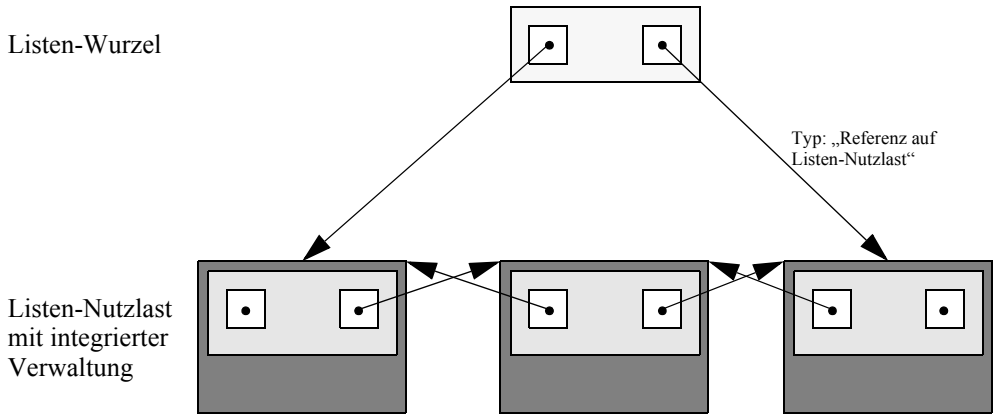


Abbildung A-4: Verwaltungsknoten in Nutzlast integriert

In solchen Fällen kann durch eine Verschmelzung von Listen-Knoten und Listen-Nutzlast (siehe Abb. A-4) ein wesentlich höherer Abstraktionsgrad erreicht werden, indem die Listenzugehörigkeit des Nutzlast-Objekts zu einem *integrierten Objektattribut* wird. Bezüglich der Verwaltung der Listenzugehörigkeit kann ein solches integriertes Nutzlast-Objekt viele Automatismen ohne die Notwendigkeit expliziter Management-Maßnahmen auf Anwendungsebene realisieren. So wird z. B. bei der Destruktionen eines integrierten Nutzlast-Objekts auch automatisch der darin enthaltene Listen-Knoten destruiert, wodurch eine automatische Auskettung aus der Listenstruktur ermöglicht wird. Ebenso kann die exklusive Zugehörigkeit zu (höchstens) einer Liste beim Einfügen des integrierten Nutzlast-Objekts in eine andere Liste ebenfalls durch eine automatische Auskettung aus der bisherigen Liste gesichert werden.

Zu diesem Zweck enthält die Listenstruktur zusätzliche Feldelemente, wie z. B. eine Referenz jedes integrierten Listen-Knotens auf die Listen-Wurzel. Damit ergibt sich in die in Abb. A-5 dargestellte Datenstruktur, die die interne Struktur einer `XList` darstellt.

Die herausragende Eigenschaft der Listen-Implementierung durch die Klassen-Templates `XList` und `XListNode` ist jedoch die Art der typsicheren Integration des Listen-Knotens (`XListNode`) in das Nutzlast-Objekt. Dazu muss jede Nutzlast-Klasse die Klasse `XListNode` als Basisklasse beinhalten. Die Referenzen innerhalb von `XListNode` auf die

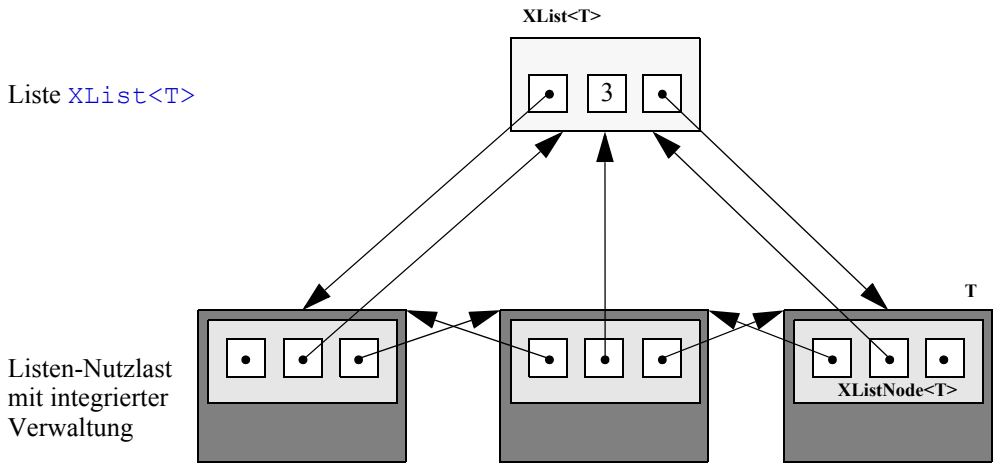


Abbildung A-5: Listenstruktur mit Nutzlast `T`, `XListNode<T>` und `XList<T>`

benachbarten Listenelemente verweisen jedoch nicht direkt auf den Typ `XListNode`, sondern auf die davon abgeleitete Nutzlast-Klasse (siehe Ziel der Pfeile in Abb. A-5). Dies wird es später beim Durchlaufen solcher Listen ermöglichen, ohne Typkonvertierungen und ohne explizite Trennung der Eigenschaften der Nutzlast und des enthaltenen Listen-Knotens durch die Liste zu navigieren.

Damit die Klasse `XListNode` Referenzen auf Instanzen einer von sich selbst abgeleiteten Klasse enthalten kann, wird dieser abgeleitete Typ als Typ-Parameter an die Template-Klasse `XListNode` übergebenen (siehe auch UML-Diagramm in Abb. A-6).

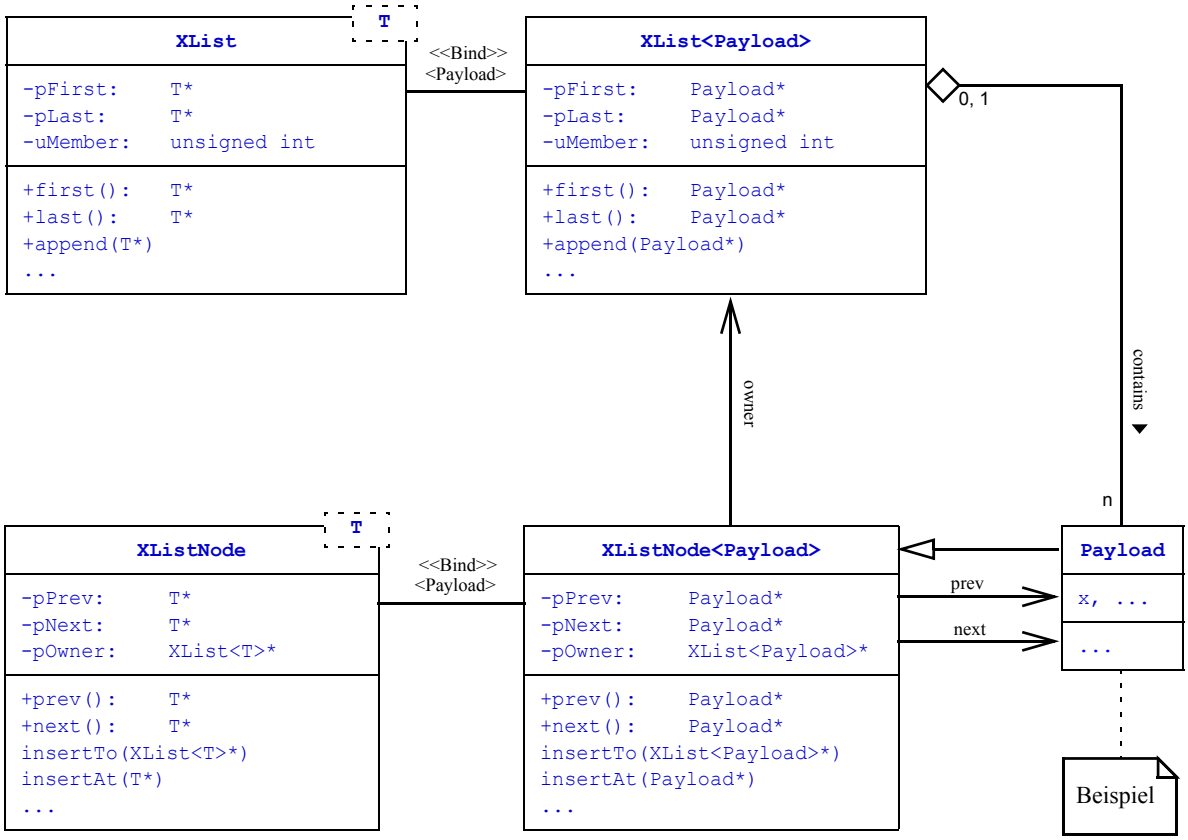


Abbildung A-6: UML-Diagramm der Anwendungs-Struktur von „`XList`“ und „`XListNode`“

Es entsteht damit eine Klassendefinition wie im folgenden Beispiel einer Nutzlast-Klasse namens „Payload“:

Beispiel A.68-a: Anwendung von XList: Nutzlast-Klassendefinition

```
class Payload : public XListNode<Payload> {
    public:
        int x, y, z;                    // example components
        Payload(int x_) : x(x_) {}      // example constructor
};
```

(Ende von Beispiel A.68-a)

Die Nutzlast-Klasse Payload hat als Basisklasse die Klasse XListNode, die umgekehrt wiederum als Typ-Parameter die (gerade in Definition befindliche) Klasse Payload erhält. Offensichtlich handelt es sich um eine rekursive Klassendefinition, die jedoch gemäß des C++-Standards [ISO98] auf Grund der Art der Verwendung des Parameter-Typs in XListNode[12] wohlfundiert ist.

Um nun ein Liste zur Aufnahme solcher Payload-Objekte anzulegen, genügt es, eine Instanz der Klasse XList<Payload> anzulegen:

Beispiel A.68-b: Anwendung von XList: Instanziierung einer Liste

```
XList<Payload> myList;
```

(Ende von Beispiel A.68-b)

Das Hinzufügen und Entfernen von Listenelementen erfolgt durch entsprechende Methoden von XList bzw. XListNode. So werden im folgenden Beispiel drei Payload-Objekte an die Liste angehängt.

Beispiel A.68-c: Anwendung von XList: Anfügen von Nutzlast-Objekten

```
myList.append(new Payload( 5));          // --> (5)
myList.append(new Payload( 7));          // --> (5, 7)
myList.append(new Payload(11));          // --> (5, 7, 11)
```

(Ende von Beispiel A.68-c)

Das Durchlaufen einer Liste geschieht typsicher, ausgehend vom ersten bzw. letzten Listenelement (Methode first() bzw. last() von XList) durch sukzessiven Aufruf der Methode next() (bzw. prev()) von XListNode. Dabei ist zu beachten, dass als Index-Referenz ein Zeiger auf den Nutzlast-Typen Payload eingesetzt wird, der sowohl Zugriffe auf direkte Attribute und Methoden der Nutzlast-Klasse (z. B. x), als auch auf Listenoperationen aus der Klasse XListNode (z. B. next()) erlaubt:

12. Insbesondere aggregiert XListNode den übergebenen Typ nicht als Strukturkomponente, da sonst durch die gegenseitige Aggregation eine nicht fundierte rekursive Datenstruktur erzeugt werden würde.

Beispiel A.68-d: Anwendung von XList: Sukzessives Durchlaufen einer Liste

```
for (   Payload* p = myList.first() ;
        p != NULL ;
        p = p->next()
) {
        cout << p->x << "\n";
}
```

(Ende von Beispiel A.68-d)

Die mit `XList` und `XListNode` aufgebauten Listenstrukturen implementieren auch die oben genannten Automatismen beim Listen-Management. So wird, wie wir bereits gesehen haben, beim Löschen eines Listen-Knotens dieser Knoten automatisch aus einer ihn möglicherweise enthaltenden Liste ausgekettet. Umgekehrt werden bei der Destruktion des Listenobjekts standardmäßig auch automatisch alle in der Liste enthaltenen Objekte im Sinne einer starken Aggregation destruiert.[13] Diese Automatismen bewirken, dass die Listenverwaltung viele Aspekte der Datenhaltung des Codegenerators XEC und des Laufzeitsystems XECRT automatisiert und auf hohem Niveau abstrahiert.

Besonderes Augenmerk verdient auch die konsequente Unterstützung von `const`-Attributen bei diesen Listen. Eine `const`-attributiertes Listenobjekt (z. B. `myConstList` in Beispiel A.68-e) liefert beim Zugriff auf die einzelnen Listen-Elemente diese ebenfalls `const`-attributiert zurück. Dies wird in C++ dadurch realisiert, dass die entsprechenden Zugriffs-Methoden jeweils doppelt ausgeführt sind, wobei die `const`-attributierten Methoden (welche beim Zugriff auf ein `const`-attributiertes Listenobjekt ausschließlich zur Verfügung stehen) entsprechend attributierte Rückgabetypen liefern. Dadurch wird vollautomatisch das durch das `const`-Attribut ausgedrückte Verbot einer Modifikation der Liste auf die in der Liste enthaltenen Elemente fortgesetzt und kann vollständig vom C++-Compiler geprüft werden. Diese Technik unterstützt in besonderem Maße die konsequente Umsetzung der Trennung zwischen modifizierenden und nicht modifizierenden Zugriffen auf Komponenten der Anwendung.

Beispiel A.68-e: Anwendung von XList: `const`-Attribute auf Listen und Listenelementen

```
const XList<Payload>& myConstList = myList;
for (   const Payload* p = myConstList.first() ;
        p != NULL ;
        p = p->next()
) {
        cout << p->x << "\n";
}
```

(Ende von Beispiel A.68-e)

Ein Beispiel für die vorgestellten Automatismen ist in Kapitel 3 die Modul-Instanz-Hierarchie der von XEC generierten Implementierungen. Man kann dort sehen, dass der Einsatz der vorgestellten Listen-Implementierung es zum Beispiel erlaubt, das Löschen einer beliebigen Modulinstanz durch Anwendung einer `delete`-Operation auf das sie implementierende C++-Objekt zu realisieren, wobei ohne explizite Maßnahmen auf Anwendungsebene automatisch so-

13. Dieses Default-Verhalten kann durch Aufruf der `XList`-Methode „`setNodeAutoDispose(false)`“ auf ein reines Ausketten der Knoten bei der Destruktion der Liste abgeändert werden.

wohl die Aktualisierung der Listenstrukturen der übergeordneten Modulinstanzen erfolgt, als auch alle aggregierten Komponenten (Kindmodulinstanzen, Transitions-Objekte, Timer-Objekte, etc.) behandelt werden.

So wird im folgenden Beispiel beim zweiten Einfügen von *p7 zunächst das Element automatisch aus seiner bisherigen Liste (bzw. Listenposition) entfernt, bevor es an die angegebene Liste angefügt wird. Dieser Mechanismus arbeitet ohne Suchoperationen auf Listen und funktioniert auch beim Wechsel der Listenzugehörigkeit völlig transparent. Analog wird beim Löschen eines Nutzlastobjekts dies ggf. aus seiner Liste entfernt (siehe „delete p5“).

Beispiel A.68-f: Anwendung von XList: Löschen und Verschieben von Nutzlast-Objekten

```
Payload* p5  = new Payload( 5);
Payload* p7  = new Payload( 7);
Payload* p11 = new Payload(11);
myList.append(p5 );                // --> (5)
myList.append(p7 );                // --> (5, 7)
myList.append(p11);                // --> (5, 7, 11)
myList.append(p7 );                // --> (5, 11, 7)
delete p5;                            // --> (11, 7)
```

Zuletzt betrachten wir noch die zu Beginn dieses Abschnittes diskutierten Listen von Referenzen auf Nutzlast-Objekte, die insbesondere zur Modellierung der Zugehörigkeit eines Objektes zu mehreren Listen eingesetzt werden kann. Bei genauerer Betrachtung wird deutlich, dass es sich dabei nur um einen Spezialfall der hier vorgestellten XList handelt, bei der der XList-Nutzlast-Typ lediglich eine Hilfsklasse ist, die als Attribut eine solche Referenz enthält (siehe Abb. A-7).

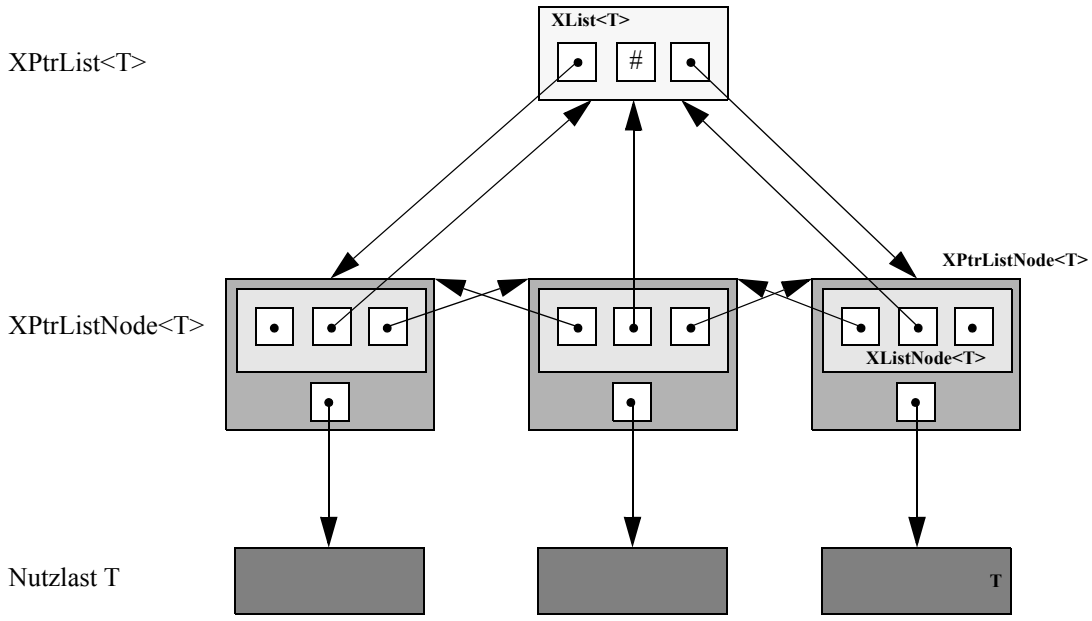


Abbildung A-7: XPtrList als Spezialfall der XList

Eine solche Referenz-Listen-Erweiterung wird aufbauend auf den Klassen-Templates XList und XListNode durch die beiden Klassen-Templates XPtrList und XPtrListNode implementiert. Beim sukzessiven Durchlaufen einer solchen Liste kann dann von jedem Listen-Knoten (XPtrListNode) aus über die Methoden data() auf das referenzierte Objekt zuge-

griffen werden. Wir beschließen diesen Abschnitt mit einem kurzen (zum obigen Beispiel analogen) Anwendungsbeispiel dieser Klassen. Die vollständige Definition der Listenklassen ist in Anhang A.6 zu finden.

Beispiel A.69: Anwendung von „XPtrList“

```
#include <iostream.h>
#include "baselib.h"

class Payload {                          // no baseclass required
    public:
        int x, y, z;                     // example component
        Payload(int x_) : x(x_) {}       // example constructor
};

class XPtrList<Payload> myPtrList;

int main() {
        myPtrList.appendPtr(new Payload(5));
        myPtrList.appendPtr(new Payload(7));
        myPtrList.appendPtr(new Payload(11));

        for (   XPtrListNode<Payload>* pn = myPtrList.first() ;
                pn != NULL ;
                pn = pn->next()
        ) {
                Payload* p = pn->data();
                cout << p->x << "\n";
        }
        return 0;
}
```

(Ende von Beispiel A.69)

A.6: Template-Klassendefinitionen für XList und XPtrList aus „baselib.h“

Der folgende Auszug aus der Bibliothek baselib.h enthält die Definition der Klassen-Templates XList, XListNode, XPtrList und XPtrListNode (siehe auch Abschnitt 3.3.1).

```
#ifdef TEST_XLIST
  #undef  TEST_XLIST
  #define TEST_XLIST(P) assert((P)->checkIntegrity())
#else
  #define TEST_XLIST(P)
#endif

template <class T> class XListNode;
        // forward


/*************************************************************************
****                                                                  ****
****                       template class XList                       ****
****                                                                  ****
*************************************************************************/

template <class T>
class XList {
        friend XListNode<T>;
    private:
        T* pFirst;
        T* pLast;
        unsigned uMember;
        bool bNodeAutoDispose;
        XList(const XList&)              {assert(0);}
    public:
        T* first()                       {return pFirst;}
        T* last()                        {return pLast;}
        const T* first() const           {return pFirst;}
        const T* last() const            {return pLast;}
        unsigned count() const           {return uMember;}
        bool isEmpty() const             {return !uMember;}

        unsigned forall(unsigned (*f)(T* pNode),
                                          bool bForward = true);
        unsigned forall(unsigned (*f)(const T* pNode),
                                          bool bForward = true) const;
        bool checkIntegrity() const;
        inline void insert(T* pNode);
        inline void append(T* pNode);
        inline void clear(bool bDisposeNodes);
        void clear()                     {clear(bNodeAutoDispose);}
        void setNodeAutoDispose(bool b) {bNodeAutoDispose = b;}
```

```
        XList()
                {pFirst=pLast=NULL; uMember=0; bNodeAutoDispose=true;}
        XList(bool b)
                {pFirst=pLast=NULL; uMember=0; bNodeAutoDispose=b;}
        ~XList()
                {clear(bNodeAutoDispose);}
};


template <class T>
inline void XList<T>::insert(T* pNode) {
        TEST_XLIST(this);
        pNode->unlink();
        if (pFirst)
                pNode->insertAt(pFirst);
        else {
                pFirst = pLast = pNode;
                pNode->pOwner = this;
                uMember = 1;
        }
        TEST_XLIST(this);
}

template <class T>
inline void XList<T>::append(T* pNode) {
        TEST_XLIST(this);
        pNode->unlink();
        if (pLast)
                pNode->appendAt(pLast);
        else {
                TEST_XLIST(this);
                pFirst = pLast = pNode;
                pNode->pOwner = this;
                uMember = 1;
        }
        TEST_XLIST(this);
}

template <class T>
inline void XList<T>::clear(bool bDisposeNodes) {
        TEST_XLIST(this);
        if (bDisposeNodes)
                while (pFirst)
                        delete pFirst;
        else
                while (pFirst)
                        pFirst->unlink();
        TEST_XLIST(this);
}


template <class T>
inline unsigned XList<T>::forall(
```

```
        unsigned (*f)(T* pNode),
        bool bForward
) {
        if (bForward) {
                for (T* pAkt = first() ; pAkt ; pAkt = pAkt->next()) {
                        unsigned uRet = f(pAkt);
                        if (uRet)
                                return uRet;
                }
        }
        else {
                for (T* pAkt = last() ; pAkt ; pAkt = pAkt->prev()) {
                        unsigned uRet = f(pAkt);
                        if (uRet)
                                return uRet;
                }
        }
        return 0;
}

template <class T>
inline unsigned XList<T>::forall(
        unsigned (*f)(const T* pNode),
        bool bForward
) const {
        if (bForward) {
                for (   const T* pAkt = first() ;
                        pAkt ;
                        pAkt = pAkt->next()
                ) {
                        unsigned uRet = f(pAkt);
                        if (uRet)
                                return uRet;
                }
        }
        else {
                for (   const T* pAkt = last() ;
                        pAkt ;
                        pAkt = pAkt->prev()
                ) {
                        unsigned uRet = f(pAkt);
                        if (uRet)
                                return uRet;
                }
        }
        return 0;
}

template <class T>
bool XList<T>::checkIntegrity() const {
        /* vorwaerts */
        unsigned uLimit = count();
        const T* pNode = first();
        while (uLimit--) {
```

```
                if (!pNode || (pNode->owner() != this))
                        return false;
                pNode = pNode->next();
        }
        if (pNode)
                return false;

        /* rueckwaerts */
        uLimit = count();
        pNode = last();
        while (uLimit--) {
                if (!pNode || (pNode->owner() != this))
                        return false;
                pNode = pNode->prev();
        }
        if (pNode)
                return false;
        return true;
}
```

```
/*****************************************************************
****                                                         ****
****                    template class XListNode             ****
****                                                         ****
*****************************************************************/

template <class T>
class XListNode {
        friend XList<T>;
    private:
        T* pPrev;
        T* pNext;
        XList<T>* pOwner;
        XListNode(const XListNode<T>&)  {assert(0);}

    public:
        T* next()                       {return pNext;}
        T* prev()                       {return pPrev;}
        XList<T>* owner()               {return pOwner;}

        const T* next() const           {return pNext;}
        const T* prev() const           {return pPrev;}
        const XList<T>* owner() const   {return pOwner;}

        void insertTo(XList<T>* pList)  {pList->insert((T*)this);}
        void appendTo(XList<T>* pList)  {pList->append((T*)this);}
        void makeFirst()                {if (pOwner && pPrev)
                                             pOwner->insert((T*)this);}
        void makeLast()                 {if (pOwner && pNext)
                                             pOwner->append((T*)this);}
        inline void insertAt(T* pNode);
```

```
        inline void appendAt(T* pNode);
        inline void unlink();

        XListNode()                         {pPrev=pNext=NULL;
                                             pOwner=NULL;}
        XListNode(XList<T>* pList)          {pPrev=pNext=NULL;
                                             pOwner=NULL;
                                             pList->append((T*)this);}
        virtual ~XListNode()                {unlink();}
};


template <class T>
inline void XListNode<T>::unlink() {
        if (pOwner) {
                TEST_XLIST(pOwner);
                pOwner->uMember--;
                if (pPrev)
                        pPrev->pNext = pNext;
                else
                        pOwner->pFirst = pNext;
                if (pNext)
                        pNext->pPrev = pPrev;
                else
                        pOwner->pLast = pPrev;
                pPrev = pNext = NULL;
                TEST_XLIST(pOwner);
                pOwner = NULL;
        }
}


template <class T>
inline void XListNode<T>::insertAt(T* pNode) {
        // *this VOR node einfuegen
        unlink();
        if (pNode->pOwner) {
                pOwner = pNode->pOwner;
                TEST_XLIST(pOwner);
                pOwner->uMember++;
                if (pNode->pPrev) {
                        pPrev = pNode->pPrev;
                        pNode->pPrev->pNext = (T*) this;
                }
                else
                        pOwner->pFirst = (T*) this;
                pNext = pNode;
                pNode->pPrev = (T*) this;
                TEST_XLIST(pOwner);
        }
}


template <class T>
```

```
inline void XListNode<T>::appendAt(T* pNode) {
        // *this NACH node einfuegen
        unlink();
        if (pNode->pOwner) {
                pOwner = pNode->pOwner;
                TEST_XLIST(pOwner);
                pOwner->uMember++;
                if (pNode->pNext) {
                        pNext = pNode->pNext;
                        pNode->pNext->pPrev = (T*) this;
                }
                else
                        pOwner->pLast = (T*) this;
                pPrev = pNode;
                pNode->pNext = (T*) this;
                TEST_XLIST(pOwner);
        }
}
```

```
/*************************************************************************
****                                                                 ****
****                    template class XPtrListNode                  ****
****                                                                 ****
*************************************************************************/

template <class T>
class XPtrListNode : public XListNode<XPtrListNode<T> > {
    private:
        T* pData;
    public:
        T* data()                       {return pData;}
        void data(T* pData_)            {pData = pData_;}
        const T* data() const           {return pData;}
        XPtrListNode(T* pData_ = NULL)  {pData = pData_;}
        ~XPtrListNode()                 {}
};
```

```
/*************************************************************************
****                                                                 ****
****                     template class XPtrList                     ****
****                                                                 ****
*************************************************************************/

template <class T>
class XPtrList : public XList<XPtrListNode<T> > {
    public:
        XPtrList()
                : XList<XPtrListNode<T> >()     {}
        XPtrList(const XPtrList<T>& src)
```

```
                : XList<XPtrListNode<T> >()     {copy(src);}
        XPtrList(bool b)
                : XList<XPtrListNode<T> >(b)    {}
        XPtrList(const XPtrList<T>& src, bool b)
                : XList<XPtrListNode<T> >(b)    {copy(src);}

        void appendPtr(T* p)          {append(new XPtrListNode<T>(p));}
        void insertPtr(T* p)          {insert(new XPtrListNode<T>(p));}
        bool addPtr(T* p)             {bool b=!isMember(p);
                                       if(b) appendPtr(p);
                                       return b;}
        inline bool subPtr(T* pData);
        inline bool isMember(const T* pData) const;
        inline void copy(const XPtrList<T>& src);
};


template <class T>
inline bool XPtrList<T>::subPtr(T* pData) {
        XPtrListNode<T>* pNode = first();

        while (pNode) {
                if (pNode->data() == pData) {
                        delete pNode;
                        return true;
                }
                pNode = pNode->next();
        }
        return false;
}

template <class T>
inline bool XPtrList<T>::isMember(const T* pData) const {
        const XPtrListNode<T>* pNode = first();

        while (pNode) {
                if (pNode->data() == pData)
                        return true;
                pNode = pNode->next();
        }
        return false;
}

template <class T>
inline void XPtrList<T>::copy(const XPtrList<T>& src) {
        for (    const XPtrListNode<T>* pNode = src.first() ;
                 pNode ;
                 pNode = pNode->next()
        )
                appendPtr((T*)pNode->data());
}
```

A.7: Namens-Präfixe von XEC

In Tabelle A.35 sind die von XEC generierten Namenspräfixe für aus der Estelle-Spezifikation übernommene C++-Identifier angegeben (siehe Kapitel 3).

Anwendung	Namens-Präfix	Bedeutung
Modulstruktur	`SPEC_`	Spezifikationsmodul
	`HEADER_`	Modulheader
	`BODY_`	Module
Kommunikation und externe Modulschnittstelle	`CHANNEL_`	Kanäle
	`IP_`	Interaktionspunkte
	`IA_`	Interaktionen (Nachrichtentypen)
	`EXPVAR_`	exportierte Variablen (für Modulheader)
Transitionen	`TRANS_`	Transitionen
Konstanten, Typen, Variablen	`CONST_`	Konstanten, Aufzählungswerte
	`TYPE_`	Typen
	`VAR_`	Variablen
	`MODVAR_`	Modulvariablen
Funktionen und Prozeduren	`PAR_`	Parameter (für Funktionen, Prozeduren, Module, Interaktionen)
	`FUNCPAR_`	funktionale Parameter
	`FUNC_`	Funktionen
	`PROC_`	Prozeduren

Tabelle A.35: Namens-Präfixe von XEC

A.8: Generierte Makefiles und Makefile-Templates

<opt>	gcc[a]	NDEBUG[b]	Log-ging[c]	Proto-typen[d]	Inline C++[e]	Pretty-Printer[f]	Debug-ger[g]	Moni-toring[h]
all	-g		+	+	+	+	+	+
debugger	-g		+	+	+	+	+	
default	-g		+	+	+	+		
monitor	-O2	+	+	+	+	+		+
optimize	-O2	+			+			
optstat	-O2	+	+	+	+			
quick				+				

Tabelle A.36: Übersetzungs-Optionen zu XEC-Makefiles

a. Kommandozeilenoptionen für den C++-Compiler (hier: gcc); „`-g`“ erzeugt Debuginformationen für einen Binär-Debugger, „`-O2`“ aktiviert die Code-Optimierung des C++-Compilers (s. u.)
b. Durch setzen der Variablen „`NDEBUG`“ werden einige Fehlerüberwachungen mit „`assert`“-Statements im generierten Code und den Laufzeitbibliotheken deaktiviert.
c. Mit Logging bezeichnen wir hier die Ausgabe von Statusinformationen zur Laufzeit des generierten Codes (gesteuert durch die Kommandozeilen-Option „`-v`“ der generierten Implementierung, siehe Abschnitt 3.6)
d. Steuert, ob vom Codegenerator erzeugte Prototypen für „`PRIMITIVE`“- oder „`EXTERNAL`“-attributierte Estelle-Funktionen aktiviert werden.
e. Steuert, ob C++-Quellcode aus Pseudokommentaren der Estelle-Spezifikation („`{$C$ ... }`“) übernommen wird.
f. Erzeugt lesbare Ausgabe von Daten (z. B. Interaktionsparameter).
g. Aktiviert Unterstützung für externen XEC-Debugger (z. B. EGD, siehe Abschnitt 3.6.5).
h. Aktiviert Erzeugung von „timed traces“ für Performance-Monitoring (siehe Abschnitt 3.6.3).

Das generierte Makefile selbst enthält im Wesentlichen nur die Definition einiger Variablen und Anweisungen zur textuellen Inklusion von vorgefertigten Makefile-Templates und Parameterdateien aus dem XEC-Installationsverzeichnis.[14] In Anhang A.8.1 ist das Makefile eines XEC-Aufrufs wiedergegeben:

Die Angabe der Option „`-m`“ beim Aufruf von XEC steuert direkt den Wert von „`MAKE_OPTION`“ in dieser Datei, analog steuert die Option „`-l`“ den Wert von „`XEC`“. Die Anpassung existierender bzw. die Erzeugung neuer Options-Templates erfolgt also durch Neuanlage bzw. Änderung der jeweiligen Dateien „`$(XEC)/xecmk_$(MAKE_OPTION).mk`“.

Die darin wiederum inkludierte Datei „`$(XEC)/xecmk__configure.mk`“ stammt aus der automatischen Konfiguration[15] von XEC auf die jeweilige Plattform bei dessen Übersetzung und beinhaltet z. B. die für den C++-Compiler spezifischen Optionen. So enthält allein diese zentrale Konfigurationsdatei im Falle von `gcc` z. B. die Zuordnung von „`-O2`“ auf die Variable

14. Diese wird bei der Konfiguration zum Zeitpunkt der Installation von XEC festgelegt. Die Quelle der inkludierten Dateien kann nachträglich mit der Option „`-l <dir>`“ oder durch Anpassung des generierten Makefiles gesteuert werden.

CXX_OPTIMIZE sowohl für die Übersetzung von XEC selbst, wie auch für die Übersetzung der generierten Implementierungen und unterstützt so die Portabilität des Systems erheblich (siehe Anhang A.8.4).

Die zuletzt inkludierte Datei „$(XEC)/xecmk__base.mk“ beinhaltet die eigentlichen Abhängigkeiten und Übersetzungsregeln und baut dabei auf die Werte der zuvor gesetzten Variablen auf (siehe Anhang A.8.3).

A.8.1: Generiertes Beispiel-Makefile „test.mk“

```
# makefile generated for test.obj by XEC (V 1.3.3)

# basename of generated files
DEST            = test

# paths to source (c++) and destination (binary) of library files
XEC             = /local/lib/xec/1.3.3

# name-extension for makefile-include (see below)
MAKE_OPTION     = default

# --- begin imported file "/local/lib/xec/1.3.3/xecmk_default.mk" ---

include $(XEC)/xecmk__configure.mk

### more logging / data-dumping?
  CPPFLAGS  += -DLOGGING
  CPPFLAGS  += -DPRETTYPRINT

### use your "{$C$...} inline code?
  CPPFLAGS  += -DINLINE_C

### create prototypes for primitives/externals?
  CPPFLAGS  += -DPROTOTYPES

### more compiler optimization and less self-checks?
# CXXFLAGS  += $(CXX_OPTIMIZE)
# CPPFLAGS  += -DNDEBUG

### support for machine-level debugging / heap-analysis
  CXXFLAGS  += $(CXX_DEBUG)
# LIB_CC    += $(XEC)/xecrt_heapdebug.cc
```

15. Bei der Installation des XEC-Toolkits werden durch Aufruf eines „AUTOCONF“-Scriptes eine Vielzahl von Systemparametern der Zielplattform (z. B. Typ und Aufrufparameter des C++-Compilers, verfügbare Bibliotheken, etc.) bestimmt. Als eines der Ergebnisse dieses Konfigurationslaufs wird die Datei „configure.mk“ erzeugt, die u. a. die o. g. Make-Variablen für die Übersetzung des XEC-Toolkits selbst definiert. Für die Übersetzung des generierten Codes wird auf die selben Konfigurationsparameter zurückgegriffen, indem bei der Installation mit „xecrt_configure.mk“ eine Kopie dieser Datei angelegt wird.

```
### support for xec¥s debugger-interface?
# CPPFLAGS  += -DDEBUGGER
# LIB_CC    += $(XEC)/xecrt_debugger.cc
# LIB_DEP   += $(XEC)/xecrt_debugger.h
# LINKFLAGS += $(LINK_SOCKET) -lnsl

### support for xec¥s monitoring and tracing?
# CPPFLAGS  += -DMONITOR
# LIB_CC    += $(XEC)/xecrt_monitor.cc
# LIB_DEP   += $(XEC)/xecrt_monitor.h

include $(XEC)/xecmk__base.mk

# --- end imported file "/local/lib/xec/1.3.3/xecmk_default.mk" -----
```

A.8.2: Makefile-Template „xecmk_monitoring.mk“

```
# makefile: compile for "monitoring"

include $(XEC)/xecmk__configure.mk

CPPFLAGS  += -DLOGGING
CPPFLAGS  += -DPROTOTYPES
CPPFLAGS  += -DINLINE_C
CPPFLAGS  += -DPRETTYPRINT
CPPFLAGS  += -DNDEBUG
CPPFLAGS  += -DMONITOR
CXXFLAGS  += $(CXX_OPTIMIZE)
LIB_CC    += $(XEC)/xecrt_monitor.cc
LIB_DEP   += $(XEC)/xecrt_monitor.h

include $(XEC)/xecmk__base.mk
```

A.8.3: Makefile-Include „xecmk__base.mk“

```
#### base makefile ####

############ project-specific files ###############

LIB_DEP   +=  $(XEC)/xecrt_context.h\
              $(XEC)/xecrt_modules.h\
              $(XEC)/xecrt_estelle.h\
              $(XEC)/xecrt_baselib.h\
              $(XEC)/xecrt.h

LIB_CC    +=  $(XEC)/xecrt_context.cc \
              $(XEC)/xecrt_modules.cc \
              $(XEC)/xecrt_estelle.cc \
              $(XEC)/xecrt_baselib.cc
```

```
LIB_O  +=     $(notdir $(LIB_CC:.cc=.o))

LIB_A     =   libxecrt_$(MAKE_OPTION).a

CPPFLAGS  +=  $(HRTIME_AVAIL) -I$(XEC)

############ default-goal ###############

all: $(DEST)


############ rules ###############

%.o: %.cc %.h $(LIB_DEP)
     ###### COMPILE $< ######
     $(CXX) $(CXXFLAGS) $(CPPFLAGS) -o $@ $<

%.o: $(XEC)/%.cc $(XEC)/%.h $(LIB_DEP)
     ###### COMPILE $< ######
     $(CXX) $(CXXFLAGS) $(CPPFLAGS) -o $@ $<

$(LIB_A) : $(LIB_A)($(LIB_O))

$(DEST): $(DEST).o $(LIB_A)
     ###### LINK $@ ######
     $(LINK) $(LINKFLAGS) $^ -o $@
     ###### TRANSLATION COMPLETE ######
```

A.8.4: Beispiel-Makefile-Include „`configure.mk`“

```
# Generated automatically from configure.mk.in by configure.

###################### C++-compiler and linker ###################

CXX               = c++
CXXFLAGS          = -c -W -Wall
CPPFLAGS          =

HRTIME_AVAIL      = -DHRTIME
FLEXLEXER_INC     =

CXX_DEBUG         = -g
CXX_OPTIMIZE      = -O2
CXX_DEP           = -MM

X_CFLAGS          = -I/usr/openwin/include -I/usr/dt/include
X_LIBS            = -L/usr/openwin/lib -L/usr/dt/lib

LINK              = $(CXX)
LINK_SOCKET       = -lsocket
```

```
######################### tools ########################

MV                = mv -f
RM                = rm -f
CP                = cp -f
MKDIR             = mkdir
SED               = sed
BISON             = bison -d
# BISON           += --verbose --no-lines --debug
FLEX              = flex
CHMOD_EXEC        = chmod a+x
INSTALL_DIR       = ../support/install -d -m755
INSTALL_EXEC      = ../support/install -m755
INSTALL_FILE      = ../support/install -m644


######################### directories ######################

prefix            = /home/thees/local
exec_prefix       = ${prefix}
bindir            = ${exec_prefix}/bin
includedir        = ${prefix}/include
libdir            = ${exec_prefix}/lib
datadir           = ${prefix}/share

goals             = pet xec monitoring
monitoring_goals  = mondump mon2pato
```

A.9: Timed-Traces

In Abschnitt 3.6.3 wurde das Monitoring-System von XEC vorgestellt. Wir geben hier zur Illustration die Inhalte einer Ausgabedatei beispielhaft wieder.

A.9.1: Datei-Header

Nachfolgend ist als Beispiel auszugsweise der Header einer Ausgabe-Datei des Monitoring-Systems von XEC dargestellt:

```
MONITOR_LOGFILE V1.11

[INFO]
DESCRIPTION:
T_STARTED:      1100526693 = Mon Nov 15 14:51:33 2004
T_FINISHED:     1100526693 = Mon Nov 15 14:51:33 2004
STAMPS_CATCHED: 18985
STAMPS_LOST:    0
CALIBRATIONS:   100
TIMESCALE:      2524000000 TICKS PER SECOND
BYTEORDER:      04030201 0807060504030201

[TYPE DESCRIPTIONS]
#       id      description
#       --      -----------
        1       Cycle Start
        2       Deadlock
        3       Sleep Start
        4       Sleep Abort
        5       Selection Local Start
        6       Selection Children Start
        7       Selection End
        8       Fire Start
        9       Fire End
        10      Init Start
        11      Init End
        12      Fire Start
        13      Fire End
        14      Test Start
        15      Test End
        16      Explicit Trace

[STATIC DESCRIPTIONS]
#       id      class   parent  description
#       --      -----   ------  -----------
        0       SYSTEM  0       SYSTEM
        1       MODULE  0       SPEC_slidingwindowprotocol
        2       INIT    1       INIT_1
        3       MODULE  1       BODY_transmitteruserbody
        4       INIT    3       INIT_1
```

```
          5       TRANS   3       TRANS_send
          6       TRANS   3       TRANS_new_round
          7       MODULE  1       BODY_receiveruserbody
          8       INIT    7       INIT_1
          9       TRANS   7       TRANS_receive
          10      MODULE  1       BODY_umbody
          11      INIT    10      INIT_1
          12      TRANS   10      TRANS_transport_t_r
          13      TRANS   10      TRANS_transport_r_t
          14      TRANS   10      TRANS_retired
          15      MODULE  10      BODY_piperbody

  [...]

  [DYNAMIC DESCRIPTIONS]
  #       id      parent  static
  #       --      ------  ------
          0       0       0
          1       0       1
          2       1       3
          3       1       7
          4       1       23
          5       1       30
          6       1       10
          7       6       15
          8       6       15
          9       6       15
          10      1       18

  [CALIBRATION 100 24]

  [ hier folgen die Timed-Traces ... ]
```

A.9.2: Mondump

Mit dem Werkzeug `mondump` können schließlich auch die Zeitstempel lesbar ausgegeben werden:

```
No.              Stamp Type              StaticId  dId            Time                Diff
 ------------------------------------------------------------------sec. ms. us. ns------sec. ms. us. ns
    0               Init Start  SPEC_slidingwindowprotocol    1              0                   0
    1               Init Start  BODY_transmitteruserbody    2             78.716              78.716
    2                 Init End  BODY_transmitteruserbody    2            120.756              42.040
    3               Init Start  BODY_receiveruserbody    3              139.572              18.816
    4                 Init End  BODY_receiveruserbody    3              139.832                 260
    5               Init Start   BODY_transmitterbody    4              208.856              69.024
    6                 Init End   BODY_transmitterbody    4              213.128               4.272
    7               Init Start      BODY_receiverbody    5              231.104              17.976
    8                 Init End      BODY_receiverbody    5              231.708                 604
    9               Init Start           BODY_umbody    6              258.408              26.700
   10               Init Start        BODY_piperbody    7              292.820              34.412
   11                 Init End        BODY_piperbody    7              292.884                  64
   12               Init Start        BODY_piperbody    8              298.848               5.964
   13                 Init End        BODY_piperbody    8              298.908                  60
   14               Init Start        BODY_piperbody    9              306.180               7.272
   15                 Init End        BODY_piperbody    9              306.232                  52
   16                 Init End           BODY_umbody    6              307.696               1.464
   17               Init Start        BODY_timerbody   10              358.268              50.572
   18                 Init End        BODY_timerbody   10              358.888                 620
   19                 Init End  SPEC_slidingwindowprotocol    1         360.692               1.804
```

```
    20              Cycle Start                 SYSTEM    0            365.736              5.044
    21      Selection Local Start  BODY_transmitteruserbody    2            374.284              8.548
    22               Test Start             TRANS_send    2            380.284              6.000
    23                 Test End             TRANS_send    2            380.800                516
    24            Selection End  BODY_transmitteruserbody    2            381.756                956
    25               Fire Start  BODY_transmitteruserbody    2            382.232                476
    26               Fire Start             TRANS_send    2            384.280              2.048
    27                 Fire End             TRANS_send    2            387.932              3.652
    28                 Fire End  BODY_transmitteruserbody    2            388.012                 80
    29              Cycle Start                 SYSTEM    0            394.648              6.636
    30      Selection Local Start   BODY_transmitterbody    4            394.920                272
    31            Selection End   BODY_transmitterbody    4            397.644              2.724
    32               Fire Start   BODY_transmitterbody    4            397.844                200

[ .... ]
```

Anhang B: Benchmarkspezifikationen

B.1: Datenübertragungsbenchmark (Standard-Version)

In diesem Abschnitt ist der Estelle-Spezifikationstext des in Abschnitt 2.3.3 bzw. in Abschnitt 6.2.4 eingeführten Benchmarks zur Bewertung der Effizienz von Datenübertragungsmechanismen in Protokoll-Implementierungen wiedergegeben.

```
 1: SPECIFICATION TestSpec;
 2:
 3: DEFAULT COMMON QUEUE;
 4: {TIMESCALE seconds;}
 5:
 6: CONST UserDataSize      = 1000;
 7: CONST Hops              = 5;
 8: CONST Rounds            = 10000;
 9: CONST Resend_Buffer_Size= 3;
10:
11:
12: TYPE T_Byte = 0..255;
13: TYPE T_UserData = ARRAY [1 .. UserDataSize] OF T_Byte;
14:
15:
16:
17:
18:
19: { define PDU- and SDU-types with some dummy header and trailer }
20:
21: TYPE SDU_T_3 = T_UserData;
22:
23:
24: TYPE PDU_T_3 =  RECORD  header: INTEGER;
25:                         payload: SDU_T_3;
26:                         trailer: INTEGER
27:                 END;
28:
29: TYPE SDU_T_2 =  PDU_T_3;
30: TYPE PDU_T_2 =  RECORD  header: INTEGER;
31:                         payload: SDU_T_2;
32:                         trailer: INTEGER
33:                 END;
34:
35: TYPE SDU_T_1 =  PDU_T_2;
36: TYPE PDU_T_1 =  RECORD  header: INTEGER;
37:                         payload: SDU_T_1;
38:                         trailer: INTEGER
39:                 END;
40:
41: TYPE SDU_T_0 =  PDU_T_1;
```

```
42:
43:
44: { define channels between PMs }
45:
46: CHANNEL ch_3(user, provider);
47:   BY user,provider: msg(data: SDU_T_3);
48:
49: CHANNEL ch_2(user, provider);
50:   BY user,provider: msg(data: SDU_T_2);
51:
52: CHANNEL ch_1(user, provider);
53:   BY user,provider: msg(data: SDU_T_1);
54:
55: CHANNEL ch_0(user, provider);
56:   BY user,provider: msg(data: SDU_T_0);
57:
58:
59: { define external interfaces for users }
60:
61: MODULE hdr_usr systemactivity(send: boolean; rounds: INTEGER);
62:   IP ipdown: ch_3(user);
63:   END;
64:
65:
66: { define bodies for users }
67:
68: BODY body_usr FOR hdr_usr;
69:
70:   VAR data_buf: T_UserData;
71:
72:   INITIALIZE
73:     BEGIN
74:       ALL i: 1 .. UserDataSize DO
75:         data_buf[i] := 99;
76:     END;
77:
78:   TRANS
79:     PROVIDED send AND (rounds > 0)
80:       NAME sending:
81:       BEGIN
82:         { send data from local variable }
83:         OUTPUT ipdown.msg(data_buf);
84:         send := false;
85:         rounds := rounds - 1;
86:       END;
87:
88:
89:   TRANS
90:     WHEN ipdown.msg
91:       NAME receiving:
92:       BEGIN
93:         { store data in local variable }
94:         data_buf := data;
95:         send := true;
96:       END;
97:
98: END; {BODY}
99:
```

```
100:
101: { define external interfaces of PMs }
102:
103: MODULE hdr_PM_3 systemactivity;
104:   IP ipup:   ch_3(provider);
105:      ipdown: ch_2(user);
106:   END;
107:
108: MODULE hdr_PM_2 systemactivity;
109:   IP ipup:   ch_2(provider);
110:      ipdown: ch_1(user);
111:   END;
112:
113: MODULE hdr_PM_1 systemactivity;
114:   IP ipup:   ch_1(provider);
115:      ipdown: ch_0(user);
116:   END;
117:
118:
119: { define bodies of PMs }
120:
121: BODY body_PM_3 FOR hdr_PM_3;
122:
123:   TRANS
124:     WHEN ipup.msg
125:       VAR pdu: PDU_T_3;
126:       NAME frame:
127:       BEGIN
128:         { frame sdu (down) }
129:         pdu.header  := 0;
130:         pdu.payload := data;
131:         pdu.trailer := 0;
132:         OUTPUT ipdown.msg(pdu);
133:       END;
134:
135:   TRANS
136:     WHEN ipdown.msg
137:       NAME unframe:
138:       BEGIN
139:         { unframe sdu (up) }
140:         OUTPUT ipup.msg(data.payload);
141:       END;
142:
143: END; {BODY}
144:
145:
146: BODY body_PM_2 FOR hdr_PM_2;
147:
148:   VAR
149:         Resend_Buffer: ARRAY [1 .. Resend_Buffer_Size] OF PDU_T_2;
150:         Resend_Buffer_Idx: 1 .. Resend_Buffer_Size;
151:
152:   INITIALIZE
153:     BEGIN
154:       Resend_Buffer_Idx := 1;
155:     END;
156:
157:   TRANS
```

```
158:     WHEN ipup.msg
159:       VAR pdu: PDU_T_2;
160:       NAME frame:
161:       BEGIN
162:         { frame sdu and send (down) }
163:         pdu.header  := 0;
164:         pdu.payload := data;
165:         pdu.trailer := 0;
166:         OUTPUT ipdown.msg(pdu);
167:
168:         { save PDU for possible resend (not implemented) }
169:         Resend_Buffer[Resend_Buffer_Idx] := pdu;
170:         Resend_Buffer_Idx := (Resend_Buffer_Idx MOD Resend_Buffer_Size) + 1
;
171:       END;
172:
173:   TRANS
174:     WHEN ipdown.msg
175:       NAME unframe:
176:       BEGIN
177:         { unframe sdu (up) }
178:         OUTPUT ipup.msg(data.payload);
179:       END;
180:
181: END; {BODY}
182:
183:
184: BODY body_PM_1 FOR hdr_PM_1;
185:
186:   TRANS
187:     WHEN ipup.msg
188:       VAR pdu: PDU_T_1;
189:       NAME frame:
190:       BEGIN
191:         { frame sdu (down) }
192:         pdu.header  := 0;
193:         pdu.payload := data;
194:         pdu.trailer := 0;
195:         OUTPUT ipdown.msg(pdu);
196:       END;
197:
198:   TRANS
199:     WHEN ipdown.msg
200:       NAME unframe:
201:       BEGIN
202:         { unframe sdu (up) }
203:         OUTPUT ipup.msg(data.payload);
204:       END;
205:
206: END; {BODY}
207:
208:
209: { define external interfaces for "hop"-chain }
210:
211: MODULE hdr_router systemactivity;
212:   IP ipdown: ARRAY [0..1] OF ch_0(user);
213:   END;
214:
```

```
215: MODULE hdr_net systemactivity;
216:   IP ipup: ARRAY [0..1] OF ch_0(provider);
217:   END;
218:
219:
220: { define bodies for "hop"-chain }
221:
222: BODY body_router FOR hdr_router;
223:
224:   TRANS
225:     ANY i: 0..1 DO
226:       WHEN ipdown[i].msg
227:         VAR pdu: PDU_T_1;
228:         NAME route:
229:         BEGIN
230:           { unframe and reframe sdu }
231:           pdu.payload := data.payload;
232:           OUTPUT ipdown[1-i].msg(pdu);
233:         END;
234:
235: END; {BODY}
236:
237: BODY body_net FOR hdr_net;
238:
239:   TRANS
240:     ANY i: 0..1 DO
241:       WHEN ipup[i].msg
242:         NAME transmit:
243:         BEGIN
244:           { just transmit sdu }
245:           OUTPUT ipup[1-i].msg(data);
246:         END;
247:
248: END; {BODY}
249:
250:
251: { main }
252:
253: MODVAR
254:   net:    ARRAY [0..10] OF hdr_net;
255:   router: ARRAY [1..10] OF hdr_router;
256:   pm_1:   ARRAY [0..1] OF hdr_PM_1;
257:   pm_2:   ARRAY [0..1] OF hdr_PM_2;
258:   pm_3:   ARRAY [0..1] OF hdr_PM_3;
259:   user:   ARRAY [0..1] OF hdr_usr;
260:
261: INITIALIZE
262:   BEGIN
263:   { create and connect both protocol stacks }
264:   ALL i: 0..1 DO
265:     BEGIN
266:     INIT pm_1[i] WITH body_PM_1;
267:     INIT pm_2[i] WITH body_PM_2;
268:     INIT pm_3[i] WITH body_PM_3;
269:     INIT user[i] WITH body_usr(i = 0, Rounds);
270:     CONNECT pm_1[i].ipup TO pm_2[i].ipdown;
271:     CONNECT pm_2[i].ipup TO pm_3[i].ipdown;
272:     CONNECT pm_3[i].ipup TO user[i].ipdown;
```

```
273:     END;
274:
275:   { create Network "hop"-chain }
276:   ALL i: 0..Hops DO
277:     INIT net[i] WITH body_net;
278:   ALL i: 1..Hops DO
279:     BEGIN
280:     INIT router[i] WITH body_router;
281:     CONNECT router[i].ipdown[0] TO net[i-1].ipup[1];
282:     CONNECT router[i].ipdown[1] TO net[i  ].ipup[0];
283:     END;
284:   CONNECT pm_1[0].ipdown TO net[   0].ipup[0];
285:   CONNECT pm_1[1].ipdown TO net[Hops].ipup[1];
286:   END; { TRANS }
287:
288: END. { SPEC }
289:
```

B.2: Datenübertragungsbenchmark (Explizite Referenzübergabe / ALF)

In diesem Abschnitt ist der Estelle-Spezifikationstext des in Abschnitt 6.4.5 eingesetzten Benchmarks zur Bewertung der Effizienz des Datenübertragungsmechanismus „Explizite Referenzübergabe“ wiedergegeben. Der Benchmark entspricht dem in Anhang B.1 spezifizierten Funktionsumfang und basiert auf der expliziten Referenzübergabe eines globalen „Application Layer Framing“ Puffers (siehe auch Abschnitt 6.3.2).

Die modifizierten Teile sind unterstrichen dargestellt.

```
 1: SPECIFICATION TestSpec;
 2:
 3: DEFAULT COMMON QUEUE;
 4: {TIMESCALE seconds;}
 5:
 6: CONST UserDataSize      = 1000;
 7: CONST Hops              = 5;
 8: CONST Rounds            = 10000;
 9: CONST Resend_Buffer_Size= 3;
10:
11:
12: TYPE T_Byte = 0..255;
13: TYPE T_UserData = ARRAY [1 .. UserDataSize] OF T_Byte;
14:
15:
16:
17:
18:
19: { define PDU- and SDU-types with some dummy header and trailer }
20:
21: TYPE SDU_T_3 = T_UserData;
22: {TYPE SDU_T_3 = T_UserData --or-- TYPE SDU_T_3 = ANY TYPE;}
23:
24: TYPE PDU_T_3 =  RECORD  header: INTEGER;
25:                         payload: SDU_T_3;
26:                         trailer: INTEGER
27:                 END;
28:
29: TYPE SDU_T_2 =  PDU_T_3;
30: TYPE PDU_T_2 =  RECORD  header: INTEGER;
31:                         payload: SDU_T_2;
32:                         trailer: INTEGER
33:                 END;
34:
35: TYPE SDU_T_1 =  PDU_T_2;
36: TYPE PDU_T_1 =  RECORD  header: INTEGER;
37:                         payload: SDU_T_1;
38:                         trailer: INTEGER
39:                 END;
40:
41: TYPE SDU_T_0 =  PDU_T_1;
42: TYPE P_SDU_T_0= ^SDU_T_0;
43:
44:
```

```
45: { define channels between PMs }
46:
47: CHANNEL ch_3(user, provider);
48:   BY user,provider: msg(p: P_SDU_T_0);
49:
50: CHANNEL ch_2(user, provider);
51:   BY user,provider: msg(p: P_SDU_T_0);
52:
53: CHANNEL ch_1(user, provider);
54:   BY user,provider: msg(p: P_SDU_T_0);
55:
56: CHANNEL ch_0(user, provider);
57:   BY user,provider: msg(p: P_SDU_T_0);
58:
59:
60: { define external interfaces for users }
61:
62: MODULE hdr_usr systemactivity(send: boolean; rounds: INTEGER);
63:   IP ipdown: ch_3(user);
64:   END;
65:
66:
67: { define bodies for users }
68:
69: BODY body_usr FOR hdr_usr;
70:
71:   VAR data_buf: T_UserData;
72:
73:   INITIALIZE
74:     BEGIN
75:       ALL i: 1 .. UserDataSize DO
76:         data_buf[i] := 99;
77:     END;
78:
79:   TRANS
80:     PROVIDED send AND (rounds > 0)
81:       VAR p: P_SDU_T_0;
82:         NAME sending:
83:         BEGIN
84:           { send data from local variable }
85:           { here: alocate ALF-buffer on heap ... }
86:           new(p);
87:           { store payload ... }
88:           p^.payload.payload.payload := data_buf;
89:           { send ALF-buffer-reference (disposed) ... }
90:           OUTPUT ipdown.msg(p);
91:           send := false;
92:           rounds := rounds - 1;
93:         END;
94:
95:
96:   TRANS
97:     WHEN ipdown.msg
98:       NAME receiving:
99:       BEGIN
100:        { store data in local variable }
101:        { here: read from ALF-buffer ... }
102:        data_buf := p^.payload.payload.payload;
```

```
103:         { and dispose buffer ... }
104:         dispose(p);
105:         send := true;
106:       END;
107:
108: END; {BODY}
109:
110:
111: { define external interfaces of PMs }
112:
113: MODULE hdr_PM_3 systemactivity;
114:   IP ipup:   ch_3(provider);
115:      ipdown: ch_2(user);
116:   END;
117:
118: MODULE hdr_PM_2 systemactivity;
119:   IP ipup:   ch_2(provider);
120:      ipdown: ch_1(user);
121:   END;
122:
123: MODULE hdr_PM_1 systemactivity;
124:   IP ipup:   ch_1(provider);
125:      ipdown: ch_0(user);
126:   END;
127:
128:
129: { define bodies of PMs }
130:
131: BODY body_PM_3 FOR hdr_PM_3;
132:
133:   TRANS
134:     WHEN ipup.msg
135:       NAME frame:
136:       BEGIN
137:         { frame sdu (down) }
138:         { here: just setup ALF-buffer }
139:         p^.payload.payload.header  := 0;
140:         p^.payload.payload.trailer := 0;
141:         { p^.payload.payload.payload is already prepared! }
142:         { send ALF-buffer-reference (disposed) ... }
143:         OUTPUT ipdown.msg(p);
144:       END;
145:
146:   TRANS
147:     WHEN ipdown.msg
148:       NAME unframe:
149:       BEGIN
150:         { unframe sdu (up) }
151:         { here: just send ALF-buffer-reference (disposed) ... }
152:         OUTPUT ipup.msg(p);
153:       END;
154:
155: END; {BODY}
156:
157:
158: BODY body_PM_2 FOR hdr_PM_2;
159:
160:   VAR
```

```
161:         { resend buffer now stores complete ALF-frames }
162:         Resend_Buffer: ARRAY [1 .. Resend_Buffer_Size] OF SDU_T_0;
163:         Resend_Buffer_Idx: 1 .. Resend_Buffer_Size;
164:
165:   INITIALIZE
166:     BEGIN
167:       Resend_Buffer_Idx := 1;
168:     END;
169:
170:   TRANS
171:     WHEN ipup.msg
172:       NAME frame:
173:       BEGIN
174:         { save PDU for possible resend (by copying the ALF-buffer) }
175:         Resend_Buffer[Resend_Buffer_Idx] := p^;
176:         Resend_Buffer_Idx := (Resend_Buffer_Idx MOD Resend_Buffer_Size) + 1
;
177:
178:         { frame sdu (down) }
179:         { here: just setup ALF-buffer }
180:         p^.payload.header  := 0;
181:         p^.payload.trailer := 0;
182:         { p^.payload.payload is already prepared! }
183:         { send ALF-buffer-reference (disposed) ... }
184:         OUTPUT ipdown.msg(p);
185:       END;
186:
187:   TRANS
188:     WHEN ipdown.msg
189:       NAME unframe:
190:       BEGIN
191:         { unframe sdu (up) }
192:         { here: just send ALF-buffer-reference (disposed) ... }
193:         OUTPUT ipup.msg(p);
194:       END;
195:
196: END; {BODY}
197:
198:
199: BODY body_PM_1 FOR hdr_PM_1;
200:
201:   TRANS
202:     WHEN ipup.msg
203:       NAME frame:
204:       BEGIN
205:         { frame sdu (down) }
206:         { here: just setup ALF-buffer }
207:         p^.header  := 0;
208:         p^.trailer := 0;
209:         { p^.payload is already prepared! }
210:         { send ALF-buffer-reference (disposed) ... }
211:         OUTPUT ipdown.msg(p);
212:       END;
213:
214:   TRANS
215:     WHEN ipdown.msg
216:       NAME unframe:
217:       BEGIN
```

```
218:         { unframe sdu (up) }
219:         { here: just send ALF-buffer-reference (disposed) ... }
220:         OUTPUT ipup.msg(p);
221:       END;
222:
223: END; {BODY}
224:
225:
226: { define external interfaces for "hop"-chain }
227:
228: MODULE hdr_router systemactivity;
229:   IP ipdown: ARRAY [0..1] OF ch_0(user);
230:   END;
231:
232: MODULE hdr_net systemactivity;
233:   IP ipup: ARRAY [0..1] OF ch_0(provider);
234:   END;
235:
236:
237: { define bodies for "hop"-chain }
238:
239: BODY body_router FOR hdr_router;
240:
241:   TRANS
242:     ANY i: 0..1 DO
243:       WHEN ipdown[i].msg
244:         NAME route:
245:         BEGIN
246:           { unframe and reframe sdu }
247:           { here: just forward ALF-buffer-reference (disposed) ... }
248:           OUTPUT ipdown[1-i].msg(p);
249:         END;
250:
251: END; {BODY}
252:
253: BODY body_net FOR hdr_net;
254:
255:   TRANS
256:     ANY i: 0..1 DO
257:       WHEN ipup[i].msg
258:         NAME transmit:
259:         BEGIN
260:           { just transmit sdu }
261:           OUTPUT ipup[1-i].msg(p);
262:         END;
263:
264: END; {BODY}
265:
266:
267: { main }
268:
269: MODVAR
270:   net:    ARRAY [0..10] OF hdr_net;
271:   router: ARRAY [1..10] OF hdr_router;
272:   pm_1:   ARRAY [0..1] OF hdr_PM_1;
273:   pm_2:   ARRAY [0..1] OF hdr_PM_2;
274:   pm_3:   ARRAY [0..1] OF hdr_PM_3;
275:   user:   ARRAY [0..1] OF hdr_usr;
```

```
276:
277: INITIALIZE
278:   BEGIN
279:   { create and connect both protocol stacks }
280:   ALL i: 0..1 DO
281:     BEGIN
282:     INIT pm_1[i] WITH body_PM_1;
283:     INIT pm_2[i] WITH body_PM_2;
284:     INIT pm_3[i] WITH body_PM_3;
285:     INIT user[i] WITH body_usr(i = 0, Rounds);
286:     CONNECT pm_1[i].ipup TO pm_2[i].ipdown;
287:     CONNECT pm_2[i].ipup TO pm_3[i].ipdown;
288:     CONNECT pm_3[i].ipup TO user[i].ipdown;
289:     END;
290:
291:   { create Network "hop"-chain }
292:   ALL i: 0..Hops DO
293:     INIT net[i] WITH body_net;
294:   ALL i: 1..Hops DO
295:     BEGIN
296:     INIT router[i] WITH body_router;
297:     CONNECT router[i].ipdown[0] TO net[i-1].ipup[1];
298:     CONNECT router[i].ipdown[1] TO net[i  ].ipup[0];
299:     END;
300:   CONNECT pm_1[0].ipdown TO net[   0].ipup[0];
301:   CONNECT pm_1[1].ipdown TO net[Hops].ipup[1];
302:   END; { TRANS }
303:
304: END. { SPEC }
305:
```

B.3: Datenübertragungsbenchmark (Containertyp-Version)

In diesem Abschnitt ist der Estelle-Spezifikationstext des in Abschnitt 6.4.6 eingesetzten Benchmarks zur Bewertung der Effizienz des Datenübertragungsmechanismus der verdeckten Referenzübergabe auf Basis der in Abschnitt 5.2.2 eingeführten Containertyp-Erweiterung (ANY TYPE) wiedergegeben. Der Benchmark entspricht der in Anhang B.1 angegebenen Spezifikation vollständig bis auf die modifizierte Typdefinition in Zeile 21 und wird entsprechend an dieser Stelle nur auszugsweise wiedergeben.

Die modifizierten Teile sind unterstrichen dargestellt.

```
 1: SPECIFICATION TestSpec;
 2:
 3: DEFAULT COMMON QUEUE;
 4: {TIMESCALE seconds;}
 5:
 6: CONST UserDataSize      = 1000;
 7: CONST Hops              = 5;
 8: CONST Rounds            = 10000;
 9: CONST Resend_Buffer_Size= 3;
10:
11:
12: TYPE T_Byte = 0..255;
13: TYPE T_UserData = ARRAY [1 .. UserDataSize] OF T_Byte;
14:
15:
16:
17:
18:
19: { define PDU- and SDU-types with some dummy header and trailer }
20:
21: TYPE SDU_T_3 = ANY TYPE;
22:
23:
24: TYPE PDU_T_3 =  RECORD  header: INTEGER;
25:                         payload: SDU_T_3;
26:                         trailer: INTEGER
27:                 END;
28:
29: TYPE SDU_T_2 =  PDU_T_3;
30: TYPE PDU_T_2 =  RECORD  header: INTEGER;
31:                         payload: SDU_T_2;
32:                         trailer: INTEGER
33:                 END;
34:
35: TYPE SDU_T_1 =  PDU_T_2;
36: TYPE PDU_T_1 =  RECORD  header: INTEGER;
37:                         payload: SDU_T_1;
38:                         trailer: INTEGER
39:                 END;
40:
41: TYPE SDU_T_0 =  PDU_T_1;
42:
43:
```

```
44: { define channels between PMs }
45:
46: CHANNEL ch_3(user, provider);
47:   BY user,provider: msg(data: SDU_T_3);
48:
49: CHANNEL ch_2(user, provider);
50:   BY user,provider: msg(data: SDU_T_2);
51:
52: CHANNEL ch_1(user, provider);
53:   BY user,provider: msg(data: SDU_T_1);
54:
55: CHANNEL ch_0(user, provider);
56:   BY user,provider: msg(data: SDU_T_0);
57:
58:
59: { define external interfaces for users }
60:
61: MODULE hdr_usr systemactivity(send: boolean; rounds: INTEGER);
62:   IP ipdown: ch_3(user);
63:   END;
64:
65:
66: { define bodies for users }
67:
68: BODY body_usr FOR hdr_usr;
69:
70:   VAR data_buf: T_UserData;
71:
72:   INITIALIZE
73:     BEGIN
74:       ALL i: 1 .. UserDataSize DO
75:         data_buf[i] := 99;
76:     END;
77:
78:   TRANS
79:     PROVIDED send AND (rounds > 0)
80:       NAME sending:
81:       BEGIN
82:         { send data from local variable }
83:         OUTPUT ipdown.msg(data_buf);          (* here: any-type-conversion *)
84:         send := false;
85:         rounds := rounds - 1;
86:       END;
87:
88:
89:   TRANS
90:     WHEN ipdown.msg
91:       NAME receiving:
92:       BEGIN
93:         { store data in local variable }
94:         data_buf := data;                     (* here: any-type-conversion *)
95:         send := true;
96:       END;
97:
98: END; {BODY}
99:
```

[...]

B.4: Sliding-Window-Benchmark

In diesem Abschnitt ist der Estelle-Spezifikationstext des in Abschnitt 2.3.3 eingeführten Benchmarks wiedergegeben.

```
{$C$
        #define nLog 0
        /* >=1: rounds, >=2: users, >=3: net, 4: Timeouts */
        #include "messung.c"
}

specification SlidingWindowProtocol;

{Originally from K.J. Turner: "Using Formal Description Techniques",
 Wiley, 1993}
{Extended by J. Thees}

default common queue;

(********************)
timescale milliseconds;
(********************)

(********************)
const
  TWSMax = 20;                  { Maximum window size (sender) }
  RWSMax = 20;                  { Maximum window size (receiver) }
  retrans_timeout = 300;        { timeout for packet retransmission }
  medium_delay = 0;             { delay for every packet }
  send_delay = 0;               { delay between packets sended }
  send_count = 10;              { number of packets to send in one burst }
  rounds = 10;                  { number of bursts }
  numpiper = 3;
(********************)

type
  SeqType = integer;
  UserDataType =
   (********************)
    record
      num: INTEGER;
    end;
   (********************)
  DTPDUType =
    record
      Seq: SeqType;
      Msg: UserDataType;
    end;
  AKPDUType =
    record
      Seq: SeqType;
    end;


(********************)
channel SigUser(TUser, RUser);
```

```
  by RUser:
    Complete;
(********************)

channel TxUser(User, Transmitter);
  by User:
    SDTreq(Data: UserDataType);

channel RxUser(User, Receiver);
  by Receiver:
    SDTind(Data: UserDataType);

channel Tx(Transmitter, Medium);
  by Transmitter:
    MDTreq(PDU: DTPDUType);
  by Medium:
    MAKind(PDU: AKPDUType);

channel Rx(Receiver, Medium);
  by Receiver:
    MAKreq(PDU: AKPDUType);
  by Medium:
    MDTind(PDU: DTPDUType);

channel Time(Transmitter, Timer);
  by Transmitter:
    TimeReq(Seq: SeqType);
    TimeCanc(Seq: SeqType);
  by Timer:
    TimeResp(Seq: SeqType);

module TransmitterUser systemactivity;
  ip ST: TxUser(User);
  (********************)
       UU: SigUser(TUser);
  (********************)
end;

body TransmitterUserBody for TransmitterUser;
  (********************)
  var n: integer;
      round: integer;

  initialize
    begin
    n := 0;
    round := 1;
    {$C$ if (nLog >= 1) cout << "==== ROUND "
                             << parent()->VAR_round << " ====\n";}
    {$C$ start();}
    end;

  trans
    provided n < send_count
      delay (send_delay)
        var data: UserDataType;
        name send:
        begin
```

```
108:          n := n+1;
109:          data.num := n;
110:          output ST.SDTreq(data);
111:          {$C$ if (nLog>=2) cout << "SEND(" << VAR_data.COMP_num << ")'";}
112:          end;
113:
114:    trans
115:      when UU.Complete
116:          name new_round:
117:          begin
118:                  begin
119:                  {$C$ stop();}
120:                  end;
121:          if round < rounds then
122:            begin
123:            n := 0;
124:            round := round+1;
125:                  begin
126:                  {$C$ if (nLog >= 1) cout << "==== ROUND "
                                              << parent()->VAR_round
                                              << " ====';}
127:                  {$C$ start();}
128:                  end;
129:            end
130:          else
131:            begin
132:            {$C$ save("messung.log");}
133:            end;
134:          end;
135:
136:    (********************)
137:    end;
138:
139: module ReceiverUser systemactivity;
140:    ip SR: RxUser(User);
141:    (********************)
142:          UU: SigUser(RUser);
143:    (********************)
144: end;
145:
146: body ReceiverUserBody for ReceiverUser;
147:    (********************)
148:    var n: integer;
149:
150:    initialize
151:      begin
152:      n := 0;
153:      end;
154:
155:    trans
156:      when SR.SDTind(Data)
157:        name receive:
158:        begin
159:        n := n+1;
160:        if n = send_count then
161:          begin
162:          n := 0;
163:          output UU.Complete;
```

```
164:         end;
165:       {$C$ if (nLog>=2) cout << "              RECEIVE("
                                  << param().PAR_data.COMP_num << ")'";}
166:       end;
167:
168:   (*******************)
169:   end;
170:
171:
172:
173: (***************************** Network ************************)
174:
175:
176: module UM systemactivity;
177:   ip
178:     MT: Tx(Medium);
179:     MR: Rx(Medium);
180: end;
181:
182: body UMBody for UM;
183:   (*******************)
184:
185:         module piper independent activity;
186:           ip
187:             MT: Tx(Transmitter);
188:             MR: Tx(Medium);
189:         end;
190:
191:         body piperBody for piper;
192:           trans
193:               when MT.MAKind(PDU)
194:                 name pipe_t_r:
195:                 begin
196:                   output MR.MAKind(PDU);
197:                 end;
198:
199:           trans
200:               when MR.MDTreq(PDU)
201:                 name pipe_r_t:
202:                 begin
203:                   output MT.MDTreq(PDU);
204:                 end;
205:         end; (* module piperBody *)
206:
207:
208:   var
209:     retire: array [0..1] of boolean;
210:
211:   modvar pipers: array [1..numpiper] of piper;
212:   ip myMR: Tx(Medium);
213:
214:   initialize
215:     var i: integer;
216:     begin
217:     retire[0] := false;
218:     retire[1] := false;
219:     for i:= 1 to numpiper do
220:       begin
```

```
221:       init pipers[i] with piperBody;
222:       if i > 1 then
223:         connect pipers[i-1].MT to pipers[i].MR;
224:       end;
225:     attach MT to pipers[1].MR;
226:     connect myMR to pipers[numpiper].MT;
227:     end;
228:
229:   trans
230:     provided not retire[0]
231:       when myMR.MDTreq(PDU)
232:         name transport_t_r:
233:         begin
234:           {$C$ if (nLog>=3) cout << "        MSG("
                                    << param().PAR_pdu.COMP_seq << ")-->'";}
235:           retire[0] := (medium_delay>0);
236:           output MR.MDTind(PDU);
237:         end;
238:
239:   trans
240:     provided not retire[1]
241:       when MR.MAKreq(PDU)
242:         name transport_r_t:
243:         begin
244:           {$C$ if (nLog>=3) cout << "     <--ACK("
                                    << param().PAR_pdu.COMP_seq << ")'";}
245:           retire[1] := (medium_delay>0);
246:           output myMR.MAKind(PDU);
247:         end;
248:
249:   trans
250:     any i: 0..1 do
251:       provided retire[i]
252:         delay (medium_delay)
253:           name retired:
254:           begin
255:             retire[i] := false;
256:           end;
257:
258:   (*******************)
259:   end;
260:
261:
262:
263:
264: (************************** Timer ***************************)
265:
266:
267:
268: module Timer systemactivity;
269:   ip T: Time(Timer);
270: end;
271:
272: body TimerBody for Timer;
273:   (*******************)
274:   var running: array [0..TWSMax] of SeqType;
275:
276:   initialize
```

```
277:     begin
278:     all i: 0..TWSMax do
279:       running[i] := -1;
280:     end;
281:
282:   trans
283:     when T.TimeReq(Seq)
284:       name timer_start:
285:       begin
286:         running[seq mod TWSMax] := Seq;
287:       end;
288:
289:   trans
290:     when T.TimeCanc(Seq)
291:       name timer_stop:
292:       begin
293:         running[seq mod TWSMax] := -1;
294:       end;
295:
296:   trans
297:     any i: 1..TWSMax do
298:       provided running[i] >= 0
299:         delay (retrans_timeout)
300:           name timeout_detected:
301:           begin
302:           {$C$ if (nLog>=4) cout << "TIMEOUT: "
                                << parent()->VAR_running[ANYPAR_i] << endl;}
303:           output T.TimeResp(running[i]);
304:           running[i] := -1;
305:           end;
306:
307:   (********************)
308:   end;
309:
310: module Transmitter systemactivity;
311:   ip
312:     ST: TxUser(Transmitter);
313:     MT: Tx(Transmitter);
314:     T:  Time(Transmitter);
315: end;
316:
317:
318:
319:
320: { ------------ Transmitter module body ---------------}
321:
322: body TransmitterBody for Transmitter;
323:
324:   state SENDING;
325:
326:   (********************)
327:   var PacketBuffer: array [0..TWSMax] of UserDataType;
328:   (********************)
329:
330:   procedure BuffSave(s: SeqType; d: UserDataType);
331:     begin
332:     (********************)
333:     PacketBuffer[s mod TWSMax] := d;
```

```
334:     (********************)
335:     end;
336:
337:   procedure BuffFree(s: SeqType);
338:     begin
339:     (********************)
340:     {$C$ (void)PAR_s; /* avoid warning */}
341:     (********************)
342:     end;
343:
344:   function BuffRetrieve(s: SeqType): UserDataType;
345:     begin
346:     (********************)
347:     BuffRetrieve := PacketBuffer[s mod TWSMax];
348:     (********************)
349:     end;
350:
351:   function Corrupted(PDU: AKPDUType): boolean;
352:     begin
353:     (********************)
354:     {$C$ (void)PAR_pdu; /* avoid warning */}
355:     Corrupted := false;
356:     (********************)
357:     end;
358:
359:   { construct a DT PDU froom the user and sequence number }
360:   function PDUDT(s: SeqType; d: UserDataType): DTPDUType;
361:     var pdu: DTPDUType;
362:     begin
363:     (********************)
364:     pdu.seq := s;
365:     pdu.msg := d;
366:     PDUDT := pdu;
367:     (********************)
368:     end;
369:
370:   var
371:     LowestUnacked: SeqType;
372:     HighestSent: SeqType;
373:     TWS: integer;
374:
375:   initialize
376:     to SENDING
377:       provided TWSMax > 0
378:         begin
379:           LowestUnacked := 1;
380:           HighestSent := 0;
381:           TWS := TWSMax;
382:         end;
383:
384:   trans
385:
386:     { Transmit while window not full}
387:     from SENDING to same
388:       when ST.SDTreq
389:         provided HighestSent - LowestUnacked + 1 < TWS
390:           begin
391:           HighestSent := HighestSent + 1;
```

```
392:           output T.TimeReq(HighestSent);
393:           output MT.MDTreq(PDUDT(HighestSent, Data));
394:           BuffSave(HighestSent, Data);
395:         end;
396:
397:     { Receive acknowledgement }
398:     from SENDING to same
399:       when MT.MAKind
400:         provided (PDU.Seq >= LowestUnacked)
                  and (PDU.Seq <= HighestSent)
                  and not Corrupted(PDU)
401:           var S: SeqType;
402:           begin
403:             for S := LowestUnacked to PDU.Seq do
404:               begin
405:                 output T.TimeCanc(S);
406:                 BuffFree(S);
407:               end;
408:             LowestUnacked := PDU.Seq + 1;
409:           end;
410:
411:         { Receive acknowledgement not in window }
412:         provided otherwise
413:           begin
414:             { Ignore }
415:           end;
416:
417:     { Timer response }
418:     from SENDING to same
419:       when T.TimeResp
420:         provided (Seq >= LowestUnacked) and (Seq <= HighestSent)
421:           var S: SeqType;
422:           begin
423:             for S := Seq to HighestSent do
424:               begin
425:                 output T.TimeCanc(S);
426:                 output MT.MDTreq(PDUDT(S, BuffRetrieve(S)));
427:                 output T.TimeReq(S);
428:               end;
429:             end;
430:
431:         provided otherwise
432:           begin
433:           end;
434:
435: end; { TransmitterBody }
436:
437:
438:
439:
440: module Receiver systemactivity;
441:   ip
442:     SR: RxUser(Receiver);
443:     MR: Rx(Receiver);
444: end;
445:
446:
447:
```

```
448:
449: { -------------------- Receiver module body ----------------- }
450:
451: body ReceiverBody for Receiver;
452:
453:   state RECEIVING;
454:
455:   (*******************)
456:   var
457:     PacketBuffer: array [0..RWSMax] of DTPDUType;
458:     BadPacket: DTPDUType;
459:   (*******************)
460:
461:   { Construct an AK PDU, given the sequence number }
462:   function PDUAK(S: SeqType): AKPDUType;
463:     var pdu: AKPDUType;
464:     begin
465:     (*******************)
466:     pdu.seq := s;
467:     PDUAK := pdu;
468:     (*******************)
469:     end;
470:
471:   { Retrieve PDU with sequence number S from buffer, returning a PDU
472:     with sequence number 0 if not in buf }
473:   function PDURetrieve(S: SeqType): DTPDUType;
474:     begin
475:     (*******************)
476:     if PacketBuffer[s mod RWSMax].Seq = s then
477:       PDURetrieve := PacketBuffer[s mod RWSMax]
478:     else
479:       PDURetrieve := BadPacket;
480:     (*******************)
481:     end;
482:
483:   { Save the PDU in the buffer }
484:   procedure PDUSave(PDU: DTPDUType);
485:     begin
486:     (*******************)
487:     PacketBuffer[PDU.seq mod RWSMax] := PDU;
488:     (*******************)
489:     end;
490:
491:   function Corrupted(PDU: DTPDUType): boolean;
492:     begin
493:     (*******************)
494:     {$C$ (void)PAR_pdu; /* avoid warning */}
495:     Corrupted := false;
496:     (*******************)
497:     end;
498:
499:   function UserData(p: DTPDUType): UserDataType;
500:     begin
501:     (*******************)
502:     UserData := p.msg;
503:     (*******************)
504:     end;
505:
```

```
506:   var
507:     NextRequired: SeqType;
508:     HighestReceived: SeqType;
509:     RWS: integer;
510:
511:   initialize
512:     to RECEIVING
513:       provided RWSMax > 0
514:         begin
515:           NextRequired := 1;
516:           HighestReceived := 0;
517:           RWS := RWSMax;
518:           all i: 0..RWSMax do
519:             PacketBuffer[i].Seq := 0;
520:           BadPacket.Seq := 0;
521:         end;
522:
523:   trans
524:
525:     { Receive Message in window }
526:     from RECEIVING to same
527:       when MR.MDTind
528:         provided (PDU.Seq >= NextRequired)
                  and (PDU.Seq < NextRequired + RWS) and not Corrupted(PDU)
529:           var
530:             S: SeqType;
531:             TPDU: DTPDUType;
532:             Done: boolean;
533:           begin
534:             PDUSave(PDU);
535:             S := NextRequired;
536:             Done := false;
537:             repeat
538:               TPDU := PDURetrieve(S);
539:               if TPDU.Seq = S then
540:                 begin
541:                   output SR.SDTind(UserData(TPDU));
542:                   S := S+1;
543:                 end
544:               else
545:                 Done := true;
546:             until Done;
547:             NextRequired := S;
548:             output MR.MAKreq(PDUAK(NextRequired-1));
549:           end;
550:
551:         { Receive Message that is not in window or corrupted }
552:         provided otherwise
553:           begin
554:             output MR.MAKreq(PDUAK(NextRequired-1));
555:           end;
556:
557:
558: end; { Receiver body }
559:
560:
561:
562: modvar
```

```
563:   TransmitterInstance: Transmitter;
564:   ReceiverInstance: Receiver;
565:   TransmitterUserInstance: TransmitterUser;
566:   ReceiverUserInstance: ReceiverUser;
567:   UMInstance: UM;
568:   TimerInstance: Timer;
569:
570:   initialize
571:     begin
572:       init TransmitterUserInstance with TransmitterUserBody;
573:       init ReceiverUserInstance with ReceiverUserBody;
574:       init TransmitterInstance with TransmitterBody;
575:       init ReceiverInstance with ReceiverBody;
576:       init UMInstance with UMBody;
577:       init TimerInstance with TimerBody;
578:
579:       connect TransmitterUserInstance.ST to TransmitterInstance.ST;
580:       connect ReceiverUserInstance.SR to ReceiverInstance.SR;
581:       connect TransmitterInstance.MT to UMInstance.MT;
582:       connect ReceiverInstance.MR to UMInstance.MR;
583:       connect TransmitterInstance.T to TimerInstance.T;
584:       (********************)
585:       connect TransmitterUserInstance.UU to ReceiverUserInstance.UU;
586:       (********************)
587:     end;
588: end. { Sliding Window Protocol }
589:
```

Anhang C: Formale Definition der Syntax von Open-Estelle

Im Folgenden ist die formale Definition der Sprachelemente von Open-Estelle („Language Elements of Open-Estelle“) aus [ThGo97c] wiedergegeben. Der Text ist als direkte Erweiterung zur formalen Sprachdefinition in [ISO97] ausgelegt und daher in englischer Sprache wiedergegeben.

Wir stellen der Definition von Open-Estelle den folgenden Auszug aus Abschnitt 5.2.1 („Nesting of modules“) von [ISO97] als Referenz voran:

There are the following five nesting principles for defining modules within modules:

(a) Every active module must be attributed; i.e., if for a given MODULE-HEADER-DEFINITION there is at least one MODULE-BODY-DEFINITION with its transition-part non-empty, then the MODULE-HEADER-DEFINITION must include one of the four keywords: `PROCESS`, `ACTIVITY`, `SYSTEMPROCESS` or `SYSTEMACTIVITY`.

(b) A system module cannot be nested (i.e., embodied) within an attributed module. As a consequence all modules embodying system modules must be inactive.

(c) Process and activity modules must be nested within a system module.

(d) A process or systemprocess module may be substructured only into process or activity modules; i.e., descendent (embodied) modules of a process or systemprocess module must be attributed with the keyword process or activity. Together with (b) and (e), this means that non-attributed modules are only those inactive modules embodying system modules, if such modules exist.

(e) An activity or systemactivity module may be substructured only into activity modules.

In addition to these static principles, it is important to stress the dynamic constraint that a module instance may be created and released or terminated exclusively by the instance whose MODULE-BODY-DEFINITION directly includes its definition (i.e., the parent module instance). The same is true as regards the creation and destruction of interaction point links.

Language Elements of Open-Estelle

This appendix defines the language elements of Open-Estelle based on and extending the Standard-Estelle language definition given in [ISO97]. Since Open-Estelle is a proper extension of Standard-Estelle, only new productions and extensions of existing productions are given. In the latter case, the productions and constraints given in this appendix take precedence over the Standard-Estelle productions.

Besides the start symbol SPECIFICATION of the Standard-Estelle grammar, the Open-Estelle grammar has two additional start symbols: INTERFACE-DEFINITION (Appendix C.1) and BEHAVIOUR-DEFINITION (Appendix C.2). Accordingly there are three distinct[1] types of textual units[2] containing a terminal string produced from one of the three start symbols: *Specification-textual-units* (in case of SPECIFICATION), *interface-textual-units* (in case of INTERFACE-DEFINITION) and *behaviour-textual-units* (in case of BEHAVIOUR-DEFINITION).

In Clauses 7.1.2.1, 7.1.2.2, 7.1.2.9, and 7.1.2.10 of [ISO97], the occurrences of „SPECIFICATION“ shall be replaced by „SPECIFICATION or INTERFACE-DEFINITION or BEHAVIOUR-DEFINITION“. In Clauses 7.1.2.1, 7.1.2.3, and 7.1.2.9 of [ISO97], the occurrences of „BODY-DEFINITION“ shall be replaced by „BODY-DEFINITION or INTERFACE-DECLARATION-PART or BEHAVIOUR-DECLARATION-PART“. In Clause 7.1.2 of [ISO97] all occurrences of „IDENTIFIER“ shall be replaced by „IDENTIFIER or QUALIFIED-IDENTIFIER“.

NOTE — All identifiers contained by a SPECIFICATION, INTERFACE-DEFINITION or BEHAVIOR-DEFINITION are
(1) required constants, types, procedures, or functions of Standard Estelle (see Clause 7.1.2.10 of [ISO97]) or
(2) have their defining-point inside the containing text (see Clause 7.1.2.3 of [ISO97]) or
(3) have their defining-point inside an *imported interface* (see Appendix C.3).

1. The first keyword of a textual unit („SPECIFICATION“, „INTERFACE“ or „BEHAVIOR“) uniquely identifies its type.
2. Since the Estelle standard does not state a representation for a specification text, we use the term „*textual unit*“ for self-contained syntactical objects (e.g. a specification).

C.1: Interface Definition

C.1.1: Syntax

```
INTERFACE-DEFINITION =          "INTERFACE" IDENTIFIER ";"
                                [ DEFAULT-OPTIONS ]
                                [ IMPORT-OPTIONS ]
                                INTERFACE-DECLARATION-PART
                                "END" "." .
INTERFACE-DECLARATION-PART =    { INTERFACE-DECLARATIONS } .
INTERFACE-DECLARATIONS =        CONSTANT-DEFINITION-PART
                                | TYPE-DEFINITION-PART
                                | CHANNEL-DEFINITION
                                | MODULE-HEADER-DEFINITION
                                | MODULE-BODY-DECLARATION .
MODULE-BODY-DECLARATION =       "BODY" IDENTIFIER "FOR" HEADER-IDENTIFIER ";"
                                "EXTERNAL" ";" .
```

C.1.2: Constraints

With the exception of module attribution rules[3] (Clause 7.3.6.2 of [ISO97]), all constraints and interpretation rules for SPECIFICATION (Clause 7.2 of [ISO97]) shall also be valid for INTERFACE-DEFINITION, the ones for DECLARATIONS (Clause 7.3 of [ISO97]) shall also be valid for INTERFACE-DECLARATIONs, and the ones for MODULE-BODY-DEFINITION (Clause 7.3.7 of [ISO97]) shall also be valid for MODULE-BODY-DECLARATION. Applying scope rules of Clause 7, Clause 8, and Annex C of [ISO97] the INTERFACE-DEFINITION shall be handled like a SPECIFICATION.

NOTE — Syntactically a MODULE-BODY-DECLARATION is a specialized MODULE-BODY-DEFINITION.

The IDENTIFIER of the INTERFACE-DEFINITION defines the *interface name*. It is a matter of the interpreting context to uniquely map any interface name (INTERFACE-IDENTIFIER) to a syntactically valid INTERFACE-DEFINITION that has the given name, or to the unique token "⊥". The interface name shall have no further significance within the INTERFACE-DEFINITION.

NOTE — A UNIX environment could use a file-naming convention and a search-path to implement this mapping.

3. Module attribution is handled in Appendix C.4.

C.1.3: Informal Semantics

An INTERFACE-DEFINITION is a container for the *declaration* of a set of open systems together with all necessary underlying definitions[4] as described above. It is one of the two additional start symbols of the Open-Estelle grammar and does not appear on the right-hand side of any production. An INTERFACE-DEFINITION is intended to be represented inside a separate textual unit (*interface-textual-unit*).

NOTE — An INTERFACE-DEFINITION can import other INTERFACE-DEFINITIONs and apply their definitions.

Every MODULE-BODY-DECLARATION contained in an INTERFACE-DEFINITION *declares* an open system. In doing so, it defines the external interface of the open system by referring to a MODULE-HEADER-DEFINITION, which describes a set of typed interaction-points and exported variables.

The *definition* of an open system is given in a BEHAVIOUR-DEFINITION inside a separate textual unit (see Appendix C.2). The pair of a BEHAVIOUR-DEFINITION and its referred INTERFACE-DEFINITION completely defines the declared set of open systems.

NOTE — An INTERFACE-DEFINITION may be referred by any number of BEHAVIOR-DEFINITIONs.

NOTE — Analogous to Standard Estelle the appearance of the keyword "**EXTERNAL**" inside a MODULE-BODY-DECLARATION of an INTERFACE-DEFINITION leads to an syntactically correct, but incomplete description (see clauses 7.3.7.3 and 9 of [ISO97]). But in opposition to a SPECIFICATION or a BEHAVIOR-DEFINITION, where only a textual modification can abolish this incompleteness, the attachment of an appropriate BEHAVIOR-DEFINITION (see Appendix C.2) leads to a complete description of the declared open systems.

NOTE — The definitions of constants, types, channels, and module-headers are independent of any BEHAVIOR-DEFINITION that refers the INTERFACE-DEFINITION. Therefore an INTERFACE-DEFINITION without any MODULE-BODY-DECLARATIONs is a complete description by itself.

C.1.4: Notes

Syntactically an INTERFACE-DEFINITION is very similar to a SPECIFICATION[5] with the following main modifications:

- The first token is the keyword "**INTERFACE**" instead of "**SPECIFICATION**".
- It has no SYSTEM-CLASS, INITIALIZATION-PART or TRANSITION-DECLARATION-PART. These are not necessary for an interface, since it is only a container for the description of the external interface of the open system (inside its INTERFACE-DECLARATION-PART) and not of its behaviour.
- It has no TIME-OPTIONS, since there is no TRANSITION-DECLARATION-PART and therefore there are no transitions with delay clauses.
- There are extended module attribution rules (see appendix C.4).
- There is an INTERFACE-DECLARATION-PART instead of a DECLARATION-PART.

---

4. An interface can gain access to the definitions and declarations given in other interfaces by *importing* these interfaces (see Appendix C.3).

5. The import-options have also been added to the SPECIFICATION production of Open-Estelle (see Appendix C.3).

An INTERFACE-DECLARATION-PART is a DECLARATION-PART with the following restrictions:

- It contains no INTERACTION-POINT-DECLARATION-PART, MODULE-VARIABLE-DECLARATION-PART, VARIABLE-DECLARATION-PART, STATE-DEFINITION-PART, STATE-SET-DEFINITION-PART or PROCEDURE-AND-FUNCTION-DECLARATION-PART, since these are not necessary for the definition of the interface.
- Every MODULE-BODY-DEFINITION is a MODULE-BODY-DECLARATION.
- Every MODULE-BODY-DECLARATION must be a process-module or an activity-module.

A MODULE-BODY-DECLARATION is a MODULE-BODY-DEFINITION that has no BODY-DEFINITION but contains the keyword "**EXTERNAL**". It only gives a declaration (not a definition) of an open system in form of an "**EXTERNAL**" module body.

In Standard-Estelle, this construct is used to give a syntactical correct specification that is left incomplete in a semantic sense — it has no formal semantics and cannot be completed without textual modification. In Open-Estelle, we extend its meaning so that every MODULE-BODY-DECLARATION inside of an interface-textual-unit declares an open system, which is defined inside a behaviour-textual-unit (see Appendix C.2).

C.2: Behaviour Definition

C.2.1: Syntax

```
BEHAVIOUR-DEFINITION =          "BEHAVIOUR" IDENTIFIER "FOR" INTERFACE-IDENTIFIER ";"
                                [ DEFAULT-OPTIONS ]
                                [ TIME-OPTIONS ]
                                [ IMPORT-OPTIONS ]
                                BEHAVIOUR-DECLARATION-PART
                                "END" "." .
BEHAVIOUR-DECLARATION-PART =    { BEHAVIOUR-DECLARATIONS } .
BEHAVIOUR-DECLARATIONS =        MODULE-BODY-DEFINITION .
INTERFACE-IDENTIFIER =          IDENTIFIER .
```

C.2.2: Constraints

With the exception of module attribution rules[6] (Clause 7.3.6.2 of [ISO97]), all constraints and interpretation rules for SPECIFICATION (Clause 7.2 of [ISO97]) shall also be valid for BEHAVIOUR-DEFINITION, and the ones for DECLARATIONS (Clause 7.3 of [ISO97]) shall also be valid for BEHAVIOUR-DECLARATIONS. Applying scope rules of Clause 7, Clause 8, and Annex C of [ISO97], the BEHAVIOUR-DEFINITION shall be handled like a SPECIFICATION.

The IDENTIFIER of the BEHAVIOUR-DEFINITION shall be the name of the BEHAVIOUR-DEFINITION, which shall have no significance within the BEHAVIOUR-DEFINITION.

The INTERFACE-IDENTIFIER of the BEHAVIOUR-DEFINITION shall be mapped by the interpreting context to a valid INTERFACE-DEFINITION with this name (see also Appendix C.1.2). This *referred* INTERFACE-DEFINITION is implicitly imported as if its name appeared in the IMPORT-OPTIONS (see Section C.3).

NOTE — The name of the BEHAVIOR-DEFINITION and the name of the INTERFACE-DEFINITION it refers to may be identical. This is especially useful if there is only one BEHAVIOR-DEFINITION for an INTERFACE-DEFINITION.

A BEHAVIOUR-DEFINITION has to directly contain exactly one *matching* MODULE-BODY-DEFINITION for each MODULE-BODY-DECLARATION within the referred INTERFACE-DEFINITION, i.e. they have the same name and refer to the same MODULE-HEADER-DEFINITION.

NOTE — For a given BEHAVIOR-DEFINITION this leads to a 1:1 relationship between the open system *definitions* in this BEHAVIOR-DEFINITION and the open system *declarations* in its referred INTERFACE-DEFINITION.

C.2.3: Informal Semantics

A BEHAVIOUR-DEFINITION is a container for the *definition* of a set of open systems. It is one of the two additional start symbols of the Open-Estelle grammar and does not appear on the right-hand side of any production. A BEHAVIOUR-DEFINITION is intended to be represented inside a separate textual unit (*behaviour-textual-unit*).

6. Module attribution is handled in Appendix C.4.

A BEHAVIOUR-DEFINITION refers to an INTERFACE-DEFINITION, which *declares* a set of open systems. For each of these *open system declarations*, the BEHAVIOUR-DEFINITION contains exactly one *matching* MODULE-BODY-DEFINITION, which *defines* the behaviour and internal structure of the previously declared open system.

NOTE — MODULE-BODY-DEFINITIONS inside a BEHAVIOR-DEFINITION may contain the keyword "**EXTERNAL**". Analogous to Standard Estelle such a BEHAVIOR-DEFINITION is syntactically correct but *incomplete* (see clauses 7.3.7.3 and 9 of [ISO97]) and therefore has no formal semantics.

Declaration and definition of the open system have the same name and refer to the same MODULE-HEADER-DEFINITION. Consequently the open system definition has the same external interface as the matching open system declaration. A MODULE-BODY-DEFINITION containing a BODY-DEFINITION supplements a behaviour to the open system, which formerly was only described in terms of its external syntactical interface.

It is possible to define different open system *definitions* for the same open system *declaration* by defining several BEHAVIOUR-DEFINITIONs (in separate textual units), each of them referring to the same INTERFACE-DEFINITION (which declares the open system). Since an *importing environment* (see Appendix C.3) only imports an INTERFACE-DEFINITION, the attachment of a matching BEHAVIOUR-DEFINITION to the importing environment is a matter of the interpreting context.

NOTE — This 1:n relationship between the declaration of an open system and its several (behavior) definitions allows some kind of *polymorphism* at the application of open systems.

NOTE — The formal representation of a completed system of open system definitions and importing environments could be a set of Open-Estelle textual units that is (1) closed in respect to imported of INTERFACE-DEFINITIONS and (2) contains for every (incomplete) INTERFACE-DEFINITIONS exactly one matching BEHAVIOR-DEFINITION.

NOTE — An Estelle Compiler under UNIX could use a script-file or command-line parameters to denote the BEHAVIOR-DEFINITIONS to be linked. By default the BEHAVIOR-DEFINITION with the same name as the imported INTERFACE-DEFINITION could be uniquely located[7] and linked (see Appendices C.1.2 and C.2.2).

C.2.4: Notes

In the previous section we introduced the INTERFACE-DEFINITION as a means for the formal description of the interface of a set of open systems. For the complete description of the open system we have to add the description of the *behaviour* of the open systems in a separated textual unit.

Since Open-Estelle represents an open system as a module-body, we just have to specify one BODY-DEFINITION for each BODY-DECLARATION given in the interface. As a container for the description of these BODY-DEFINITIONs Open-Estelle introduces the new non-terminal "BEHAVIOUR-DEFINITION". It is the second of the two additional start symbols of the Open-Estelle grammar and does not appear on the right-hand side of any production. An BEHAVIOUR-DEFINITION is intended to be used inside of a separated textual unit that we will refer to as *behaviour-textual-unit*.

Syntactically an BEHAVIOUR-DEFINITION is very similar to a SPECIFICATION[8] with the following restrictions and extensions:

7. A UNIX environment could use a file-naming convention and a search-path to implement this mapping.
8. The import-options have also been added to the specification production of Open-Estelle.

- It contains no CHANNEL-DEFINITION, MODULE-HEADER-DEFINITION, INTERACTION-POINT-DECLARATION-PART, MODULE-VARIABLE-DECLARATION-PART, VARIABLE-DECLARATION-PART, STATE-DEFINITION-PART, STATE-SET-DEFINITION-PART or PROCEDURE-AND-FUNCTION-DECLARATION-PART.
- It has no SYSTEM-CLASS, INITIALIZATION-PART or TRANSITION-DECLARATION-PART. These are not necessary for an BEHAVIOUR-DEFINITION, since it is only a container for the description of the behaviour of a set of open systems; this container itself does not have any behaviour.

C.3: Import of Interfaces

C.3.1: Syntax

```
IMPORT-OPTIONS =            "IMPORT" INTERFACE-IDENTIFIER { "," INTERFACE-IDENTIFIER } ";" .
BODY-DEFINITION =           [ IMPORT-OPTIONS ]
                            DECLARATION-PART
                            INITIALIZATION-PART
                            TRANSITION-DECLARATION-PART .
CONSTANT-IDENTIFIER =       IDENTIFIER | QUALIFIED-IDENTIFIER .
TYPE-IDENTIFIER =           IDENTIFIER | QUALIFIED-IDENTIFIER .
CHANNEL-IDENTIFIER =        IDENTIFIER | QUALIFIED-IDENTIFIER .
HEADER-IDENTIFIER =         IDENTIFIER | QUALIFIED-IDENTIFIER .
BODY-IDENTIFIER =           IDENTIFIER | QUALIFIED-IDENTIFIER .
QUALIFIED-IDENTIFIER =      INTERFACE-IDENTIFIER "::" IDENTIFIER .
INTERFACE-IDENTIFIER =      IDENTIFIER .
```

C.3.2: Constraints

Any INTERFACE-IDENTIFIER of the IMPORT-OPTIONS shall be mapped by the interpreting context to a valid INTERFACE-DEFINITION with this name (see also Appendix C.1.2). All of these INTERFACE-DEFINITIONs are *imported* into the closest containing SPECIFICATION, MODULE-BODY-DEFINITION, INTERFACE-DEFINITION or BEHAVIOUR-DEFINITION. (We will refer to these as the "*importing environments*").

The *import* of an INTERFACE-DEFINITION defines a *qualified visibility* in the importing environment for all IDENTIFIERs that have a defining-point whose region is the imported INTERFACE-DEFINITION.

To define a *qualified visibility* for an IDENTIFIER means that applying the scope rules of Clause 7, Clause 8, and Annex C of [ISO97] the QUALIFIED-IDENTIFIER for this IDENTIFIER is handled like an identifier that has a defining-point whose region is the importing environment. Applying compatibility rules (e.g. type- or assignment-compatibility), a reference to an imported QUALIFIED-IDENTIFIER refers to its native definition (inside a INTERFACE-DEFINITION).

NOTE — The import of items from INTERFACE-DEFINITIONs into importing environments defines the *imports-relation* over the set of valid Open-Estelle textual units.

NOTE — The *imports-relation* is not transitive: If INTERFACE-DEFINITION I_1 is the native defining point of IDENTIFIER n, INTERFACE-DEFINITION I_2 imports INTERFACE-DEFINITION I_1, and an different importing environment E imports I_2 but not I_1, then n is qualified visible in I_2, but not in E.

NOTE — A necessary condition for the syntactical correctness of an importing environment is the syntactical correctness of all imported INTERFACE-DEFINITIONs. The foundation of this requirement forbids any direct or indirect recursion of imports between INTERFACE-DEFINITIONs, i.e. the transitive closure of the *imports-relation* shall be irreflexive.

QUALIFIED-IDENTIFIERs are only valid for *applied occurrences* of imported identifiers (see Clause 7.1.2.8 of [ISO97]). For an IDENTIFIER that has a defining-point whose region is an INTERFACE-DEFINITION, the following shall be valid: (1) the INTERFACE-IDENTIFIER of the QUALIFIED-IDENTIFIER shall be the name of the INTERFACE-DEFINITION and (2) the IDENTIFIER of the QUALIFIED-IDENTIFIER shall be the native IDENTIFIER.

NOTE — The names of SPECIFICATIONs, INTERFACE-DEFINITIONs and BEHAVIOR-DEFINITIONs have no defining-point in the sense of clause 7, clause 8, and annex C of [ISO97].

NOTE — A QUALIFIED-IDENTIFIER *globally* identifies at most one unique item, because every INTERFACE-IDENTIFIER is uniquely mapped to at most one valid INTERFACE-DEFINITION (see Appendix C.1.2) and every IDENTIFIER that has a native defining-point whose region is the INTERFACE-DEFINITION, identifies a unique item inside the INTERFACE-DEFINITION (see Clause 7.1.2 of [ISO97]).

C.3.3: Informal Semantics

The import of an INTERFACE-DEFINITION gains the importing environment access to its definitions and declarations. This includes the MODULE-BODY-DECLARATIONs contained in the INTERFACE-DEFINITION, each of them declaring an open system. These can be handled like normal incompletely defined MODULE-BODY-DEFINITIONs in Standard-Estelle, i.e. several instances of them can be created with INIT-STATEMENTS and can be further handled like usual module instances.

In contrast to Standard-Estelle, an importing environment incorporating an imported open system declaration (which syntactically is an incompletely defined MODULE-BODY-DEFINITION) can be completed without any textual modifications, if for every imported INTERFACE-DEFINITION an appropriate BEHAVIOUR-DEFINITION is attached. In this case the instantiation of the open system (declared in an INTERFACE-DEFINITION) leads to the instantiation of the MODULE-BODY-DEFINITION given inside the appropriate BEHAVIOUR-DEFINITION.

NOTE — An open system definition *formally describes* an open system. The open system itself is an *instance* of this description (see also Clause 7.2.4 of [ISO97]). Consequently an importing environment may create several open systems of an open system definition by creating several module instances.

NOTE — It is possible to import the same INTERFACE-DEFINITION into several BODY-DEFINITIONS of an importing environment simultaneously. Consequently several instances of the same open system can be part of the module instance tree of an specification instance at independent positions.

C.3.4: Notes

The import of an INTERFACE-DEFINITION gains the importing environment (a SPECIFICATION, MODULE-BODY-DEFINITION, INTERFACE-DEFINITION or BEHAVIOUR-DEFINITION) access to the definitions and declarations (CONSTANT-DEFINITIONs, TYPE-DEFINITIONs, CHANNEL-DEFINITIONs, MODULE-HEADER-DEFINITIONs and MODULE-BODY-DECLARATIONs) of the imported INTERFACE-DEFINITION.

The import of an interface makes all items *qualified* visible to the importing environment whose defining-point is the appropriate INTERFACE-DEFINITION. This includes for example the CONSTANT-IDENTIFIERS of an ENUMERATED-TYPE definition which appear anywhere inside

of an INTERFACE-DEFINITION. It *does not* include the identifiers of items which are only indirectly imported (they are imported by an imported interface). This means the *imports-relation* is not transitive.

The scope of imported items is the same as if they had been defined (with their qualified name) at their importing environment, but their defining-point is still their unique INTERFACE-DEFINITION. This implies that a multiple import of the same interface out of different importing environments only gains *access* to the same definitions (which therefore are fully compatible[9]) instead of defining them several times. It is also allows to import the same interface into the same environment several times[10], since the imported items are still defined only once, namely just in their INTERFACE-DEFINITION.

In the previous section we introduced MODULE-BODY-DEFINITIONs directly contained in a BEHAVIOUR-DEFINITION as the syntactical representation of open systems. Now we will show how an open system can be applied by different types of Estelle environments. There will be no syntactical dependencies between the behaviour definition of an open system and any environment that applies it, since the interface of the open system is sufficient for the separated specification of both of them. This will allow the specification of an environment that applies one or more *abstract* open systems, i.e. open systems which are only described in terms of their external interface. Later on we will show how such environments can be composed with appropriate open systems.

The first step for the application of an open system is the *import* of the INTERFACE-DEFINITION which contains the MODULE-BODY-DECLARATION defining its external interface. The import of an INTERFACE-DEFINITION makes its definitions and declarations *visible* for the importing environment. So the importing environment also gains access to the MODULE-BODY-DECLARATIONs of the INTERFACE-DEFINITION, each of them declaring the external interface of an open system. The importing environment can handle these open system declarations like normal Standard-Estelle module bodies[11]: They may create various instances of the open systems by executing appropriate INIT-STATEMENTs, connect the external interaction points of these module instances, access their exported variables, terminate them, and so on.

The syntactical representation of the *import* of a set of interfaces is the new non-terminal "IMPORT-OPTIONS". We already used an optional IMPORT-OPTIONS at the definitions of INTERFACE-DEFINITION and BEHAVIOUR-DEFINITION (figure and). Since even every module body (including the specification) may import interfaces, we also introduce an optional IMPORT-OPTIONS into the productions of SPECIFICATION and MODULE-BODY-DEFINITION of Open-Estelle. We will refer to any of these four possible environments of an IMPORT-OPTIONS as an *importing environment*.

An IMPORT-OPTIONS consists of the new keyword "IMPORT", a comma separated list of INTERFACE-IDENTIFIERs, and a closing semicolon. For every INTERFACE-IDENTIFIER that occurs in this list the appropriate matching[12] interface is imported into the this environment.

9. The definitions given inside of an interface are interpreted in the context of the interface, independent of any importing environments.
10. Multiple Import
11. Section 7.2.4 defines the modified module nesting rules for Open-Estelle.
12. The unique attachment of an interface is implementation dependent, since the interface is located in a separated textual unit.

Imported items are only accessible by their *qualified name*. This is their simple IDENTIFIER (as given at their definition inside the interface) preceded by the INTERFACE-IDENTIFIER (of their defining interface) and a pair of colons. Qualified names can only be used to refer to imported items.

The INTERFACE-IDENTIFIERs are handled in a separated name-space, independently from the name-space of any other (qualified or unqualified) IDENTIFIERs, since their syntactical position inside the Open-Estelle grammar always allows to separate between INTERFACE-IDENTIFIERS and other IDENTIFIERs. Therefore there are no naming conflicts or name coverings between the names of (1) imported interfaces, (2) items imported from these interfaces and (3) locally defined items.

The reader should note that every (defined) qualified identifier globally identifies exactly one unique item, because every INTERFACE-IDENTIFIER has to identify an unique interface and every IDENTIFIER with its defining-point inside an INTERFACE-DEFINITION has to be unique for this interface. Therefore there are no naming interferences between different interfaces.

In the next section we will introduce an additional syntactical requirement at the application of imported open systems. This requirement modifies the module attribution rules of Standard-Estelle to have regard to the separation between the point of definition and the point of application of open systems in Open-Estelle.

C.4: Module Attribution Rules

The following module attribution and nesting rules shall replace the rules given in Clauses 5.2.1 and 7.3.6.2 of [ISO97]:

1. Each active module shall be attributed.
2. A module[13] that is not attributed or attributed "SYSTEMACTIVITY" or "SYSTEMPROCESS" shall be a SPECIFICATION or be directly contained in an INTERFACE-DEFINITION, a BEHAVIOUR-DEFINITION or a not attributed module.
3. A module that is attributed "**activity**" shall be directly contained inside an INTERFACE-DEFINITION, a BEHAVIOUR-DEFINITION or an attributed module.
4. A module that is attributed "**PROCESS**" shall be directly contained inside an INTERFACE-DEFINITION, a BEHAVIOUR-DEFINITION or a module that is attributed "**PROCESS**" or "**SYSTEMPROCESS**".
5. Within an importing environment that is a not attributed module, any imported MODULE-BODY-DECLARATION that is attributed "**PROCESS**" or "**ACTIVITY**" is handled like it was attributed "**SYSTEMPROCESS**" (in case of "**PROCESS**") or "**SYSTEMACTIVITY**" (in case of "**ACTIVITY**").
6. If the BODY-IDENTIFIER of an INIT-STATEMENT denotes an imported MODULE-BODY-DECLARATION, this MODULE-BODY-DECLARATION has to be attributed in such a manner that the module closest containing the INIT-STATEMENT could contain a child module with this attribution (according to the preceding attribution rules).

NOTE — All rules can be validated statically.

NOTE — This rules relax the restrictions to module attribution given in Clauses 5.2.1 and 7.3.6.2 of [ISO97]: They do not limit the possible attributions of MODULE-HEADER-DEFINITIONS but directly limit the possible attributions of nested modules and the possible instantiation of imported modules. The resulting restrictions to the attribution of the *static module nesting* (1-4) are identical to the ones in [ISO97].

NOTE — The module attribution rules (1-6) lead to the same restrictions to the attribution of the *dynamic module-instance tree* like the rules given in Clauses 5.2.1 and 7.3.6.2 of [ISO97].

NOTE — There are no restrictions to the attribution of MODULE-BODY-DECLARATIONs or MODULE-BODY-DEFINITIONs that are directly contained in an INTERFACE-DEFINITION or a BEHAVIOR-DEFINITION. Moreover there are no restrictions to the attribution of MODULE-HEADER-DEFINITIONs or MODULE-BODY-DECLARATIONs imported into an arbitrarily attributed module.

NOTE — Only the attribution of those imported MODULE-BODY-DECLARATIONs that are referred by an INIT-STATEMENT in the importing environment are restricted by module attribution rules. This allows the definition of interface-definitions that contain arbitrarily heterogeneous attributed MODULE-BODY-DECLARATIONs, which can be imported into any importing environment. The rules only restrict which of the imported MODULE-BODY-DECLARATIONs can be instantiated.

NOTE — Because of (5), an open system that is attributed "**activity**" can be imported and applicated (i.e. instantiated) by any (arbitrarily attributed) module.

13. With the term "module" we refer to a SPECIFICATION or a MODULE-BODY-DEFINITION.

Anhang D: Anwendung von Open-Estelle

D.1: Beispiel: Binary-Service

In diesem Beispiel wird die Spezifikation eines offenen Systems dargestellt (siehe Abschnitt 7.2).

Wir beginnen mit der Spezifikation der externen Schnittstelle (dem Interface) des Dienstes „binaryService“ (siehe Abb. 7-8 auf Seite 325). Diese wird typischerweise in einer Datei Namens „binaryService.sti“ (Estelle-Interface) abgelegt:

```
 1: INTERFACE binaryService;
 2:         TYPE    tOperand = RECORD x1, x2: REAL; END;
 3:                 tResult = REAL;
 4:         CHANNEL binaryServiceChannel (user, provider);
 5:                 BY user: request (x: tOperand);
 6:                 BY provider: respond (y: tResult);
 7:         MODULE binaryOperatorHeader ACTIVITY;
 8:                 IP toUser: binaryServiceChannel
                           (provider) COMMON QUEUE;
 9:                 END;
10:         BODY binaryOperator FOR binaryOperatorHeader;
11:                 EXTERNAL;
12: END.
```

Das Interface kann mit folgendem Kommando in die Objektdatei „binaryService.obi“ (Object-Interface) übersetzt werden:

```
pet -xopen binaryService.sti
  >> Portable Estelle Translator (PET) 2.01
  >> interface search path is "."
  >> translating input file binaryService.sti
  >> writing object file binaryService.obi
```

Die zugehörige Spezifikation der internen Aspekte (Behaviour) des oben deklarierten offenen Systems erfolgt nun durch die Behaviour-Definition namens „binaryAdder“ (siehe Abb. 7-10 auf Seite 327). Diese spezifiziert ein konkretes Verhalten und wird typischerweise in einer Datei namens „binaryAdder.stl“ (Estelle-Specification) abgelegt. Es ist zu beachten, dass zum obigen Interface durchaus mehrere verschiedene Behaviour-Definitionen angelegt werden können, die dann i. A. auch unterschiedliches Verhalten haben.

```
 1: BEHAVIOUR binaryAdder FOR binaryService;
 2:         BODY binaryOperator
                  FOR binaryService::binaryOperatorHeader;
 3:                 TRANS
```

```
4:                          WHEN toUser.request(x)
5:                          BEGIN
6:                                  OUTPUT toUser.respond(x.x1+x.x2);
7:                          END;
8:          END;
9: END.
```

Diese Spezifikations-Datei kann mit folgendem Kommando in die Objektdatei „binaryAdder.obj“ (Object-File) übersetzt werden:

```
pet -xopen binaryAdder.stl
  >> Portable Estelle Translator (PET) 2.01
  >> interface search path is "."
  >> translating input file binaryAdder.stl
  >> try "./binaryService.obi"
  >> import interface "binaryService" from "./binaryService.obi"
  >> writing object file binaryAdder.obj
```

Die oben erzeugte obi-Datei des Interfaces wurde dabei im Standard-Include-Pfad (implizites „-I .“) aufgelöst und geladen.

Definieren wir nun eine Spezifikation „test“, die das oben spezifizierte Interface importiert und das dort deklarierte offene System instanziiert. Sie wird typischerweise in einer Datei namens „test.stl“ (Estelle-Specification) abgelegt.

```
1: SPECIFICATION test;
2:          IMPORT binaryService;
3:          MODVAR mv: binaryService::binaryOperatorHeader;
4:          INITIALIZE
5:                  BEGIN
6:                          INIT mv WITH binaryService::binaryOperator;
7:                  END;
8:          { ... }
9: END.
```

Diese Spezifikations-Datei kann mit folgendem Kommando in die Objektdatei „test.obj“ (Object-File) übersetzt werden. Auch hier wird die oben erzeugte obi-Datei des Interfaces im Include-Pfad („-I .“) aufgelöst und geladen:

```
pet -xopen test.stl
  >> Portable Estelle Translator (PET) 2.01
  >> interface search path is "."
  >> translating input file test.stl
  >> try "./binaryService.obi"
  >> import interface "binaryService" from "./binaryService.obi"
  >> writing object file test.obj
```

Es ist zu beachten, dass dabei kein Zugriff auf die Behaviour-Definition oder davon abhängige Dateien erfolgt. Lediglich auf das gemeinsame Interface wird von Seiten der Behaviour-Definition und der importierenden Umgebung Bezug genommen. Entsprechend könnte es von beiden jeweils verschiedene Varianten geben, die erst zu einem späteren Zeitpunkt zu einem *Gesamtsystem* zugeordnet werden (siehe auch Abb. 7-16 auf Seite 343).

Die Zuordnung der Komponenten zu einem Gesamtsystem erfolgt erst durch die weiterführenden Werkzeuge zur Verarbeitung der Objektdateien. Wir demonstrieren dies im folgenden Anhang anhand der textuellen Verschmelzung der Systemteile zu einer geschlossenen Spezifikation und in Abschnitt 7.5 anhand der direkten Implementierung der Systemteile durch den Estelle-Compiler XEC.

D.2: Textuelle Verschmelzung mit petresolve

Oben haben wir aus der Interface-Definition, einer Behaviour-Definition und einem importierenden System die drei Objektdateien „`binaryService.obi`“, „`binaryAdder.obj`“ und „`test.obj`“ erzeugt. Nun *verschmelzen* wir die letzteren beiden Dateien mit Hilfe des Werkzeugs „`petresolve`“ zu einer geschlossenen Spezifikation (siehe Abschnitt 7.4):

```
petresolve binaryService=binaryAdder.obj test.obj > test.resolved.stl
  >> PET-resolve for PET 2.0 with Open-Estelle extension
  >> loading environment "test.obj"
  >> collecting "binaryService" from file "binaryAdder.obj"
  >> dumping unified specification
```

Wie bereits erwähnt, muss dabei auch die Zuordnung zwischen Behaviour-Definition und importierender Umgebung erfolgen, die jeweils zu dem Interface passen. Hier geschieht dies über den Kommandozeilenparameter „`binaryService=binaryAdder.obj`“.

Das Ergebnis der Verschmelzung ist eine korrekte Standard-Estelle-Spezifikation in der Datei „`test.resolved.stl`“, die keinerlei Bezüge mehr auf Open-Estelle enthält. Zu beachten ist hier u. a. auch die korrekte Umwandlung der Modulattributierung des Modulheaders `binaryOperatorHeader` von `ACTIVITY` nach `SYSTEMACTIVITY`, die aufgrund der Attributierung der importierenden Umgebung erforderlich war (siehe auch Abschnitt 7.2.4).[1]

```
 1: SPECIFICATION test ;
 2:
 3:
 4: TYPE
 5:         __OPEN13_binaryService_tOperand = record
 6:                x1: real;
 7:                x2: real;
 8:           end;
 9:
10:         __OPEN13_binaryService_tResult = real;
11:
12: channel __OPEN13_binaryService_binaryServiceChannel(user,provider);
13:   by user:
14:     request( x: __OPEN13_binaryService_tOperand);
15:   by provider:
16:     respond( y: __OPEN13_binaryService_tResult);
17:
18:
19:
20:
```

1. Die Einrückung des erzeugten Spezifikationstextes wurde zugunsten einer besseren Lesbarkeit nachbearbeitet.

```
21: MODULE __OPEN13_binaryService_binaryOperatorHeader SYSTEMACTIVITY;
22:  IP
23:     toUser: __OPEN13_binaryService_binaryServiceChannel(provider)
                    common queue;
24: END; { end of module __OPEN13_binaryService_binaryOperatorHeader }
25:
26:
27:

28: BODY __OPEN13_binaryService_binaryOperator
      for __OPEN13_binaryService_binaryOperatorHeader;
29:
30:
31: TRANS
32:      WHEN toUser.request
33: BEGIN
34:      output toUser.respond((x.x1 + x.x2));
35:
36: END; { end of transition block }
37:
38:
39: END; { end of body __OPEN13_binaryService_binaryOperator }
40:
41:
42:

43: MODVAR
44:         mv: __OPEN13_binaryService_binaryOperatorHeader;
45:
46:
47:
48: INITIALIZE
49: BEGIN
50:      init mv with __OPEN13_binaryService_binaryOperator;
51:
52: END; { end of transition block }
53:
54:
55: END. { end of specification test }
```

D.3: Direkte Implementierung offener Systeme mit XEC

Als Alternative zur Verschmelzung mit petresolve können die in Anhang D.1 gewonnenen Objektdateien „binaryAdder.obj“ und „test.obj“ auch direkt mit XEC getrennt voneinander implementiert werden (siehe Abschnitt 7.5). Wir stellen die Implementierung daher in zwei getrennten Unterabschnitten dar.

D.3.1: Implementierung des offenen Systems „binaryAdder“

Zunächst erfolgt die Generierung des C++-Codes (binaryAdder.cc und binaryAdder.h) sowie des Makefiles (binaryAdder.mk) durch xec:

```
xec binaryAdder.obj
  >> XEC (V 1.3.3) (built Sep  9 2004)
  >> reading "binaryAdder.obj"
  >> creating c++-files "./binaryAdder.cc" and "./binaryAdder.h"
  >> imported INTERFACE_binaryservice into BEHAVIOUR_binaryadder
  >> this is behaviour "binaryAdder" for interface "binaryService"
  >> creating makefile "./binaryAdder.mk" for goal "./binaryAdder"
  >> translation complete
```

Danach wird mittels des Makefiles die Übersetzung des C++-Codes gestartet. Anders als bei geschlossenen Systemen werden jedoch die Maschinen-Objektdateien hier noch nicht zu einem ausführbaren Programm gebunden, da dies aufgrund der Unvollständigkeit des Teilsystems nicht möglich wäre. Dazu wird das Make-Ziel „nolink“[2] verwendet:

```
make -f binaryAdder.mk nolink
  g++ -c -W -Wall  -g  \
        -DLOGGING  -DPROTOTYPES  -DINLINE_C -DPRETTYPRINT \
        -o binaryAdder.o binaryAdder.cc
```

Das Ergebnis der Übersetzung ist die Maschinen-Objektdatei binaryAdder.o.

D.3.2: Implementierung von der importierenden Umgebung „test“

Analog zur (aber unabhängig von der) Implementierung des offenen Systems kann die Implementierung der importierenden Umgebung erfolgen:

```
xec test.obj test
  >> XEC (V 1.3.3) (built Sep  9 2004)
  >> reading "test.obj"
  >> creating c++-files "./test.cc" and "./test.h"
```

2. Wenn nötig werden jedoch die XEC-Laufzeitmodule (hier in die Datei libxecrt_default.a) erzeugt.

```
  >> imported INTERFACE_binaryservice into SPEC_test
  >> creating makefile "./test.mk" for goal "./test"
  >> translation complete

make -f test.mk nolink
  g++ -c -W -Wall  -g  \
        -DLOGGING  -DPROTOTYPES  -DINLINE_C  -DPRETTYPRINT  \
        -o test.o test.cc
```

Das Ergebnis der Übersetzung ist die Maschinen-Objektdatei `test.o`.

D.3.3: Binden der Systemkomponenten

Zuletzt müssen die in Anhang D.3.1 und D.3.2 erzeugten Teilkomponenten zu einem ausführbaren Gesamtsystem zusammengebunden werden. Das Binden[3] erfolgt dabei vorteilhafterweise ebenfalls durch Aufruf von `g++`:

```
g++ test.o binaryAdder.o libxecrt_default.a -o testadder
```

Das resultierende Programm „testadder“ kann wie eine normale XEC-Implementierung genutzt werden (siehe auch Abschnitt 3.1.3).

3. gemeinhin auch als „Linken“ bezeichnet

Literaturverzeichnis

[AlSeUl85] Aho, A. V., Seth, R., Ullman, J. D.: Compilers. Principles, Techniques, and Tools. Addison-Wesley 1985.

[Bre97] Bredereke, J.: *Communication Systems Design with Estelle – On Style, Efficiency, and Analysis*, PhD thesis, Shaker Verlag, Aachen, Germany, ISBN 3-8265-2764-X, Aug. 1997

[BrGo94] Bredereke, J., Gotzhein, R.: *Increasing the Concurrency in Estelle*, in: R. L. Tenney, P. D. Amer, M. U. Uyar (Hrsg.), *Formal Description Techniques VI*, North-Holland, 1994

[Bud92] Budkowski, S.: Estelle Development Toolset, Computer Networks and ISDN Systems, Vol. 25, No.1, 1992

[Bud93] Budkowski, S. (Hrsg.): *Formal Specification, Validation and Performance Evaluation of the Xpress Transfer Protocol (XTP)*, Research Report No. 931004, Institute Nationale des Télécommunications, Evry, Frankreich, 1993

[CaBo96] Catrina, O., Borcoci, E.: *Estelle specification and validation of XTP 4.0*. Deliverable for Workpackage 2, Task 2.1, Copernicus Project COP62, 1996.

[Cat98] Catrina, O., Nogai, A.: "*On the Improvement of the Estelle Based Automatic Implementations*", Proceedings of Formal Description Techniques (XI) and Protocol Specification, Testing and Verification (XVIII) 1998 [FORTE / PSTV 1998], Kluwer Academic Publishers, Paris - France, pp 371-386

[ClTe90] Clark, D. D., Tennenhouse, D. L.: *Architectural Considerations for a New Generation Protocols*. Computer Communication Review, Vol. 20, No. 4, SIGCOMM '90, September 1990

[Dah96] Dahl, S.: *Performancevergleich von Protokollimplementierungen*, Diplomarbeit, Universität Kaiserslautern, Oktober 1996

[DeBu89] Dembinski, P., Budkowski, S.: *Specification Language Estelle*, in: M. Diaz et al. (eds.), *The Formal Description Technique, Estelle*, North-Holland, 1989, pp. 35-75

[DeKo92] Deitel, H. M., Kogan, M. S. : *The Design of OS/2*, Addison-Wesley, 1992

[DrPeDa94] Druschel, P., Peterson, L. L., Davie, B. S.: *Experiences with a high-speed network adaptor: A software perspective*. In Proceedings of the ACM SIGCOMM Conference, August 1994

[FaSch97] Fayad, M. E., Schmidt, D. C.: *Object-Oriented Application Frameworks*, Communication of the ACM, Volume 40, Number 10, Oct. 1997

[FiHo94] Fischer, S., Hofmann, B.: *An Estelle Compiler for Multiprocessor Platforms*, in Formal Description Techniques VI Proceedings of

FORTE'93, Boston, USA, October 1993, Elsevier Science Publishers B.V. North-Holland, Amsterdam, 1994.

[Fli+04] Fliege, I., Geraldy, A., Gotzhein, R., Schaible, P.: *A Flexible Micro Protocol Framework*, Proc. of 4rd SAM (SDL And MSC) Workshop, 2004, Ottawa, Canada

[Gam95] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns.* Elements of Reusable Objectoriented Software. 1995

[GBJ00] Grune, D.; Bal, H.E.; Jacobs, C.J.H.; Langendoen, K.G.: Modern Compiler Design. John Wiley & Sons, Ltd, 2000.

[GKS03] Gotzhein, R., Khendek, F., Schaible, P.: *Micro Protocol Design: The SNMP Case Study*, Telecommunications and beyond: The Broader Applicability of SDL and MSC, E. Sherratt (Ed.), LNCS 2599, pp. 61-73, Springer, 2003

[Got+96] Gotzhein, R., Bredereke, J., Effelsberg, W., Fischer, S., Held, T., König, H.: *Improving the Efficiency of Automated Protocol Implementation Using Estelle*, in Computer Communications, Vol. 19, No. 14, pages 1226-1235, 1996

[Got+01] Gotzhein, R., Peper, Ch., Schaible, P., Thees, J.: *Customization of Communication Systems*, 3rd International Conference on *Product Focused Software Process Improvement*, Kaiserslautern, Germany, September 10-13, 2001

[Got+03] Gotzhein, R. , Peper, Ch., Schaible, P., Thees, J.: *Durchgängige Entwicklung großer verteilter Systeme – Die SILICON-Fallstudie*, in: K. Irmscher, K. Fähnrich (Eds.): *Kommunikation in Verteilten Systemen* (KiVS), 13. Fachtagung Kommunikation in Verteilten Systemen. KiVS 2003, 25.-28. Februar 2003, Leipzig, Informatik Aktuell Springer 2003, ISBN 3-540-00365-7, pp.131-141

[GoRoTh96] Gotzhein, R., Rößler, F., Thees, J.: *Towards Open Estelle,* in: U. Herzog, H. Hermanns (Hrsg.): *Formale Beschreibungstechniken für verteilte Systeme*, Proceedings des 6. GI/ITG-Fachgesprächs, Erlangen, 20.-21.6.1996, pp. 89-98

[HeMiKö97] Henke, R., Mitschele-Thiel, A., König, H.: *On the Influence of Semantic Constraints on the Code Generation from Estelle Specifications*., Formal Description Techniques and Protocol Specification, Testing and Verification, FORTE X / PSTV XVII'97, 1997

[Hei+00] Heinkel, U., et.al.: *The VHDL Reference: A Practical Guide to Computer-Aided Integrated Circuit Design including VHDL-AMS*, John Wiley & Sons, Ltd, ISBN: 0-471-89972-0, April 2000

[ISO81] ISO/TC 97/SC 16, ISO 7498, „*Data Processing – Open Systems Interconnection – Basic Reference Model*“, 1981

[ISO88a] ISO/IEC 8824:1988, CCITT X.208 „*Specification of Abstract Syntax Notation One (ASN.1)*“

[ISO88b] ISO/IEC 8825-1, CCITT X.209-1 „*Part 1: Basic Encoding Rules (BER)*“

[ISO89] ISO/TC 97/SC 21, ISO 9074, „*Information Processing Systems – Open Systems Interconnection – Estelle: A Formal Description Technique Based on an Extended State Transition Model*“, 1989

[ISO97] ISO/TC 97/SC 21, ISO 9074:1997, „*Information Processing Systems – Open Systems Interconnection – Estelle: A Formal Description Technique Based on an Extended State Transition Model*“, 1997

[ISO98] ISO/IEC 14822-1998, „*Information Technologie – Programming Languages – C++*“, 1998

[ISO00] ISO/IEC 8824-1:2002, ITU-T Rec. X.680 (2002) „*Abstract Syntax Notation One (ASN.1)*“

[ITU94] ITU-T, Recommendation Z.100 (03/93), „*CCITT Specification and Description Language (SDL)*“, 1994

[ITU00] ITU-T, Recommendation Z.100 (11/99), „*Specification and description language (SDL)“*, 2000

[KrGo93] Kreuz, D., Gotzhein R.: *A Compiler for the Parallel Execution of Estelle Specifications*, in: H. König (Hrsg.), *Formale Methoden für verteilte Systeme*, Fokus-Band 8, Saur-Verlag, München, 1993

[LaFi95] Lallet, E., Fischer, S., Francois Verdier, J.: *A New Approach for Distributing Estelle Specifications*, in Formal Description Techniques VIII (Proc. of FORTE'95, Montreal, Canada, 1995), G. v. Bochmann and R. Dssouli and O. Rafiq, pages 439-448, Chapman & Hall, London, 1995

[LaLeMa91] Lallet, E., Lebrun, Ch. A., Martin, J.-F., „Un outils de génération automatique de l'environnement d'exécution de specification Estelle“, Acte du 'Colloque Francophone surl'Ingénierie des Protocoles (CFIP'91)', Pau, 17-19 Sept. 1991

[May95] Mayer, O.: *Programmieren in COMMON LISP*, Spektrum Akademischer Verlag, Januar 1995

[PaDrZw00] Pai, V. S. , Druschel P., Zwaenepoel, W.: *IOLite: A unified I/O buffering and caching system*. ACM Transactions on Computer Systems (TOCS), 18, February 2000

[Pen96] Penner, H.: *Erstellung eines Software-Monitors zur Analyse automatisch generierter Protokollimplementierungen*, Diplomarbeit, Universität Kaiserslautern, Januar 1996

[Pet62] Petri, C. A.: *Kommunikation mit Automaten*, Dissertation, Technische Hochschule Darmstadt 1962

[RFC1379] Braden, R., „*Request for Comments 1379: Extending TCP for Transactions -- Concepts*“, RFC1379, Network Working Group, November 1992, http://www.rfc-editor.org

[RFC1644] Braden, R., „*Request for Comments 1379: T/TCP -- TCP Extensions for Transactions -- Functional Specification*“, RFC1644, Network Working Group, July 1994, http://www.rfc-editor.org

[RFC1700] Reynolds, J., Postel, J. (Edts.), „*Request for Comments 1700: Assigned Numbers*“, STD 2, RFC1700, Network Working Group, October 1994, http://www.rfc-editor.org

[RFC1883] Deering, S., Hinden, R. (Edts.), „*Request for Comments 1883: Internet Protocol, Version 6 (IPv6), Specification*“, RFC1883, Network Working Group, December 1995, http://www.rfc-editor.org

[RFC2460] Deering, S., Hinden, R. (Edts.), „*Request for Comments 2460: Internet Protocol, Version 6 (IPv6), Specification*“, RFC2460, Network Working Group, December 1998, http://www.rfc-editor.org

[RFC768] Postel, J. (Edt.), „*Request for Comments 768: User Datagram Protocol, Specification*“, RFC768, Information Sciences Institute, Marina del Rey, California, August 1980, http://www.rfc-editor.org

[RFC791] Postel, J. (Edt.), „*Request for Comments 791: Internet Protocol, Specification*“, RFC791, Information Sciences Institute, Marina del Rey, California, September 1981, http://www.rfc-editor.org

[RFC793] Postel, J. (Edt.), „*Request for Comments 793: Transmission Control Protocol, Specification*“, RFC793, Information Sciences Institute, Marina del Rey, California, September 1981, http://www.rfc-editor.org

[RiCl89] Richard, J. L., Claes, T.: *A Generator for C-Code for Estelle*, in: M. Diaz et al (eds.), *The Formal Description Technique Estelle*, North-Holland, 1989, pp. 397-420

[ScTh01] Schaible, P., Thees, J.: *Maßschneiderung vs. Wiederverwendung bei Kommunikationssystemen (Fallstudie SILICON)*, 11. ITG-Fachgespräch *Formale Beschreibungstechniken für verteilte Systeme*, Bruchsal, 21.-22.6.2001

[SiSt90] Sijelmassi, R., Strausser, B.: *NIST Integrated Tool Set for Estelle*, in: Quemada, J., Manos, J., Vazquez, E.: *Third International Conference on Formal Description Techniques (FORTE'90)*, Madrid, Spanien, 1990

[SiSt93] Sijelmassi, R., Strausser, B.: *The PET and DINGO tools for deriving distributed implementations from Estelle*, Computer Networks and ISDN Systems, Vol. 25, No. 7, 1993, pp. 1115-1130

[SNL95a] Sandia National Laborities: *SandiaXTP Reference Manual*, Release 1.4, Livermore, California, USA, September 1995

[SNL95b] Sandia National Laborities: *SandiaXTP — An Object-Oriented Implementation of XTP 4.0 Derived from the Meta-Transport Library: User Manual*, Release 1.4, Livermore, California, USA, October 1995

[Svo89] Svobodova, L.: *Implementing OSI Systems*. IEEE Journal on Selected Areas of Communication, vol. 7, no. 7, 1989, pp. 1115-1130.

[ThBr95] Thees, J., Bredereke, J.: *Ein Werkzeug zur Analyse von Feature-Interaktionen in IN*, in: R. Gotzhein und J. Bredereke (Hrsg.): *Formale Beschreibungstechniken für verteilte Systeme*, Proceedings des 5. GI/ITG-Fachgesprächs, Kaiserslautern, 22.- 23.6.1995, 199-208

[The95] Thees, J.: *Entwurf und Implementierung eines Werkzeugs zur Analyse von Feature-Interaktionen in Estelle-Spezifikationen,* Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern, April 1995

[The98] Thees, J.: *Protocol Implementation with Estelle – from Prototypes to Efficient Implementations,* in: S. Budkowski, S. Fischer, R. Gotzhein: *Proc. of the 1st International Workshop of the Formal Description Technique Estelle* (ESTELLE'98), Evry, France, Nov. 1998

[The99] Thees, J.: *Implementierungs- und Optimierungsmodelle des Expermental Estelle Compilers,* in: K. Spies, B. Schätz (Hrsg.): *Formale Beschreibungstechniken für verteilte Systeme* (FBT´99), Herbert Utz Verlag Wissenschaft, ISBN 3-89675-918-3, München, Juni 1999

[The02] Thees, J.: *The Estelle Compiler 'XEC'*, Posterpresentation, 6th CaberNet Radicals Workshop, 24-27 February 2002, Madeira Island

[The03] Thees, J.: *Type Abstraction in Formal Protocol Specifications with Container Types*, in: H. König, M. Heiner, A. Wolisz (Eds.): *Formal Techniques for Networked and Distributed Systems* – FORTE 2003, 23rd IFIP WG 6.1 International Conference, Berlin, Germany, September 29 - October 2, 2003, Proceedings. Lecture Notes in Computer Science 2767 Springer 2003, ISBN 3-540-20175-0, pp. 383-398

[ThGo97a] Thees, J., Gotzhein, R.: *Leistungsbewertung automatisch generierter Protokollimplementierungen mit Estelle – eine Bestandsaufnahme*, Interner Bericht 290/97, Fachbereich Informatik, Universität Kaiserslautern, 1997

[ThGo97b] Thees, J., Gotzhein, R.: *Leistungsbewertung automatisch generierter Protokollimplementierungen*, in: K. Irmscher, Ch. Mittasch, K. Richter (Hrsg.): *Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen*, Kurzbeiträge und Toolbeschreibungen zur 9. ITG/GI-Fachtagung MMB'97, Informatik in Freiberg, Band 4, TU Bergakademie Freiberg, September 1997

[ThGo97c] Thees, J., Gotzhein, R.: *A Formal Syntax and a Formal Semantics for Open Estelle*, Internal Report No. 292/97, Dept. of Comp. Sci., University of Kaiserslautern, 1997

[ThGo97d] Thees, J., Gotzhein, R.: *Generation of Efficient Protocol Implementations – an Experimental Code Generator for Estelle and its Application to XTP,* 6th Open Workshop on High Speed Networks, Stuttgart, October 8-9, 1997

[ThGo97e] Thees, J., Gotzhein, R.: *Open Estelle – A Formal Description Technique for Open Distributed Systems*, International Workshop on Dynamic Modeling of Information Systems, Yamagata, Japan, November 1997

[ThGo98a] Thees, J., Gotzhein, R.: *The eXperimental Estelle Compiler – Automatic Generation of Implementations from Formal Specifications,* in: M. Ardis (Edt.), *Proceedings of The 2nd Workshop on Formal Methods in Software Practice* (FMSP'98), Clearwater Beach, Florida, USA, March 1998

[ThGo98b] Thees, J., Gotzhein, R.: *Open Estelle – An FDT for Open Distributed Systems,* in: S. Budkowski, A. R. Cavalli, E. Najm (Eds.): *Formal*

Description Techniques and Protocol Specification, Testing and Verification, FORTE XI / PSTV XVIII'98, IFIP TC6 WG6.1, 3-6 November, 1998, Paris, France. IFIP Conference Proceedings 135 Kluwer 1998, ISBN 0-412-84760-4, pp. 19-36

[Turn93] Turner, K. J. (edt.): *Using Formal Description Techniques - An Introduction to Estelle, LOTOS and SDL*, John Wiley and Sons Ltd., 1993, ISBN 0-471-93455-0

[Wen98] Wenz, M.: Design und Implementierung einer portablen grafischen Oberfläche zur Visualisierung und Steuerung des Ablaufs von Estelle-Implementierungen, Diplomarbeit, FB Informatik, Univ. Kaiserslautern, 1998

[Wik04] Wikipedia; http://de.wikipedia.org/; Stand Oktober 2004

[Wir85] Wirth, N.: Programming in Modula-2, Springer Verlag 1982, 1983, 1985, ISBN 0-540-15078-1

[XTP95] XTP Forum, *Xpress Transport Protocol Specification*, XTP Rev. 4.0, XTP Forum, Santa Barbara, USA, 1995

Lebenslauf

Persönliche Angaben

Name:	Joachim Thees
Anschrift:	Fuchsberg 13, 67663 Kaiserslautern
Geburtsdatum/-ort:	18. Mai 1968 in Morbach
Staatsangehörigkeit:	deutsch
Familienstand:	ledig

Ausbildung

1974 – 1987	Grund- und Realschule, Gymnasium *Abschluss: Allgemeine Hochschulreife, Note: 1,8*
Okt. 1987 – Dez. 1988	Wehrdienst (Fernmelde-Übertragungstechniker)
WS 1988/89 – SS 1995	Studium der Informatik (Nebenfach Elektrotechnik) an der Universität Kaiserslautern *Abschluss: Diplom, „Mit Auszeichnung bestanden“*
17. Februar 2005	Promotion an der Technischen Universität Kaiserslautern *Abschluss: Dr. rer. nat., „Mit Auszeichnung bestanden“*

Beruf

Okt. 1995 – Juni 2002	Tätigkeit als Wissenschaftlicher Mitarbeiter in der Arbeitsgruppe „*Rechnernetze*“ des FB Informatik der Universität Kaiserslautern.
seit Juli 2002	Tätigkeit als Leiter des „*Service-Center Informatik*“ des FB Informatik der Technischen Universität Kaiserslautern.