Hasso-Plattner-Institut für Softwaresystemtechnik an der Universität Potsdam
Fachgebiet Softwaretechnik und Qualitätsmanagement

Strukturorientierte Optimierung der Qualitätseigenschaften von softwareintensiven technischen Systemen im Architekturentwurf

Dissertation
zur Erlangung des akademischen Grades
"doctor rerum naturalium"
(Dr. rer. nat.)
in der Wissenschaftsdisziplin Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

von
Dipl. Ing. (BA), Dipl. Inf. Lars Grunske

Potsdam, den 16.02.04

Erstgutachter	Prof. Dr.-Ing. Peter Liggesmeyer
Zweitgutachter	Prof. Dr.-Ing. Helmut Balzert
	Prof. Dr.-Ing. Stefan Jähnichen

Kurzfassung
Gegenstand dieser Arbeit ist die Entwicklung eines Verfahrens zur Verbesserung der Qualitätseigenschaften eines softwareintensiven technischen Systems. Dieses Verfahren soll im Speziellen auf die Architekturspezifikation angewendet werden und die Qualitätseigenschaften Sicherheit, Verfügbarkeit, Zuverlässigkeit, Wartbarkeit und Echtzeitfähigkeit verbessern.
Grundlegend für das erstellte Verfahren ist ein zyklischer Prozess, welcher in der Architekturentwurfsphase angewendet wird. In diesem Prozess werden zunächst die Qualitätseigenschaften anhand einer Architekturspezifikation ermittelt. Erfüllt eine Architekturspezifikation die Qualitätsanforderung nicht, so wird die Strukturspezifikation dieser Architektur transformiert. Die verwendeten Transformationen sollten die Qualitätseigenschaften positiv beeinflussen, ohne dabei aber das funktionale Verhalten des Systems zu verändern. Dadurch lassen sich gezielt die Qualitätseigenschaften verbessern.
Der Hauptbeitrag dieser Arbeit liegt in der Entwicklung einer geeigneten Architekturbeschreibungssprache für softwareintensive technische Systeme und eines Formalismus für die automatische Anwendung von Architekturtransformationen auf Basis dieser Architekturbeschreibungssprache. Als formale Grundlage der Architekturbeschreibungssprache werden hierarchische typisierte Hypergraphen verwendet. Dabei wird jedes Architekturelement auf ein Element eines hierarchischen typisierten Hypergraphen abgebildet. Als Konsequenz lassen sich die Architekturtransformationen als Graphtransformationsregeln in der Kategorie der hierarchischen typisierten Hypergraphen darstellen und automatisch anwenden. Die verwendeten Graphtransformationsregeln werden um Anwendungsbedingungen erweitert, wodurch eine automatische Auswahl der Architekturtransformationen ermöglicht wird. Darüber hinaus wird ein Konzept zur Spezifikation von struktur- und typgenerischen Graphtransformationsregeln eingeführt, welches die Mächtigkeit der Architekturtransformationen erhöht.
Für die Bestimmung der Qualitätseigenschaften einer Architekturspezifikation und somit zur Bestimmung des Erfolgs einer Architekturtransformation wird ein Formalismus vorgestellt, welcher nicht-dekomponierte Architekturelemente mit modularen Evaluationsmodellen annotiert. Beispiele für die verwendeten Evaluationsmodelle sind Fehlerbäume, Markov-Modelle und Scheduling-Modelle. Ausgehend von diesen Evaluationsmodellen sind die Qualitätseigenschaften des gesamten zu erstellenden Systems auf Basis der Architekturspezifikation mit kompositionsbasierten Evaluationstechniken bestimmbar.
Die praktische Umsetzbarkeit des vorgestellten Verfahrens wird mit der Durchführung von industriellen Fallstudien aus dem Bereich des Personennahverkehrs, der Reaktorsicherheit und der Luft- und Raumfahrt demonstriert. Dazu wird das im Rahmen der Dissertation entwickelte Werkzeug BALANCE genutzt. Dieses Werkzeug bietet die Möglichkeit, Architekturspezifikationen für softwareintensive technischen Systeme zu erstellen, zu evaluieren und zu transformieren.

Danksagungen
Ich möchte mich hiermit zunächst allgemein bei allen bedanken, die mich bei der Erstellung der Dissertation unterstützt haben.
Besonderer Dank gilt dabei meinem Betreuer Prof. Dr.-Ing. Peter Liggesmeyer. Er gab mir die Freiheit, aber auch alle notwendige Unterstützung, meinen eigenen wissenschaftlichen Weg zu finden. Ebenso möchte ich mich bei Prof. Dr.-Ing. Stefan Jähnichen und Prof. Dr.-Ing. Helmut Balzert für die Gutachtertätigkeit und die fachlichen Hinweise bedanken.
Neben meinen Betreuern gilt mein Dank meinen Kollegen, welche mir mit vielen fachlichen Gesprächen und wertvollen Diskussionen halfen, die Ideen dieser Arbeit zu formulieren und umzusetzen. Dabei möchte ich mich besonders bei Roland Neumann bedanken, der mir in vielen Fachgesprächen half, meine Gedanken zu sortieren. Für weitere fachliche Inspirationen möchte ich mich bei den Mitgliedern der Pattern- und der Softwarearchitektur-Community bedanken. Besonderer Dank gilt dabei Richard P. Gabriel, Neil Harrison, Nenad Medvidovic, Jan Bosch und Linda Rising.
Bei der Realisierung der Dissertation haben mich die Studenten Einar Lück, Denis Wundke, Nicolas Heess, Isabel Peuker, Martin Herbort, Andreas Leidner, Tobias Weinert, Dominic Wist, Dirk Zander, Florian Wonneberg und Matthias Senf unterstützt. Von diesen möchte ich besonders Einar Lück für seine Hilfe bei der Implementierung und Realisierung der Beispiele und Dennis Wundke für die Hilfe bei der Spezifikation des COOL Schemas danken. Aber auch den Mitgliedern des BALANCE-Teams gilt mein Dank, da durch sie erst die Validation meines Ansatzes und die Durchführung der Fallstudien ermöglicht wurden.
Für das Korrekturlesen bedanke ich mich bei Dr.-Ing. Dirk Riehle, Dr.-Ing. Ralf H. Reussner, Dr.-Ing. Peter Tabeling und Dipl.-Ing. Bernhard Gröne.
Die letzte Danksagung gilt meiner Familie für die private Unterstützung und meiner Liebe Franziska Helm, die stets zu mir gehalten hat und mir das Gefühl gibt, etwas Besonderes zu sein.

1 Einleitung

Die vorliegende Dissertation beschäftigt sich mit den Qualitätsaspekten von softwareintensiven technischen Systemen. Diese softwareintensiven technischen Systeme beeinflussen im zunehmenden Maße unser tägliches Leben. So werden sie zum Beispiel zur Steuerung von Systemabläufen und Prozessen im Automotive-, Avionik-, Schienenverkehrs-, Kraftwerks- und Telekommunikationssektor sowie in der Medizintechnik verwendet. Die Relevanz dieser Systeme ist daher offensichtlich.
Die hier betrachteten softwareintensiven technischen Systeme sind grundsätzlich aus Softwarebestandteilen, einer Hardwareplattform, auf welcher die Software ausgeführt wird, und mindestens einem Sensor und einem Aktor aufgebaut. Über die Sensoren werden Informationen über den aktuellen Zustand des gesteuerten Prozesses und der Systemumwelt ermittelt. Ausgehend von diesen Informationen und der Verhaltensspezifikation der Software werden die Systemausgaben bestimmt, die über die Aktoren die Systemumgebung beeinflussen. Die Sensoren und Aktoren bilden folglich die Schnittstelle zur Umwelt des Systems. Die Systemgrenze eines softwareintensiven technischen Systems wird daher zwischen dem zu steuernden Prozess und den Sensoren/Aktoren gezogen.
Zusammenfassend wird der Aufbau eines softwareintensiven technischen Systems schematisch in Abbildung 1-1 dargestellt.

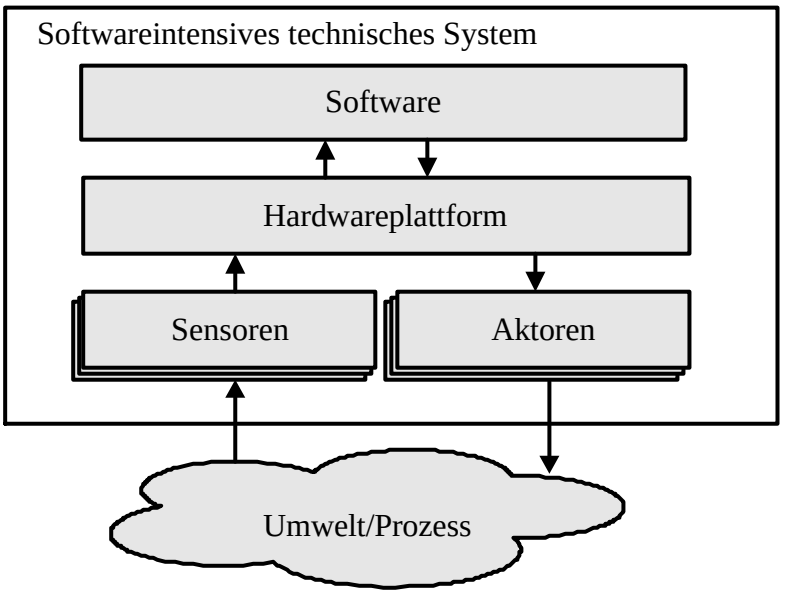


Abbildung 1-1 Konzeptueller Aufbau eines softwareintensiven technischen Systems

1.1 Problembeschreibung

Aufgrund ihrer Einsatzgebiete, z. B. in sicherheitskritischen Bereichen, werden an softwareintensive technische Systeme hohe Qualitätsansprüche gestellt. Diese Qualitätsansprüche beziehen sich nicht nur auf die Erfüllung der funktionalen Anforderungen, sondern auch auf die Erfüllung von Qualitätsanforderungen, wie zum Beispiel Anforderungen an die Sicherheit, die Verfügbarkeit, die Zuverlässigkeit, die Wartbarkeit und die Echtzeitfähigkeit eines Systems. Daraus ergeben sich für die Entwicklung von softwareintensiven technischen Systemen die im Folgenden charakterisierten Probleme, welche im Fokus dieser Dissertation stehen:

- Die Erfüllung der Qualitätsanforderungen ist mit einem erhöhten ökonomischen Aufwand verbunden /Liggesmeyer 00/. Ursache hierfür sind die erhöhten Kosten für qualifiziertes Personal und für die konstruktive und analytische Qualitätssicherung.

- Die Erfüllung der Qualitätsanforderungen muss vor dem praktischen Einsatz eines softwareintensiven technischen Systems nachgewiesen werden. Beispielsweise muss für ein System in einem sicherheitskritischen Anwendungsgebiet ein Sicherheitsnachweis erstellt werden /EN 50129/, /SAE ARP 4761/, welcher die Erfüllung der Sicherheitsanforderungen bestätigt.
- Qualitätsanforderungen stehen oft zueinander in Konflikt /Lundberg et al. 99/, /Bosch 00/, /Chung et al. 99/. Daher kann die Verbesserung einer Qualitätseigenschaft zur Verschlechterung einer anderen Qualitätseigenschaft führen.

Zusammenfassend lässt sich deshalb das der Dissertation zugrunde liegende Problem wie folgt definieren:

Softwareintensive technische Systeme besitzen hohe Qualitätsanforderungen, deren Erfüllung während des Entwicklungsprozesses nachgewiesen werden muss und welche oft in Konflikt zu ökonomischen Interessen und anderen Qualitätsanforderungen stehen.

1.2 Zieldefinition

Zur Lösung der beschriebenen Probleme bei der Entwicklung von softwareintensiven technischen Systemen soll ein Verfahren entwickelt werden, das zunächst prüft, ob die Qualitätseigenschaften des Systems den Anforderungen entsprechen. Dabei sollen speziell die Qualitätsmerkmale Sicherheit, Verfügbarkeit, Zuverlässigkeit, Wartbarkeit und Echtzeitfähigkeit betrachtet werden, da diese für softwareintensive technische Systeme von besonderer Bedeutung sind /EN 50126/, /Liggesmeyer 00/. Eine Erweiterung des Verfahrens auf weitere Qualitätsmerkmale soll jedoch möglich sein.
Werden die Anforderungen nicht erfüllt, so müssen konstruktive qualitätssichernde Maßnahmen zur Verbesserung der Qualitätseigenschaften ergriffen werden. Aus ökonomischen Gründen sollten diese möglichst frühzeitig im Systementwicklungsprozess erfolgen. Als geeignetste Phase im Systementwicklungsprozess wird die Architekturentwurfsphase gesehen /Bosch, Molin 99/, /Grunske 03c/, da auf Basis der Architekturspezifikation die frühestmögliche Bestimmung der zu erwartenden Qualitätseigenschaften möglich ist /Clements et al. 01/, /Kazeman et al. 99/.
Zusammenfassend ergibt sich das nachfolgend formulierte Ziel der Dissertation:

Ziel der Dissertation ist es, ein Verfahren zu entwickeln, das die Qualitätseigenschaften eines softwareintensiven technischen Systems in der Architekturentwurfsphase bestimmt, verbessert und optimiert.

1.3 Begriffsdefinitionen

Bei der Problembeschreibung und der Zieldefinition der Dissertation wurden bereits einige fachspezifische Begriffe genutzt, welche im Kontext der softwareintensiven technischen Systeme eine spezielle Bedeutung besitzen. Neben diesen werden in der Arbeit noch weitere Begriffe benötigt, welche unterschiedliche Interpretationen zulassen. Aus diesem Grund werden im Folgenden zur Vermeidung von Missverständnisse die in dieser Arbeit verwendeten Begriffe definiert. Das daraus resultierende Begriffsfundament wird in der Arbeit konsequent angewendet, um dem Leser das Verständnis der genutzten Fachterminologie zu erleichtern.

Qualitätsmerkmal, Qualitätsanforderung und Qualitätseigenschaft
Grundlegend für die Definition der Begriffe Qualitätsanforderung und Qualitätseigenschaft ist der Begriff des Qualitätsmerkmals. Dieser wird nach /Liggesmeyer 00/ wie folgt definiert:

> ***Qualitätsmerkmal:*** *Ein Qualitätsmerkmal ist die Eigenschaft einer Funktionseinheit, anhand derer ihre Qualität beschrieben und beurteilt wird, die jedoch keine Aussage über den Grad der Ausprägung enthält.*

Ausgehend von dieser Definition lassen sich die Begriffe Qualitätsanforderung und Qualitätseigenschaft nach /Balzert 98/ wie folgt definieren:

> ***Qualitätsanforderung:*** *Eine Qualitätsanforderung legt fest, welche Qualitätsmerkmale im konkreten Fall als relevant erachtet werden und in welcher Qualitätsstufe sie erreicht werden sollen.*

Qualitätseigenschaft: *Eine Qualitätseigenschaft beschreibt die Ausprägung des Qualitätsmerkmals eines Systems durch quantifizierte Informationen.*

Demnach spezifizieren die Qualitätsanforderungen den Kundenwunsch nach einem geforderten Qualitätsmerkmal und dessen Ausprägung. Die Qualitätseigenschaften beschreiben die tatsächliche Ausprägung der Qualitätsmerkmale im realisierten System. Die Übereinstimmung der Qualitätseigenschaften mit den Qualitätsanforderungen bestimmt somit die Qualität des Systems.
Zusammenfassend ist dieses Begriffsfundament für die Qualität in der folgenden Abbildung dargestellt:

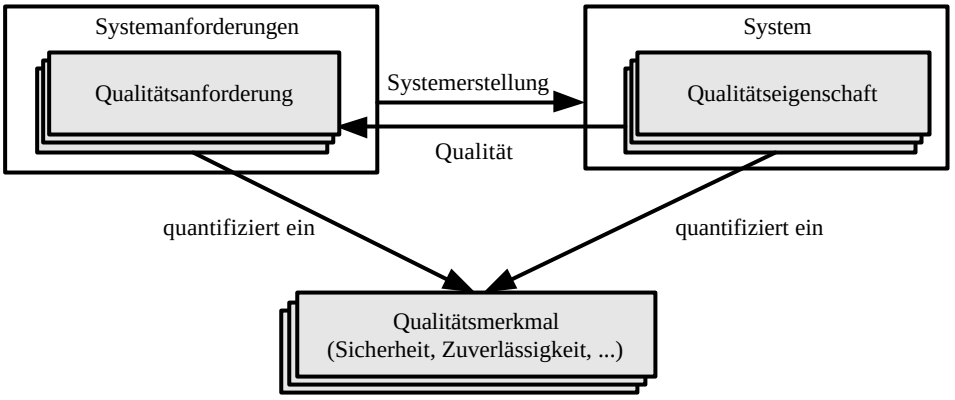


Abbildung 1-2 Qualitätsanforderung, Qualitätseigenschaft und Qualitätsmerkmal

Qualitätssicherung, Konstruktive Qualitätssicherung, Analytische Qualitätssicherung
Wie in /Liggesmeyer 00/ und /Balzert 98/ beschrieben, umfasst die Qualitätssicherung konstruktive und analytische Maßnahmen zur Bestimmung und Verbesserung der Qualität während der Entwicklung eines Systems. In der vorliegenden Arbeit wird der Begriff der Qualitätssicherung daher folgendermaßen verwendet:

Qualitätssicherung: *Gesamtheit von angemessenen, aufeinander abgestimmten Maßnahmen zur Erfüllung vorgegebener Anforderungen an die Qualität eines Systems.*

Konstruktive Maßnahmen zielen nach /Balzert 98/ darauf ab, den Entwicklungsprozess und das erstellte Produkt so zu gestalten, dass die Qualität des Systems von Anfang an (a priori) gewährleistet wird. Im Gegensatz dazu prüfen analytische Qualitätssicherungsmaßnahmen die bereits fertiggestellten End- oder Zwischenprodukte im Entwicklungsprozess.
Daher werden die Aufgaben der konstruktiven und der analytischen Qualitätssicherungsmaßnahmen wie folgt definiert:

Konstruktive Qualitätssicherung: *Die konstruktiven Qualitätssicherungsmaßnahmen gewährleisten, dass das entstehende Produkt bzw. der Erstellungsprozess a priori bestimmte Eigenschaften besitzen.*

Analytische Qualitätssicherung: *Die analytischen Qualitätssicherungsmaßnahmen messen die existierenden Qualitätseigenschaften und identifizieren Ausmaß und Ort der Defekte.*

Fehlverhalten, Fehler und Irrtum
Im Fokus der Dissertation stehen die Qualitätsmerkmale Sicherheit, Verfügbarkeit, Zuverlässigkeit, Wartbarkeit und Echtzeitfähigkeit. Für die Begriffsdefinition dieser Merkmale werden die Begriffe Fehlverhalten, Fehler und Irrtum benötigt, welche im Folgenden definiert werden.
Ein Fehlverhalten (*failure*) beschreibt nach /Liggesmeyer 00/ die Verletzung einer Qualitätsanforderung bei der Benutzung eines Produkts. Ursache eines Fehlverhaltens ist ein Fehler *(fault, defect)*. Fehler lassen sich nach /DIN VDE 0801/ und /EN 50129/ in zufällige und systematische Fehler unterteilen. Zufällige Fehler treten in softwareintensiven technischen Systemen ausschließlich in Hardwarekomponenten auf. Der Eintrittszeitpunkt eines Fehlverhaltens auf Basis eines zufälligen Fehlers ist nicht bestimmbar /DIN VDE 0801/.
Systematische Fehler entstehen durch menschliches Versagen bei der Erstellung und während des Betriebs eines Systems /EN 50129/. Beispiele für systematische Fehler von Softwarekomponenten sind Spezifikationsfehler, Implementierungsfehler, Entwurfsfehler und Programmierfehler. Beispiele für systematische

Fehler von Hardwarekomponenten sind Spezifikationsfehler, Fertigungsfehler und die falsche Bauteilauswahl und -dimensionierung.
Systematische Fehler sind prinzipiell vermeidbar, wenn auch in der Praxis eine vollständige Vermeidung nur schwer oder nur mit sehr hohem Aufwand zu erreichen ist.
Ursache eines Fehlers ist ein Irrtum *(error)* /Liggesmeyer 00/. Irrtümer werden bei der Erstellung, Bedienung und Wartung des Systems begangen und resultieren aus einem falschen Verständnis von der Auswirkung einer Handlung.
Für die Begriffe Irrtum, Fehler und Fehlverhalten werden die folgenden Definitionen verwendet:

> ***Fehlverhalten:*** *Ein Fehlverhalten ist ein nicht gewünschtes Verhalten des Systems. Es zeigt sich dynamisch bei der Benutzung des Systems. Ursache eines Fehlverhaltens ist ein Fehler.*
>
> ***Fehler:*** *Ein Fehler ist statisch im System präsent und beschreibt die Nichterfüllung einer Anforderung an das System.*
>
> ***Irrtum:*** *Ein Irrtum ist die Ursache eines Fehlers und resultiert aus einem falschen Verständnis über die Auswirkung einer Handlung bei der Systemerstellung.*

Zuverlässigkeit
Für den Begriff des Qualitätsmerkmals Zuverlässigkeit existieren mehrere Definitionen in Normen /EN 50129/, /IEC 61508/ und in der Fachliteratur /Birolini 99/, /Liggesmeyer 00/. Diese Definitionen besitzen einen einheitlichen Grundkonsens, welcher sich wie folgt zusammenfassen lässt:

> ***Zuverlässigkeit:*** *Die Zuverlässigkeit ist ein Maß für die Fähigkeit einer Betrachtungseinheit, eine geforderte Funktion unter gegebenen Bedingungen für eine gegebene Zeitdauer zu erbringen.*

Ausgehend von dieser Definition wird die Zuverlässigkeit durch die Auftrittshäufigkeit von Fehlverhalten geprägt.

Wartbarkeit
Die Wartung wird in dieser Arbeit als eine Aktivität zur Beseitigung von Fehlern nach der Inbetriebnahme eines Systems betrachtet. Die Wartbarkeit eines Systems beschreibt demnach die Zeit und den Aufwand, der mit der Wartung verbunden ist. Allgemeiner definiert wird die Wartbarkeit in der /IEEE 610.12/:

> ***Wartbarkeit:*** *Die Wartbarkeit ist ein Maß für die Leichtigkeit, mit der ein System geändert werden kann, um Fehler zu beheben, seine Fähigkeiten zu erhöhen oder es an eine veränderte Umgebung anzupassen.*

Verfügbarkeit
Der Begriff der Verfügbarkeit wird in dieser Arbeit in Anlehnung an /Liggesmeyer 00/ wie folgt verwendet:

> ***Verfügbarkeit:*** *Verfügbarkeit ist ein Maß für die Fähigkeit einer Betrachtungseinheit, zu einem gegebenen Zeitpunkt funktionstüchtig zu sein.*

Berechnet wird die Verfügbarkeit eines Systems aus dem Verhältnis der Zeit, in der das System die funktionalen Anforderungen erfüllt, zur gesamten Laufzeit des Systems. Demnach besitzen sowohl die Zuverlässigkeit und die Warbarkeit einen Einfluss auf die Verfügbarkeit eines Systems.

Sicherheit
In dieser Arbeit wird der Begriff Sicherheit im Sinne des englischen Begriffes *safety* verwendet und nach der /DIN VDE 0801/ wie folgt definiert:

> ***Sicherheit:*** *Sicherheit ist eine Sachlage, bei der das Risiko nicht größer als das Grenzrisiko ist.*

Basis dieser Definitionen ist das vom System ausgehende Risiko sowie das festgelegte Grenzrisiko, welches bestimmt, ab wann ein Risiko nicht mehr akzeptabel ist /EN 50129/. Nach /Leveson 95/ besteht ein Risiko aus den folgenden zwei Komponenten:

- die Wahrscheinlichkeit, mit der ein Ereignis oder die Kombination von Ereignissen eintritt, die zu einem Gefahrenfall führt bzw. die Häufigkeit solcher Ereignisse
- die Folgen/Konsequenzen eines Gefahrenfalls.

In den Normen /EN 50129/ und /DIN V VDE 0801/ wird der Risikobegriff wie folgt definiert:

> ***Risiko:*** *Das Risiko beschreibt die Kombination aus der Häufigkeit oder der Wahrscheinlichkeit und den Folgen eines spezifizierten gefährlichen Ereignisses /EN 50129/.*

> ***Risiko:*** *Das Risiko, das mit einem bestimmten technischen Vorgang oder Zustand verbunden ist, wird zusammenfassend durch eine Wahrscheinlichkeitsaussage beschrieben, welche die zu erwartende Häufigkeit des Eintritts eines zum Schaden führenden Ereignisses und das beim Ereigniseintritt zu erwartende Schadensausmaß berücksichtigt. /DIN V VDE 0801/.*

Relevant für diese Definitionen sind die spezifizierten gefährlichen Ereignisse, welche vom System ausgehen und dessen Umwelt gefährden. Diese gefährlichen Ereignisse sind zumeist auf ein Fehlverhalten des Systems zurückzuführen. Nach der Definition kann das Risiko jeweils nur für ein spezifiziertes gefährliches Ereignis betrachtet werden. Daher kann die Sicherheit eines Systems nur in Bezug auf dieses spezifizierte gefährliche Ereignis bestimmt werden. Zur vollständigen Bestimmung der Systemsicherheit ist aus diesem Grund die Kombination aus allen gefährlichen Ereignissen zu betrachten.

Echtzeitfähigkeit

Die Echtzeitfähigkeit des Systems ist ein grundlegendes Qualitätsmerkmal eines softwareintensiven technischen Systems. Dieses Qualitätsmerkmal wird oft auch als Echtzeitfähigkeit bezeichnet und in Anlehnung an /Meyer 97/ und /Hüsener 94/ wie folgt definiert:

> ***Echtzeitfähigkeit:*** *Die Echtzeitfähigkeit ist die Fähigkeit eines Systems, auf Eingaben aus der Systemumwelt innerhalb von definierten Zeitschranken zu reagieren.*

Systemerstellungsprozesse und Systemerstellung

In dieser Arbeit wird die Systementwicklung in Anlehnung an /Balzert 01/, /Gomaa 01/, /Hofmeister et al. 99/ und /Booch et. al. 99/ in die nachstehenden Phasen unterteilt:

- Anforderungsanalysephase
- Architekturentwurfsphase
- Modulimplementierungsphase
- Modultestphase
- Integrationstestphase
- Systemtestphase
- Betriebs- und Wartungsphase

In der Anforderungsanalysephase wird die Anforderungsspezifikation erstellt, die das extern beobachtbare Verhalten des Systems beschreibt. In der anschließenden Architekturentwurfsphase wird das System in geeignete Teilsysteme dekomponiert, deren extern beobachtbares Verhalten ebenfalls spezifiziert wird. Das Resultat der Architekturentwurfsphase ist eine Systemarchitekturspezifikation, welche die Basisstruktur des Systems beschreibt. Die Spezifikation der Interna der einzelnen Teilsysteme wird in der in der Modulimplementierungsphase vorgenommen. Ergebnis ist eine Modulspezifikation für jedes Teilsystem. Die Modulspezifikationen und die Systemarchitekturspezifikation ergeben zusammen die Systemspezifikation, die das Verhalten des Systems vollständig beschreibt. Die anschließenden Testphasen dienen der analytischen Qualitätssicherung. Dabei werden in der Modultestphase die Modulspezifikationen geprüft. In der Integrationstestphase wird inkrementell überprüft, ob die einzelnen Modulspezifikationen korrekt zusammenarbeiten und in der Systemtestphase wird geprüft, ob das vollständige System seiner Anforderungsspezifikation entspricht. Das Ergebnis der Systemtestphasen ist ein Produkt, das dem Kunden übergeben wird. Dieser Kunde benutzt das System in der Betriebsphase. Werden in dieser Phase schwerwiegende Fehlverhalten erkannt, so geht das System in die Wartungsphase über. Ziel dieser Phase ist es, die dem Fehlverhalten zugrunde liegenden Fehler zu beseitigen.

Die Anordnung und die Interaktion der einzelnen Phasen sowie die konkreten Aktivitäten zur Erstellung und Qualitätssicherung des Systems werden in einem Systemerstellungsprozess definiert. Für die konkrete Spezifikation des Systemerstellungsprozesses werden Prozessmodelle verwendet, welche sich z. B. in die folgenden Kategorien einordnen lassen:

- Phasenmodelle (z. B. Wasserfallmodell /Boehm 81/, V-Modell /V-Modell 97/)
- Inkrementelle, prototypische, evolutionäre Modelle und Spiralmodelle /Boehm 88/
- Objektorientierte Modelle /Jacobson et al. 99/, /Meyer 97/

Die in dieser Arbeit vorgestellten Verfahren sind weitgehend unabhängig von einem Prozessmodell und lassen sich daher bei allen genannten Prozessmodellen einsetzen.

1.4 Aufbau der Dissertation

Die vorliegende Dissertation enthält acht Kapitel. Nach dieser Einleitung wird in Kapitel 2 der Stand der Technik bei der Spezifikation von Qualitätsanforderungen vorgestellt. Des Weiteren wird ein Überblick über die konstruktiven und analytischen Qualitätssicherungstechniken in der Anforderungsanalysephase gegeben. Anschließend wird in Kapitel 3 gezeigt, wie im Architekturentwurf die Qualitätseigenschaften einzelner Architekturelemente spezifiziert werden. Ausgehend davon wird dargestellt, wie eine Evaluation der Qualitätseigenschaften für die gesamte Architektur erfolgt. Zusammenfassend beschreiben somit die Kapitel 2 und 3 ausgewählte Aspekte des Stands der Technik zur Behandlung von Qualitätsmerkmalen in der Anforderungsanalyse- und Architekturentwurfsphase.
Ausgehend von den dargestellten Konzepten und Spezifikationsformalismen wird in Kapitel 4 ein Verfahren zur Verbesserung der Qualitätseigenschaften eines Systems im Architekturentwurf vorgestellt. Dabei wird im Besonderen ein Automatisierungsansatz vorgestellt, der eine werkzeugunterstützte Optimierung der Qualitätseigenschaften ermöglicht.
Für die technische Realisierung des Verfahrens und des Automatisierungsansatzes wird in Kapitel 5 mit der Architekturbeschreibungssprache COOL eine geeignete Spezifikationssprache für die Architektur eines softwareintensiven technischen Systems vorgestellt. Die Idee der Architekturbeschreibungssprache ist, Hypergraphen als einheitliches Konzept zur Spezifikation einer Architektur zu verwenden. Somit lassen sich Architekturtransformationen wie in Kapitel 6 dargestellt als Hypergraphtransformationsregeln spezifizieren und automatisch auf Architekturen in der Architekturbeschreibungssprache anwenden.
Kapitel 7 befasst sich mit der praktischen Realisierbarkeit und validiert die Anwendbarkeit des erstellten Verfahrens. Dazu wird das Werkzeug BALANCE vorgestellt, welches eine Spezifikation, Evaluation und Transformation von Architekturen in der Architekturbeschreibungssprache COOL ermöglicht. Aufbauend darauf wird die praktische Umsetzbarkeit mit industriellen Fallstudien aus dem Bereich des Personennahverkehrs, der Reaktorsicherheit und der Avionik nachgewiesen.
Abgeschlossen wird die Dissertation mit einer Zusammenfassung, welche die zentralen Ergebnisse sowie den praktischen Nutzen der Dissertation beschreibt. Des Weiteren enthält dieses Kapitel einen Ausblick auf weiterführende Forschungsthemen, die sich im Rahmen der Dissertation ergeben haben.
Im Anhang der Arbeit werden die verwendeten mathematischen Grundlagen definiert. Außerdem wird eine genaue Spezifikation der erstellten Algorithmen sowie eine Beschreibung des COOL-XML-Schemas und der verwendeten Transformationsoperatoren gegeben.

2 Qualitätsanforderungen

Im Folgenden wird der Stand der Technik der Spezifikation von Qualitätsanforderungen in der Anforderungsanalysephase betrachtet. Dazu wird zunächst ein idealisiertes Vorgehensmodell der Anforderungsanalysephase vorgestellt. Aufbauend auf diesem Vorgehensmodell wird beschrieben, wie sich funktionale Anforderungen spezifizieren lassen. Anschließend wird gezeigt, wie Qualitätsanforderungen beschrieben werden und es wird ein Überblick über den Stand der Technik der analytischen und konstruktiven Qualitätssicherungstechniken innerhalb der Anforderungsanalysephase gegeben.

2.1 Idealisiertes Vorgehensmodell der Anforderungsanalysephase

In dieser Arbeit wird für die Erstellung der Anforderungsspezifikation eines softwareintensiven technischen Systems das in Abbildung 2-1 dargestellte Vorgehensmodell bevorzugt. Dieses Vorgehensmodell wurde in Anlehnung an /Nuseibeh, Easterbrook 00/, /Robertson, Robertson 99/, /Lamsweerde 00/ und /Balzert 01/ erstellt und beschreibt den Entwicklungsprozess in der Anforderungsanalysephase unter idealisierten Bedingungen.

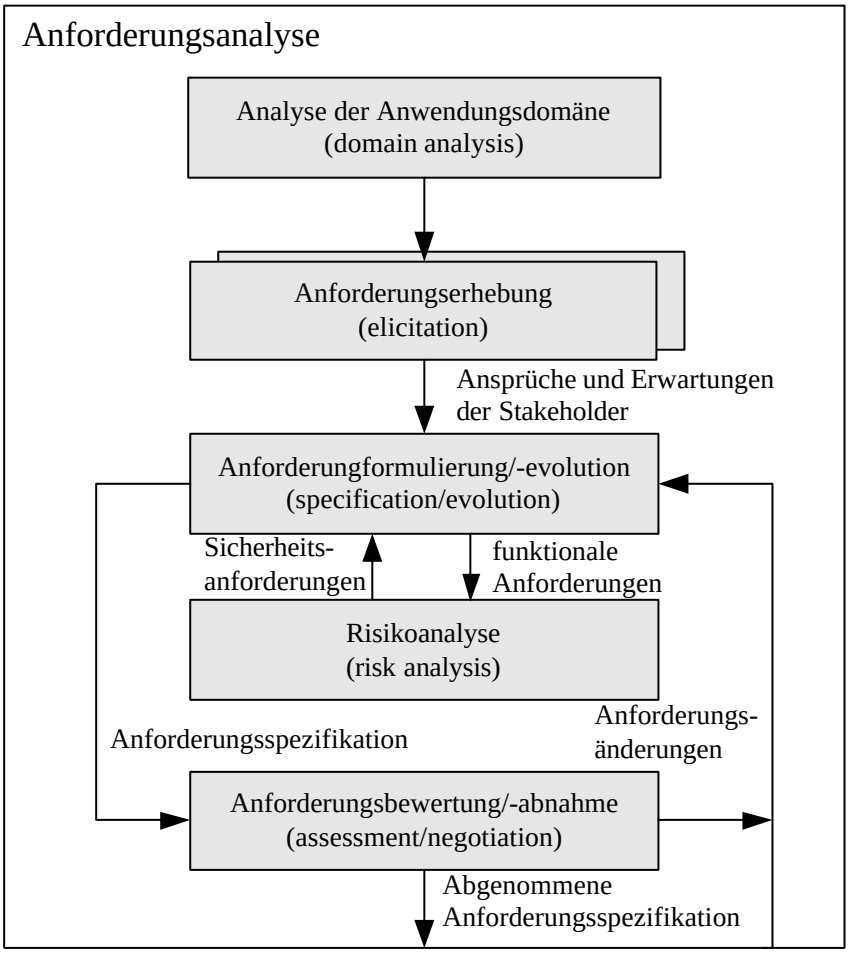


Abbildung 2-1 Vorgehensmodell der Anforderungs- und Risikoanalyse

Zur besseren Verständlichkeit des Vorgehensmodells werden im Folgenden die einzelnen Aktivitäten genauer beschrieben.

Analyse der Anwendungsdomäne
Zu Beginn der Anforderungsanalysephase sollte die Anwendungsdomäne betrachtet werden, in die das zu entwickelnde System integriert wird. Dabei werden die Akteure identifiziert, die mit dem System interagieren /Lamsweerde 00/. Beispiel für solche Akteure sind das Bedienpersonal, das Wartungspersonal sowie weitere externe technische Systeme. Das zu entwickelnde System selbst wird ebenfalls als Akteur angesehen. Die Analyse der Anwendungsdomäne ist außerdem zur Sammlung von Fachwissen und zur Identifikation von sozio-ökonomischen und technischen Randbedingungen und Normen notwendig. Diese Randbedingungen und Normen sind speziell für die Entwicklung von softwareintensiven technischen Systemen von Bedeutung /Braband 01/, da sie den Entwicklungsprozess beeinflussen.

Anforderungserhebung
In der Anforderungserhebungsphase werden die Ansprüche und Erwartungen aller am System interessierten und in der Entwicklung involvierten Personen ermittelt /Cristel, Kang 92/. Diese Personen sind beispielsweise die Endnutzer, die Auftraggeber, die Entwickler oder das Wartungspersonal /Rupp 01/, /IEEE Std 1471/. Für die Identifizierung der Ansprüche und Erwartungen der am Projekt beteiligten Personen gibt es eine Reihe von Vorgehensweisen /Balzert 98/. Als geeignete Technik gelten Interviews und Fragebögen, aber auch Untersuchungen von Referenzprodukten /Nuseibeh, Easterbrook 00/. Das Ergebnis der Anforderungserhebungsphase ist eine formlos zusammengestellte Liste mit den Ansprüchen und Erwartungen.

Anforderungsformulierung und Anforderungsevolution
Aus der Liste der Ansprüche und der Erwartungen an das System werden in der Anforderungsformulierungsphase die konkreten Anforderungen erstellt. Diese werden in einer geeigneten Notation formuliert. Das Ergebnis der Anforderungsformulierungsphase ist ein Artefakt, welches alle Anforderungen an das System übersichtlich und strukturiert präsentiert und somit die Grundlage für alle weiteren Projektphasen bildet.
Während der Entwicklung und des Betriebs eines Systems sind die Anforderungen selten stabil /Broy 97/, d.h. sie müssen gegebenenfalls geändert oder ergänzt werden. Diese Änderung von Anforderungen während der Systementwicklung wird als Anforderungsevolution bezeichnet.

Anforderungsbewertung und Anforderungsabnahme
Die Anforderungsbewertungsphase dient der analytischen Qualitätssicherung der Anforderungsspezifikation. Verwendet werden dazu unter anderem formale Analysen sowie Reviews.
In der Anforderungsabnahmephase wird die Anforderungsspezifikation den Kunden vorgelegt. Sie entscheiden, ob die Anforderungsspezifikation ihren Ansprüchen und Erwartungen entspricht. Ergebnis der Anforderungsabnahmephase ist eine vom Kunden akzeptierte und gegengezeichnete Anforderungsspezifikation. Sie ist somit ein verbindliches Dokument für die weitere Entwicklung des Systems.

Risikoanalyse
Zusätzlich zur Erfassung und Definition der Anforderungen des Kunden ist es bei vielen softwareintensiven technischen Systemen notwendig, die Sicherheitsanforderungen zu ermitteln /Braband 01/. Dies erfolgt durch die Risikoanalyse, welche vom Betreiber oder Auftraggeber eines technischen Systems durchgeführt wird /EN 50126/. Ziel der Risikoanalyse ist die Identifikation sowie die Festlegung der systemrelevanten Gefährdungen und deren tolerierbaren Gefährdungsraten. Nach /Leveson 95/ bilden sie zusammen die Sicherheitsanforderungen an das zu erstellende System und werden als zusätzliche Anforderungen in die Anforderungsspezifikation integriert. Zur Durchführung einer Risikoanalyse eignet sich das in Abbildung 2-2 dargestellte Vorgehensmodell /Braband, Lennartz 99/.

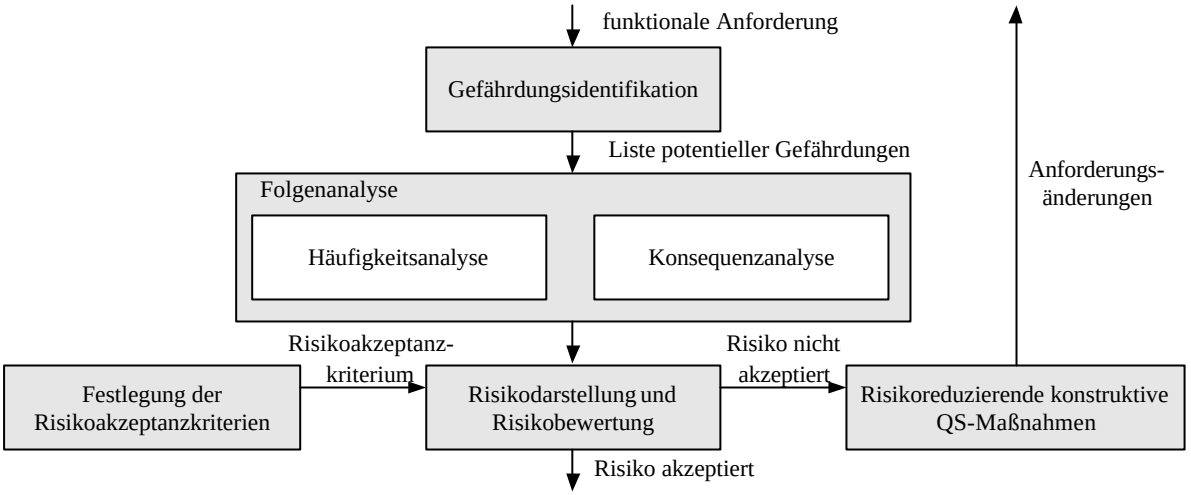


Abbildung 2-2 Vorgehensmodell der Risikoanalyse

Die dargestellten Aktivitäten des Vorgehensmodells werden im Folgenden kurz charakterisiert. Begonnen wird mit der Gefährdungsidentifikation. Diese identifiziert die potenziell vom System ausgehenden Gefährdungen auf Basis der funktionalen Anforderungsspezifikation und gliedert sich nach /Braband, Lennartz 99,00/ in eine empirische und eine kreative Phase.

In der empirischen Phase wird dabei auf das Expertenwissen aus der Entwicklung früherer Systeme zurückgegriffen. Dabei wird geprüft, ob die bei diesem System identifizierten oder aufgetretenen Gefährdungen für das neue System ebenfalls relevant sind. Mögliche Techniken zur Nutzung des Expertenwissens von vorangegangenen Projekten sind beispielsweise Checklisten.

In der kreativen Phase werden dagegen weitere vom System ausgehende Gefährdungen auf Basis der funktionalen Anforderungsspezifikation identifiziert /De Lemos et al. 95/. Verwendete Techniken dafür sind beispielsweise die systemorientierte FMECA (Failure Modes and Effects Criticality Analysis) /DIN 25448/ oder die funktionsbezogene Fehleranalyse (FFA) /SAE ARP 4754/4761/. Beide Techniken untersuchen systematisch die funktionale Anforderungsspezifikation und identifizieren ausgehend von typischen Fehlverhalten mögliche Gefährdungen. Nach /Balzert 01/ eignen sich des Weiteren zur Gefährdungsidentifikation Brainstormingtechniken sowie Was-Wäre-Wenn-Studien.

Je nach den Ereignissen in der Systemumwelt oder dem Ergreifen von geeigneten Schutzmaßnahmen kann eine Gefährdung zu verschiedenen Folgen führen. Ziel der Folgeanalyse ist zum einen, die Bedingungen zu identifizieren, bei denen eine Gefährdung einen Unfall verursacht. Zum anderen sollte die Wahrscheinlichkeit für einen solchen Unfall bestimmt werden. Dazu ist es erforderlich, die Wahrscheinlichkeiten für den Erfolg oder das Versagen der Schutzmaßnahmen sowie die Wahrscheinlichkeiten relevanter Ereignisse in der Systemumwelt zu bestimmen. Diese Wahrscheinlichkeiten müssen anschließend gemäß ihrer Abhängigkeiten verknüpft werden. Ergebnis dieses Vorgehens ist eine Aussage über die Häufigkeit *C*, mit der eine Gefährdung zu der entsprechenden Konsequenz führt. Wird diese Häufigkeit dann mit der Unfallschwere *F* verknüpft, so ergeben sich quantifizierte Aussagen über die Auswirkungen der Gefährdungen.

Zur Durchführung einer Folgeanalyse können Ereignisbaumanalysen (*event-tree-analysis*) verwendet werden. Hierbei werden die Beziehungen zwischen den Gefährdungen und Folgen anhand eines Baumes dargestellt, wobei dessen Wurzel eine bereits identifizierte Gefährdung und dessen Blätter die identifizierten Konsequenzen dieser Gefährdung charakterisieren. Der Aufbau dieses Ereignisbaumes stellt dar, unter welchen Bedingungen und mit welcher Wahrscheinlichkeit sich Gefährdungen sowie deren identifizierte Konsequenzen einstellen. Eine weitere gebräuchliche Technik ist nach /Pumfrey 99/ die Ursache-Wirkungs-Analyse (*cause consequence analysis*). Die Vorgehensweise der Ursache-Wirkungs-Analyse ähnelt der Ereignisbaumanalyse. Sie unterscheidet sich jedoch von der Ereignisbaumanalyse in der graphischen Darstellung. Darüber hinaus ist bei der Ursache-Wirkungs-Analyse eine Betrachtung der zeitlichen Abhängigkeiten zwischen den Ereignissen möglich.

In der Risikobewertungsphase werden die konkreten Sicherheitsanforderungen des zu entwickelnden Systems bestimmt. Sie bestehen aus den identifizierten Gefährdungen sowie den maximalen Auftrittswahrscheinlichkeiten dieser Gefährdungen. In der Risikobewertung ist es daher notwendig, jeder Gefährdung H_j eine tolerierbare Gefährdungsrate (THR_j) zuzuweisen.

Für die Bestimmung der THR_j ist es zunächst erforderlich, das tolerierbare individuelle Risiko (*TIR*) zu ermitteln. Dieses charakterisiert das maximale Risiko, welchem ein Individuum bei der Benutzung des Systems ausgesetzt werden darf. Bestimmt wird das TIR mit Hilfe von Risikoakzeptanzkriterien /Okstad, Hokstad 01/, welche sich bei verschiedenen Anwendungsgebieten unterscheiden /Braband 98,01/.
Ist das *TIR* für das gesamte System bekannt, werden die THR_j für die identifizierten Gefährdungen derart budgetiert, dass die folgende Ungleichung /Braband Lennartz 99/ erfüllt ist.

$$TIR > \sum_{Gefährdungen H_j} \left(THR_j \times \sum_{Unfälle A_k} C_j^k \times F^k \right)$$

Auf der rechten Seite der Ungleichung werden die von den identifizierten Gefährdungen H_j ausgehenden Risiken addiert. Sie sollten kleiner sein als das *TIR*. Das Risiko einer Gefährdung ergibt sich aus der THR_j und den in der Folgeanalyse ermittelten Konsequenzen A_k der Gefährdung.

2.2 Spezifikation der funktionalen Anforderungen

Voraussetzung für die Spezifikation der Qualitätsanforderungen ist eine präzise Spezifikation der funktionalen Anforderungen. Daher werden im Folgenden mit den szenarioorientierten, zustandsorientierten und logischen Spezifikationsnotationen, drei Hauptkategorien für die Spezifikation von funktionalen Anforderungen vorgestellt /Balzert 01/, /Sommerville, Sawyer 97/, /Douglass 99/, /Broy 97/. Neben diesen Spezifikationsnotationen lassen sich Anforderungen auch mit natürlicher Sprache spezifizieren. Auf Grund mangelnder Eindeutigkeit der natürlichen Sprache wird dies für sicherheitskritische Systeme jedoch nicht empfohlen /Leveson 95/.

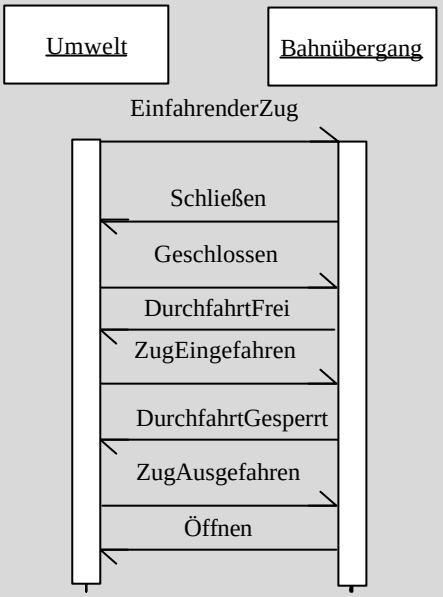
Szenarioorientierte Anforderungsspezifikation

Die szenarioorientierte Anforderungsspezifikation beschreibt die funktionalen Anforderungen an das System durch Szenarien, welche einzelne Ereignisfolgen bei einer speziellen Benutzung des Systems repräsentieren /Prowell, Poore 03/. Diese Ereignisfolgen sind exemplarisch und werden stets durch einen externen Stimulus initiiert, der von einem Sensor erkannt wird. Ausgelöst wird dieser Stimulus durch einen Akteur, welcher ein spezielles Ziel verfolgt. Dieses Ziel wird durch die Nachbedingung des Szenarios spezifiziert. Darüber hinaus muss spezifiziert werden, unter welchen Bedingungen das Szenario ausgeführt wird. Dies erfolgt durch die Spezifikation von Vorbedingungen /Meyer 97/. Die Spezifikation des eigentlichen Szenarios erfolgt durch die Beschreibung der Systemreaktionen auf den Stimulus. Diese Reaktionen können aus vom System ausgeführten Berechnungen bzw. Interaktionen mit anderen Akteuren bestehen.
Szenarien, die zu einem gemeinsamen Ziel gehören, werden zu Anwendungsfällen (*use cases*) zusammengefasst. Sie sind in der objektorientierten Softwarekonstruktion ein gebräuchliches Spezifikationsmittel /Booch et al. 99/. Allgemein stellen Anwendungsfälle typische Interaktionen zwischen der Systemumgebung und dem System dar, um das Ziel zu erfüllen /Jacobson et al. 92/, /Cockburn 97/. Ein Anwendungsfall besteht zumeist aus einem primären Szenario und ein oder mehreren sekundären Szenarien. Das primäre Szenario repräsentiert den optimistischen Hauptpfad, d.h. das angenommene optimale Verhalten des Systems bzw. der Systemumgebung. Die sekundären Szenarien beschreiben sowohl alternative Pfade als auch das Verhalten des Systems in Fehlersituationen.
Zur Formulierung von Szenarien gibt es eine Reihe von unterschiedlichen Spezifikationsnotationen. Gebräuchlich sind dabei Sequenzdiagramme /UML 2.0/ und Message Sequence Charts (MSC) /ITU Z.120/. Sequenzdiagramme basieren auf den in /Jacobsen et al. 92/ vorgestellten Interaktionsdiagrammen (*interaction diagrams*). Sie sind in der Praxis verbreitet /Booch et al. 99/, /Fowler,Scott 97/ und werden bei der objektorientierten Analyse zur Verfeinerung von Anforderungen mit der UML verwendet. MSC wurden durch die ITU standardisiert /ITU Z.120/ und werden vorwiegend in der Telekommunikationsindustrie eingesetzt.
Beide Spezifikationsnotationen haben eine ähnliche Ausdrucksmächtigkeit und Syntax /Rudolph et al. 99/. Sie stellen beide die interagierenden Akteure im oberen Teil des Diagramms nebeneinander auf der x-Achse dar. Die y-Achse repräsentiert die Zeit. Jedem Akteur ist eine vertikale Lebenslinie zugeordnet, welche gestrichelt dargestellt ist. Ist der Akteur verfügbar, so wird die Lebenslinie durch einen dicken vertikalen Balken überschrieben. Ein Nachrichtenaustausch zwischen zwei Akteuren wird durch einen Pfeil zwischen

den beiden Lebenslinien repräsentiert. Unterschiede ergeben sich im Formalisierungsgrad der beiden Notationen. Während MSC eine formal definierte Semantik besitzen /Katoen, Lambert 98/, /Mauw 96/, /Jonsson, Padilla 01/, ist die Semantik der Sequenzdiagramme in der UML noch nicht vollständig formalisiert /Latronico, Koopman 01/. Daher wird angestrebt, eine Semantik auch für die Sequenzdiagramme zu definieren und diese an die Semantik der MSC anzupassen /Rudolph et al. 99/. Daraus würde eine Vereinheitlichung beider Notationen folgen.

Beispiel 2.1

Im dargestellten Sequenzdiagramm wird am Beispiel eines Bahnübergangs die szenariobasierte Anforderungsspezifikation veranschaulicht. Ausgelöst wird das Szenario durch einen Zug, der sich einem Sensor nähert. Durch die weitere Ereignissequenz wird die Anforderung an das Verhalten des Bahnübergangs beschrieben.



Zustandsorientierte Anforderungsspezifikationen

Zustandsorientierte Anforderungsspezifikationen beschreiben die funktionalen Anforderungen des zu entwickelnden Systems in Abhängigkeit der möglichen Systemzustände. Im Gegensatz zu der szenarioorientierten Anforderungsspezifikation ist die Verhaltensbeschreibung jedoch nicht exemplarisch, sondern repräsentativ und generisch. Dadurch lassen sich aus dem Zustandsmodell die einzelnen Szenarien ableiten. Für die Definition des Zustandsmodells ist eine Definition der Systemzustände, der Übergänge zwischen den Systemzuständen und der Reaktion auf Stimuli in Abhängigkeit von den verschiedenen Systemzuständen erforderlich. Verwendbar sind dazu endliche Automaten (FSM finite state machines), welche nach /Hopcroft, Ullmann 79/ durch das folgende Tupel definiert sind: $A = (S, s_0, T, I, O, F)$.
In diesem Tupel beschreibt S die Menge der möglichen Zustände, in denen sich das System befinden kann und s_0 ist ein Element dieser Menge, welches den Startzustand des Systems bei der Initialisierung bestimmt. Die möglichen Systemstimuli werden durch das Eingabealphabet I und die mögliche Ausgabe durch das Ausgabealphabet O charakterisiert. Der Übergang zwischen den Zuständen wird durch die Transitionsrelation T wie folgt spezifiziert:

$$T \subseteq I \times S \times S$$

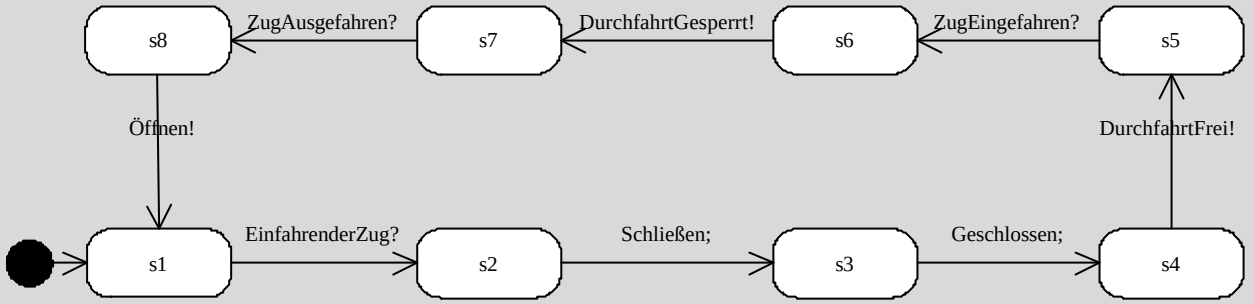
Dabei geht ein System vom Zustand $s \in S$ in den Zustand $s' \in S$ über, wenn das System die Eingabe i empfängt und $\langle i, s, s' \rangle$ ein Element von T ist.
Das funktionale Verhalten in Abhängigkeit von den Zuständen wird durch die Relation F wie folgt spezifiziert:

$F \subseteq I \times S \times O$

Endliche Automaten werden jedoch mit zunehmender Anzahl der zu modellierenden Zustände komplexer. Zur Beherrschung dieser Komplexität werden hierarchische Automaten verwendet. Beispiele dafür sind Statecharts /Harel 87/ und ROOM-Charts /Selic et al. 94/.
Neben den endlichen Automaten werden für die zustandsorientierte Modellierung auch auf Petrinetzen basierende Notationen verwendet. Beispiele für Petrinetze mit einem objektorientierten Fokus sind CO-OPN/2 (Concurrent Object-Oriented Petri Nets) /Biberstein et al. 1997/, CLOWN (CLass Orientation With Nets) /Battiston et al. 96/ und OCPN (Object Coloured Petri Nets) /Maier, Moldt 97/.

Beispiel 2.2

Durch den unten dargestellten Automaten wird die zustandsorientierte Anforderungsspezifikation an dem bereits vorgestellten Bahnübergangsbeispiel gezeigt. Aus diesem Automaten lässt sich auch das exemplarische Szenario aus dem Beispiel 2.1 ableitet.



Logische Anforderungsspezifikation

Logische Modelle charakterisieren das gewünschte Verhalten des Systems mit einer Menge von logischen Aussagen. Der Vorteil dieser Modelle ist zum einen eine lösungsneutrale Spezifikation und zum anderen die Möglichkeit, mit Hilfe von Beweisen und Modellchecking /Henzinger et al. 94/, /Bozga et al.98/ Eigenschaften des Modells nachzuweisen. Für softwareintensive technische Systeme eignen sich besonders temporale Logiken /Manna, Pneuli 92/, /Lamsweerde 01b/. Durch diese temporalen Logiken ist eine Spezifikation des gewünschten Verhaltens in Abhängigkeit vom aktuellen, vergangenen und zukünftigen Zustand möglich /Emerson 90/. Dazu werden prädikatenlogische und aussagenlogische Sprachen um die in Tabelle 2-1 dargestellten temporalen Operatoren erweitert.

Tabelle 2-1 Übersicht über temporale Operatoren

Bezeichner	Operatorsymbol/ alternativ	Bedeutung	Veranschaulichung
Next	δϕ/Xϕ	ϕ gilt im nächsten Zeitpunkt	δϕ |ϕ----------
Eventually	◊ϕ/Fϕ	ϕ gilt irgendwann	◊ϕ |------ϕ----------
Henceforth	Ωϕ/Gϕ	ϕ gilt von nun an immer	Ωϕ |ϕϕϕϕϕϕϕϕϕϕϕϕ
Until	ϕYψ/ϕUψ	ϕ gilt bis ψ eintritt	ϕYψ |ϕϕϕϕψ-------
Wait	ϕΩψ/ϕWψ	ϕ gilt bis ψ eintritt oder falls ψ nicht eintritt, gilt ϕ immer (schwaches until)	ϕΩψ |ϕϕϕϕψ------- oder ϕΩψ |ϕϕϕϕϕϕϕϕϕϕϕϕ

Beispiel 2.3

Bei der Formulierung von Anforderungen an einen Bahnübergang in temporal logischen Aussagen ist die Beschreibung des Verhaltens der Umwelt von der Beschreibung des Verhaltens des Systems zu trennen. Dies ist notwendig, da es sich nur bei der Beschreibung des Systemverhaltens um die Formulierung von Anforderungen handelt, während es sich bei der Beschreibung des Verhaltens der Umwelt um Annahmen handelt. Die Vermischung von Annahmen und Anforderungen ist nach /Manna, Pneuli 92/ zu vermeiden.

Verhalten der Umwelt:

1. (EinfahrenderZug ∧ DurchfahrtFrei) Ω ZugEingefahren
2. ZugEingefahren Ω ZugAusgefahren
3. ZugAusgefahren Ω EinfahrenderZug

Verhalten des Systems:

1. (EinfahrenderZug ∧ (¬DurchfahrtFrei) ∧ SchrankenOffen)
   Ω (((¬DurchfahrtFrei) ∧ (¬SchrankenOffen) ∧ EinfahrenderZug))
   Ω ((DurchfahrtFrei ∧ (¬SchrankenOffen) ∧ EinfahrenderZug))
2. ZugEingefahren δ (¬DurchfahrtFrei)
3. (ZugAusgefahren ∧ (¬DurchfahrtFrei)) ◊ SchrankenOffen

2.3 Spezifikation der Performanz- und Echtzeitanforderungen

Die Echtzeitfähigkeit ist bei softwareintensiven technischen Systemen von besonderer Bedeutung /Douglass 99/, /Gomaa 00/. Daher ist es erforderlich, zusätzlich zu den funktionalen Anforderungen auch Performanz und Echtzeitanforderungen zu spezifizieren. Diese Anforderungen beschränken durch Zeitbedingungen die tolerierbare Reaktionszeit des Systems auf einen Stimulus. Nach /Alur, Dill 94/ kann eine Zeitbedingung d wie folgt induktiv definiert werden:

(*a*) $d := x \otimes c$,
(*b*) $d := \neg d_1$
(*c*) $d := d_1 \oplus d_2$

In der Formel (a) entspricht x einer Zeitvariablen ($x \in P_{\geq 0}$), c einem konstanten Wert und $\otimes$ einem Platzhalter für einen Vergleichsoperator ($\leq, \geq, >, =, <$). Dadurch lassen sich einfache Zeitbedingungen, wie z. B. x<2s oder x=10ms, beschreiben. Im Definitionsteil (b) und (c) werden diese einfachen Zeitbedingungen zu logischen Formeln erweitert. Somit ist es möglich, mehrere einfache Zeitbedingungen zu komplexeren Zeitbedingungen zusammenzufassen. Der Platzhalter $\oplus$ ist dementsprechend ein Element der Menge der logischen Operatoren ($\wedge, \vee, \Leftrightarrow, \Rightarrow$).
Mit Hilfe dieser Zeitbedingungen ist eine Erweiterung der funktionalen Anforderungsspezifikationen hinsichtlich der Spezifikation von Performanz und Echtzeitanforderungen möglich. Diese Erweiterung wird im Folgenden am Beispiel der zeitannotierten Sequenzdiagramme, der zeitannotierten Automaten und der Echtzeitlogiken dargestellt.

Zeitannotierte Automaten

Für die Erweiterung von endlichen Automaten für Echtzeitspezifikationen ist es erforderlich, die akzeptierten Eingabe- und Zustandsfolgen zusätzlich durch Zeitbedingungen zu beschränken. Dazu eignen sich die in /Alur, Dill 94/ beschriebenen zeitannotierten Automaten (timed automata). Diese erweitern zunächst das Eingabealphabet. Dazu wird jedem Stimulus σ_i sein Auftrittszeitpunkt τ_i zugeordnet. Ein Element des Eingabealphabets wird demnach durch ein Tupel (σ,τ) repräsentiert. Ein zeitannotierter Automat besitzt zusätzlich eine Menge von Uhren C (*Clocks*). Eine Uhr $c \in C$ kann mit dem Befehl *c:=0* unabhängig von den anderen Uhren zurückgesetzt werden. Des Weiteren ist es möglich, Zeitbedingungen für einzelne Uhren zu definieren. Die Menge der Zeitbedingungen aller Uhren wird dann mit $\Phi(C)$ bezeichnet /Alur, Dill 94/. Mit diesen Voraussetzungen lässt sich ein zeitannotierter Automat schließlich durch das folgende Tupel definieren:

$$A = (S, s_0, C, I, O, T, F)$$

Ähnlich wie bei den endlichen Automaten bestimmt S die Menge der möglichen Zustände, s_0 den Startzustand, I das Eingabealphabet und O das Ausgabealphabet. Die Menge der Transitionen T ist definiert durch:

$T \subseteq I \times \Phi(C) \times S \times S \times 2^c$

Ein Element $\langle i, d, s, s', l \rangle$ aus der Transitionsrelation T beschreibt einen Zustandsübergang vom Zustand $s \in S$ in den Zustand $s' \in S$. Voraussetzung dafür ist, dass der Automat die Eingabe *i* aus dem Eingabealphabet *I* empfängt und die Zeitbedingungen d der Uhren erfüllt sind. Beim Schalten der Transition werden neben dem Zustandsübergang auch die in der Menge l spezifizierten Uhren zurückgesetzt.

Neben dem Übergang in einen anderen Zustand kann ein Automat auch mit einer Ausgabe aus dem Ausgabealphabet *O* auf einen Stimulus reagieren. Diese Reaktion des Automaten ist durch die Relation *F* bestimmt, welche ähnlich der Transition *T* wie folgt definiert ist:

$F \subseteq I \times \Phi(C) \times S \times O \times 2^c$

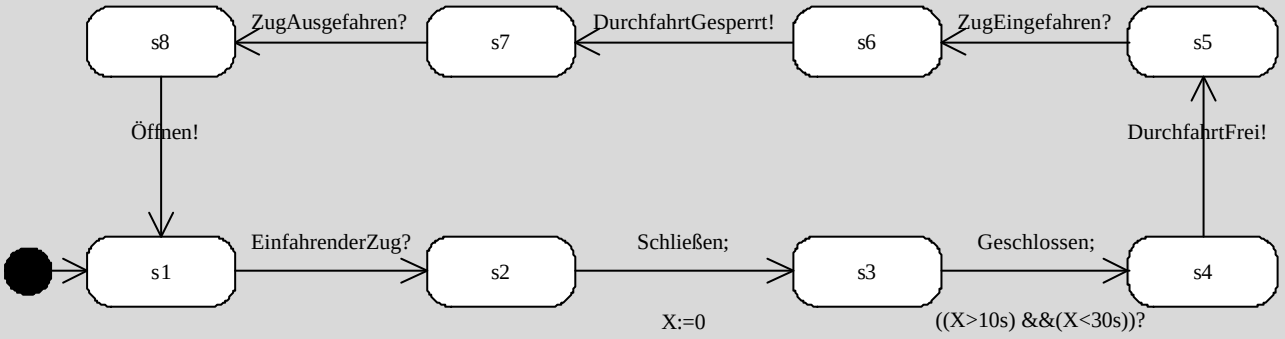
Beispiel 2.4

Zur Veranschaulichung der Spezifikation mit zeitannotierten Automaten wird das Beispiel 2.2 um die folgenden Zeitbedingungen erweitert:

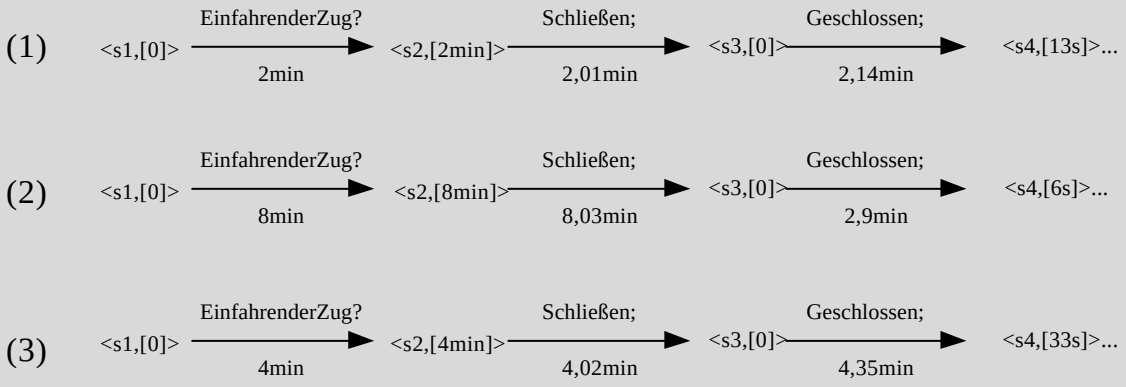
- das Schließen der Schranken muss länger als 10 Sekunden dauern

- das Schließen der Schranken muss innerhalb von 30 Sekunden abgeschlossen sein

Die untere Zeitschranke ist notwendig, um Passanten des Bahnübergangs nicht durch das zu schnelle Fallen einer Schranke zu gefährden. Die obere Zeitschranke wird durch die minimale Zeit bestimmt, die ein Zug bis zum Eintreffen im Bahnübergangsbereich benötigt.



Ausgehend vom zeitnotierten Automaten sind zum Beispiel die folgenden drei möglichen Sequenzen denkbar. Die erste Sequenz beschreibt ein korrektes Verhalten des Systems. Bei der zweiten wird die untere Zeitschranke verletzt und bei der dritten Sequenz die obere Zeitschranke.

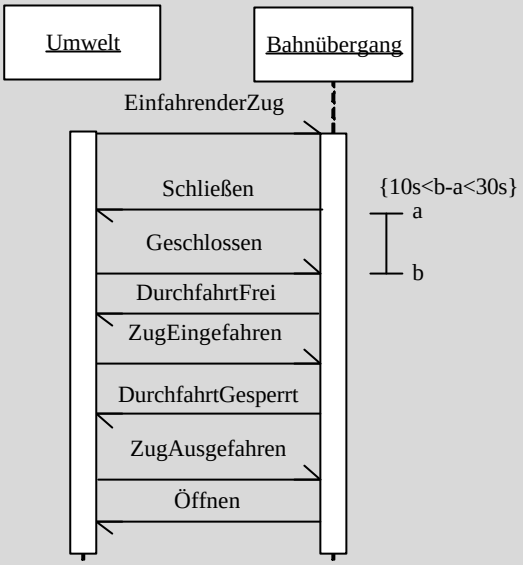


Zeitannotierte Sequenzdiagramme

Zeitannotierte Sequenzdiagramme /Gomaa 00/ stellen eine einfache Möglichkeit dar, Zeitbedingungen zwischen zwei Ereignissen zu spezifizieren. Dazu wird durch eine Variable der Zeitpunkt repräsentiert, zu dem ein Ereignis erzeugt oder empfangen wird. Diese Variable wird zur Spezifikation der Zeitbedingung genutzt. Realisierbar sind zeitannotierte Sequenzdiagramme in der UML über OCL Constraints /Douglass 98/.

Beispiel 2.5

Ausgehend von Zeitbedingungen für den Bahnübergang aus dem vorherigen Beispiel lässt sich die sequenzorientierte Anforderungsspezifikation wie folgt erweitern:



Echtzeitlogiken

Echtzeitlogiken bauen auf den temporalen Logiken auf und erweitern diese für die Spezifikation von Zeitbedingungen. Im Gegensatz zu temporalen Logiken ist es somit möglich, die temporale Ordnung nicht nur qualitativ, sondern auch quantitativ zu definieren. Dazu werden in /Alur, Henzinger 89,93/ und /Alur et al. 96/ Echtzeitlogiken vorgeschlagen, welche die temporalen Operatoren um die Zeitintervalle erweitern, in denen sie gelten. Ein Beispiel für eine solche Echtzeitlogik ist MITL (Metric Interval Teporal Logic). In MITL werden Zeitintervalle durch eine untere Schranke a und eine obere Schranke b bestimmt ($b \geq a$). Beide Schranken beziehen sich auf eine gemeinsame Zeitbasis (z B. Sekunden). In Anhängigkeit von den Intervallgrenzen sind somit die folgenden Zeitintervalle möglich: [a,b],[a,b),[a,∞),(a,b],(a,b) oder (a, ∞). Mit diesen Zeitintervallen kann eine Zeitbedingung für jeden temporalen Operator präzise definiert werden. Beispielsweise würde der Operator $\phi Y_{[0,2)}\psi$ bedeuten, dass nachdem die Formel ϕ gilt, es weniger als zwei Sekunden dauert, bis die Formel ψ gilt.
Ein anderer Ansatz ist, die temporale Logik um Operatoren zu erweitern, welche eine Zeitbedingung explizit spezifizieren. In der SCL (State Clock Logic) /Raskin, Schobbens 97/ werden dazu die Operatoren *prophecy* $>_{\sim c}$ und *history* $<_{\sim c}$ eingeführt. Bei diesen beiden Operatoren stellt *c* eine zeitbezogene Konstante und ~ einen Platzhalter für einen Vergleichsoperator ($\leq, \geq, >, =, <$) dar. Der *prophecy* Operator $>_{\sim c}$ definiert durch den tiefgestellten Term, zu welchem Zeitpunkt die Formel in der Zukunft erfüllt wird. Entsprechend definiert der *history* Operator $<_{\sim c}$, zu welchem Zeitpunkt die Formel in der Vergangenheit erfüllt wurde. Beispielsweise ist durch die SCL-Formel $>_{=3}\phi$ festgelegt, dass die Formel ϕ genau nach drei Sekunden gilt. In /Raskin, Schobbens 97/ wurde nachgewiesen, dass sich mit der SCL alle bildbaren Formeln der MITL beschreiben lassen. So entspricht beispielsweise die Formel $\phi Y_{[0,2)}\psi$ in MITL der SCL-Formel $\phi Y \psi \wedge >_{<2} \psi$.

2.4 Spezifikation der probabilistischen Anforderungen

Probabilistische Anforderungen charakterisieren die gewünschte Ausprägung der Zuverlässigkeits-, Wartbarkeits- und Verfügbarkeitseigenschaften des zu entwickelnden Systems /Liggesmeyer 00/. Diese probabi-

listischen Anforderungen beziehen sich entweder auf einzelne funktionale Anforderungen oder auf das gesamte System. Bei einem Bezug der Anforderung auf das gesamte System gilt diese Anforderung demnach für alle funktionalen Anforderungen. Sie ist somit für alle funktionalen Anforderungen identisch. Bei einer Zuweisung der Zuverlässigkeits-, Wartbarkeits- und Verfügbarkeitsanforderungen auf eine einzelne funktionale Anforderung, ist eine unterschiedliche Gewichtung möglich. Dies erlaubt, gezielt die kritischen funktionalen Anforderungen mit stärkeren Zuverlässigkeits-, Wartbarkeits- und Verfügbarkeitsanforderungen zu spezifizieren.
Im Folgenden wird einzeln charakterisiert, auf welche Weise die probabilistischen Eigenschaften für die Zuverlässigkeit, Wartbarkeit und Verfügbarkeit zu spezifizieren sind.

2.4.1 Zuverlässigkeit

Nach /Lyu 99/, /Liggesmeyer 00/ und /Birolini 99/ kann die Zuverlässigkeit eines Systems durch verschiedene Maße und Wahrscheinlichkeitsfunktionen spezifiziert werden. Ein verbreitetes Maß hierfür ist die *Mean Time to Failure* (*MTTF*). Diese spiegelt den Erwartungswert für die Zeitspanne bis zu einem Ausfall wieder. Bei reparierbaren Systemen wird der Erwartungswert zwischen zwei aufeinander folgenden Ausfällen als *Mean Time between Failure* (*MTBF*) charakterisiert.
Alternativ dazu kann die Zuverlässigkeit durch die Überlebenswahrscheinlichkeitsfunktion *R(t)* beschrieben werden. Diese beschreibt die Wahrscheinlichkeit, mit der eine Funktion zum Zeitpunkt *t* korrekt funktioniert. Für die Spezifikation der Überlebenswahrscheinlichkeitsfunktion sollte eine Verteilungsfunktion gewählt werden, welche das Ausfallverhalten des betrachteten Systems adäquat beschreibt. Häufig verwendet werden in diesem Zusammenhang Exponential- und Weibullverteilungen /Kececioglu 91/.

$$R_{exponential}(t) = e^{-\lambda t}$$

$$R_{weibull}(t) = e^{-(\gamma t)^{\beta}}$$

Für die Spezifikation der Überlebenswahrscheinlichkeitsfunktionen ist es erforderlich, die einzelnen Parameter zu bestimmen. Für die Exponentialverteilung ist dazu ausschließlich die Ausfallrate λ als Parameter anzugeben. Bei der Weibullverteilung ist der Formparameter β und der Ortsparameter γ anzugeben.
Das Komplement der Überlebenswahrscheinlichkeitsfunktion ist die Ausfallwahrscheinlichkeitsfunktion *F(t)*. Sie gibt die Wahrscheinlichkeit an, mit der ein System ausfallen wird oder eine funktionale Anforderung nicht erbringt.
Das Ausfallverhalten des Systems wird bei der Exponentialverteilung von der Ausfallrate λ beeinflusst. Dabei wird davon ausgegangen, dass die Ausfallrate während der gesamten Lebenszeit konstant ist. Dies ist bei technischen Systemen jedoch nicht gegeben /Birolini 99/, da sich die Ausfallrate während der Lebenszeit des Systems ändert. Dabei sind die folgenden drei charakteristischen Phasen feststellbar:

- Frühausfallphase: In dieser Phase vermindert sich die Ausfallrate λ. Diese Phase ist gekennzeichnet durch Materialschwächen, Produktionsfehler, aber auch durch Konstruktionsfehler.
- Konsolidierungsphase: Diese Phase ist charakterisiert durch eine konstante Ausfallrate λ. Die Auswirkungen der Frühausfälle sind abgeklungen und Abnutzungserscheinungen können noch vernachlässigt werden.
- Spätausfallphase: Die Ausfallrate λ steigt mit zunehmender Betriebszeit immer stärker an. Zu den Ausfallursachen gehören vermehrt Ausfälle durch Alterung, Abnutzung, Verschleiß usw.

Die einzelnen Phasen und der typische Verlauf der Ausfallrate werden in Abbildung 2-3 dargestellt.

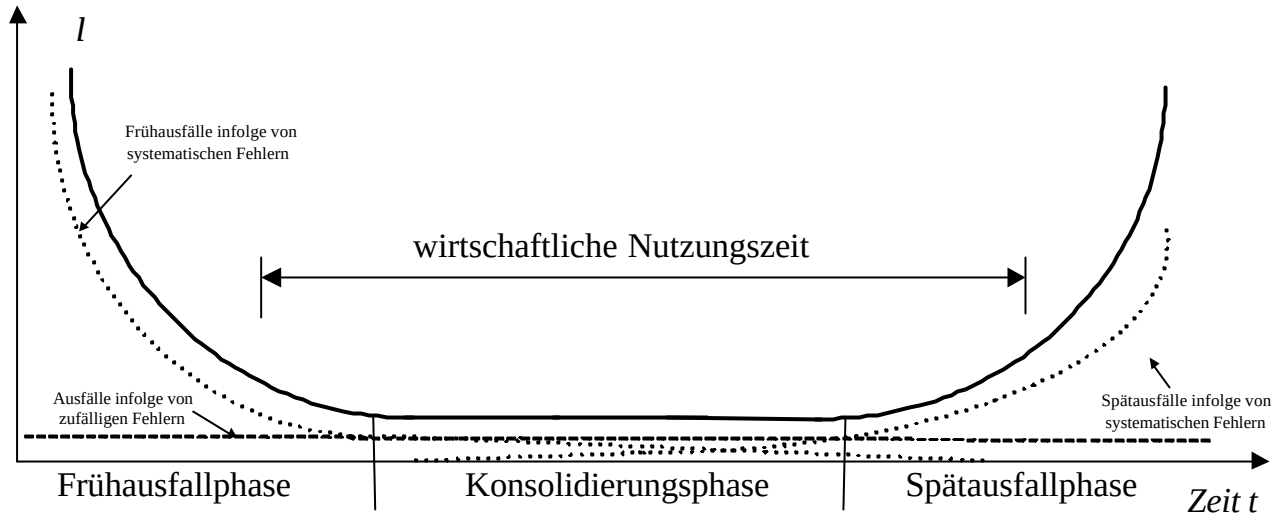


Abbildung 2-3 Ausfallrate eines technischen Systems nach /Birolini 99/

Die Ausfallrate kann demnach als Funktion $l(t)$ über der Lebenszeit des Systems spezifiziert werden. Diese Funktion gibt die Wahrscheinlichkeit eines Ausfalls im Intervall *(t; t+ ? t]* an. Die Nebenbedingungen sind, dass *? t* gegen Null strebt und die Komponente zum Zeitpunkt *t* noch intakt war.
Die einzelnen Maße und Wahrscheinlichkeitsfunktionen (*R(t), F(t), f(t), l(t), l* und *MTTF*) sind nach /Birolini 99/ und /Liggesmeyer 00/ nicht unabhängig voneinander. Sie sind durch die im Folgenden dargestellten Transformationsformeln ineinander überführbar. Dadurch wird eine einheitliche Bewertung der Zuverlässigkeit von softwareintensiven technischen Systemen ermöglicht /Lyu 96/.
Die Ausfallrate $l(t)$ ergibt sich wie folgt aus der Überlebenswahrscheinlichkeitsfunktion:

$$l(t) = \frac{dF(t)/dt}{R(t)} = \frac{-dR(t)/dt}{R(t)}$$

Die Überlebenswahrscheinlichkeitsfunktion *R(t)* ist demnach:

$$R(t) = e^{-\int_0^t l(x)\,dx}$$

Die zeitliche Ableitung der Ausfallwahrscheinlichkeitsfunktion *F(t)* ergibt die Wahrscheinlichkeitsdichte der Lebensdauer *f(t).*

$$f(t) = \frac{dF(t)}{dt} \qquad F(t) = \int_{-\infty}^{t} f(t)dt$$

Zwischen der Überlebenswahrscheinlichkeitsfunktion und der Wahrscheinlichkeitsdichte der Lebensdauer besteht nach /MIL HDBK 338B/ der folgende Zusammenhang:

$$R(t) = \int_t^{\infty} f(t)dt$$

Nach /Birolini 99/ und /Lyu 96/ ergibt sich die *MTTF* wie folgt aus der Überlebenswahrscheinlichkeitsfunktion:

$$MTTF = \int_0^{\infty} t[f(t)]dt = \int_0^{\infty} t\left[-\frac{dR(t)}{dt}\right]dt = \int_0^{\infty} R(t)dt$$

Für eine konstante Ausfallrate besteht die folgende Beziehung zwischen der *MTTF* und der Ausfallrate l.

$$MTTF = \int_0^{\infty} R(t)dt = \int_0^{\infty} e^{-lt}dt = \left|\frac{e^{-lt}}{-l}\right|_0^{\infty} = \frac{1}{l}$$

Zwischen Ausfallwahrscheinlichkeitsfunktion und Zuverlässigkeitsfunktion existiert die folgende Beziehung:

$$R(t) = 1 - F(t)$$

2.4.2 Wartbarkeit

Ähnlich wie die Zuverlässigkeit kann auch die Wartbarkeit eines Systems mit probabilistischen Kenngrößen spezifiziert werden /MIL HDBK 338B/. Die einzelnen Maße der Wartbarkeit und deren Verhältnis zu den Maßen der Zuverlässigkeit sind in Tabelle 2-2 veranschaulicht.

Tabelle 2-2 Zusammenhang von Zuverlässigkeits- und Wartbarkeitsbeschreibungsfunktionen

Zuverlässigkeit	Wartbarkeit
Lebensdauer (Wahrscheinlichkeitsdichte) $f(t)$	Reparaturdauer (Wahrscheinlichkeitsdichte) $g(t)$
Überlebenswahrscheinlichkeitsfunktion und Ausfallwahrscheinlichkeitsfunktion $R(t) = \int_t^{\infty} f(t)\,dt \quad F(t) = \int_{-\infty}^{t} f(t)\,dt$	Reparaturwahrscheinlichkeitsfunktion $M(t) = \int_0^t g(t)\ \ dt$
Ausfallrate $l(t) = \frac{f(t)}{R(t)}$	Reparaturrate $m(t) = \frac{g(t)}{1 - M(t)}$
Mean time to failure (MTTF) $MTTF = \int_{-\infty}^{\infty} t\, f(t)\,dt = \int_0^{\infty} R(t)\,dt$	Mean time to repair (MTTR) $MTTR = \int_{-\infty}^{\infty} t\, g(t)\,dt$

2.4.3 Verfügbarkeit

Die Verfügbarkeit wird mit den Quotienten aus der Zeit, in der das System die funktionalen Anforderungen erfüllt (Funktionszeit) und der Gesamtlaufzeit des Systems spezifiziert. Die Funktionszeit des Systems kann durch die *MTBF* spezifiziert werden. Für die Gesamtlaufzeit des Systems müssen die Funktionszeit und die für Reparaturen benötigte Zeit- die Mean Time to Repair (MTTR) - addiert werden. Folglich lässt sich die Spezifikation der Verfügbarkeit aus der Wartbarkeit- und Zuverlässigkeitsspezifikation ableiten.

$$Verfügbarkeit = \frac{Funktionszeit}{Gesamtlaufzeit} = \frac{MTBF}{MTBF + MTTR}$$

Es ist ersichtlich, dass die Verfügbarkeit durch Verbesserung der Zuverlässigkeit oder der Wartbarkeit, also der Verringerung der Reparaturzeiten, positiv beeinflusst wird.

2.5 Spezifikation der Sicherheitsanforderungen

Grundlage für die Spezifikation von Sicherheitsanforderungen ist die Identifizierung und Beschreibung der Gefährdungen in der Risikoanalyse. Diesen Gefährdungen wird eine tolerierbare Auftrittswahrscheinlichkeit (*THP tolerable hazard probability*) zugeordnet, welche auf der Basis des Risikoakzeptanzkriteriums ermittelt wurde. In der Praxis kann die *THP* vereinfachend durch die tolerierbare Gefährdungsrate (*THR tolerable hazard rate*) spezifiziert werden. Dies ist möglich, da während der wirtschaftlichen Nutzungszeit eines Systems, wie in Abbildung 2-3 veranschaulicht, die tolerierbare Gefährdungsrate als konstant angenommen werden kann /Leveson 95/, /Braband 01/.
Für die weitere Verwendung in den nachfolgenden Entwicklungsphasen müssen die Gefährdungen und deren *THP* in die Anforderungsspezifikation integriert werden. Dazu wird eine Gefährdung als ein sicherheitsgefährdendes Verhalten betrachtet. Bei der Entwicklung des Systems muss garantiert werden, dass die Wahrscheinlichkeit dieses sicherheitsgefährdenden Verhaltens niedriger ist als die *THP*. Demnach beschreibt eine

Sicherheitsanforderung ein Fehlverhalten des Systems in Relation zu der tolerierbaren Auftrittswahrscheinlichkeit dieses Fehlverhaltens. Zu beachten ist hier aber, dass nicht nur bei einem Fehlverhalten ein Risiko vom System ausgeht. Auch ein korrektes Verhalten des Systems kann zu einem Risiko führen, wenn die entsprechende Gefährdung bei der Risikoanalyse nicht identifiziert wurde.
Für die identifizierten Gefährdungen sind nach /Levenson 95/, /Avizienis et al. 01/ und /Laprie 92/ drei grundlegende Kategorien von Fehlverhalten die Ursache.

- Eine funktionale Anforderung wird nicht erfüllt, wenn sie gefordert wird.
- Eine funktionale Anforderung wird korrekt erfüllt, hält aber die spezifizierten Zeitbedingungen nicht ein.
- Eine funktionale Anforderung wird fehlerhaft (nicht korrekt) ausgeführt.

Bei der ersten Kategorie entsteht eine Gefährdung, wenn eine funktionale Anforderung nicht erfüllt wird. Die Spezifikation dieser Gefährdungskategorie wird durch die Zuverlässigkeitsanforderungsspezifikation abgedeckt. Dabei sollte die spezifizierte Wahrscheinlichkeit, mit der das System eine funktionale Anforderung nicht erbringt, kleiner sein als die tolerierbare Auftrittswahrscheinlichkeit der Gefährdung.
Die zweite Kategorie enthält Gefährdungen, bei denen das System zwar korrekt reagiert, aber die Reaktion zu spät erfolgt. Fehlverhalten dieser Kategorie werden durch die Echtzeitanforderungen spezifiziert. Zusätzlich zu diesen Anforderungen sind hinsichtlich der Sicherheitsanforderungen jedoch noch die tolerierbaren Auftrittswahrscheinlichkeiten zu spezifizieren.
In die dritte Kategorie werden Gefährdungen eingeordnet, die durch die fehlerhafte Erfüllung der funktionalen Anforderungen entstehen. Dies erfolgt, wenn das System auf einen Stimulus mit einem nicht korrekten Verhalten reagiert oder eine nicht geforderte Systemreaktion erzeugt wird.
Für die Spezifikation der Gefährdungen der letzten Kategorien ist das Verhalten zu spezifizieren, welches das System auf keinen Fall offenbaren darf. Bei einer Spezifikation mit Sequenzdiagrammen erfolgt dies durch die Beschreibung verbotener Sequenzfolgen. Bei einer Spezifikation mit einem logischen Modell werden Formeln definiert, welche das System nicht erfüllen darf. Für die temporale Logik werden in /Bitsch 01/ Muster (*safety patterns*) vorgeschlagen, welche die Spezifikation erleichtern. Bei einer zustandsorientierten Spezifikation müssen die Zustände markiert werden, in denen vom System eine Gefährdung ausgeht.

Beispiel 2.6

Zur Veranschaulichung der Spezifikation von Sicherheitsanforderungen wird in Abbildung 2-4 und Abbildung 2-5 dargestellt, wie die sequenzorientierte, zustandsorientierte und logische Anforderungsspezifikationen geeignet erweitert werden. Als Beispiel wird dazu der Bahnübergang und die in Tabelle 2-3 spezifizierten Fehlverhalten verwendet.

Tabelle 2-3 Überblick über die Kategorien von Fehlverhalten für die Spezifikation von Sicherheitsanforderungen

Kategorie	Generelle Beschreibung des Fehlverhaltens	Beschreibung des Fehlverhaltens im Beispiel
1	Eine funktionale Anforderung wird nicht erfüllt, wenn sie gefordert ist.	Die Freigabe der Passage wird aufgehoben, obwohl die Schranke geöffnet wird.
2	Eine funktionale Anforderung wird korrekt erfüllt, hält aber ihre Zeitbedingung nicht ein.	Zwischen Beginn und Ende des Schließens der Bahnschranke sind nicht minimal 10 Sekunden oder maximal 30 Sekunden vergangen.
3	Eine funktionale Anforderung wird fehlerhaft (nicht korrekt) ausgeführt.	Die Schranke wird geöffnet, obwohl sich ein Zug im Bereich des Bahnübergangs befindet.

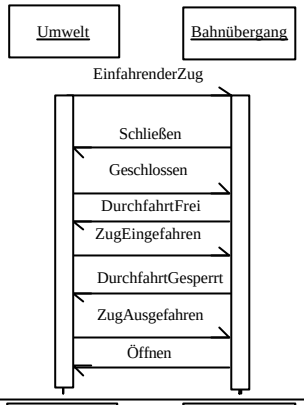
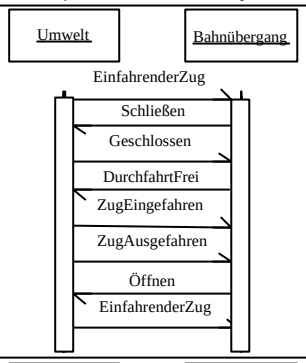
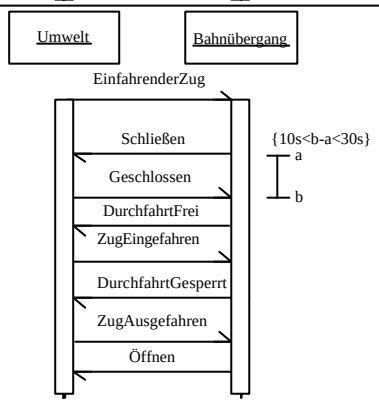
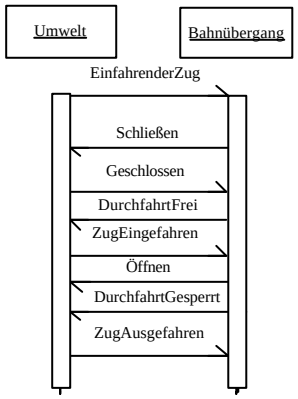
	Sequenzorientierte Anforderungsspezifikation	Logische Anforderungsspezifikation
Normalverhalten		Verhalten des Systems: 1. (EinfahrenderZug ∧ ¬DurchfahrtFrei ∧ SchrankenOffen) Ω (¬DurchfahrtFrei ∧ ¬SchrankenOffen ∧ EinfahrenderZug) Ω (DurchfahrtFrei ∧ ¬SchrankenOffen ∧ EinfahrenderZug) 2. ZugEingefahren δ ¬DurchfahrtFrei 3. (ZugAusgefahren ∧ ¬DurchfahrtFrei) $\Diamond$ SchrankenOffen
Kategorie 1: Die Freigabe der Passage wird nicht aufgehoben, obwohl die Schranke geöffnet wird.		¬ ($\Diamond$ (DurchfahrtFrei ∧ SchrankenOffen))
Kategorie 2: Zwischen Beginn und Ende des Schließens der Bahnschranke sind nicht minimal 10 Sekunden oder maximal 30 Sekunden vergangen.		¬ZugEingfahren Y ($\Diamond_{[10,30]}$ (¬ SchrankenOffen))
Kategorie 3: Die Schranke wird geöffnet, obwohl sich ein Zug im Bereich des Bahnübergangs befindet.		¬ ($\Diamond$ (ZugEingefahren ∧ SchrankenOffen))

Abbildung 2-4 Spezifikationstechniken (logisch und szenarioorientiert) zur Beschreibung von Sicherheitsanforderungen

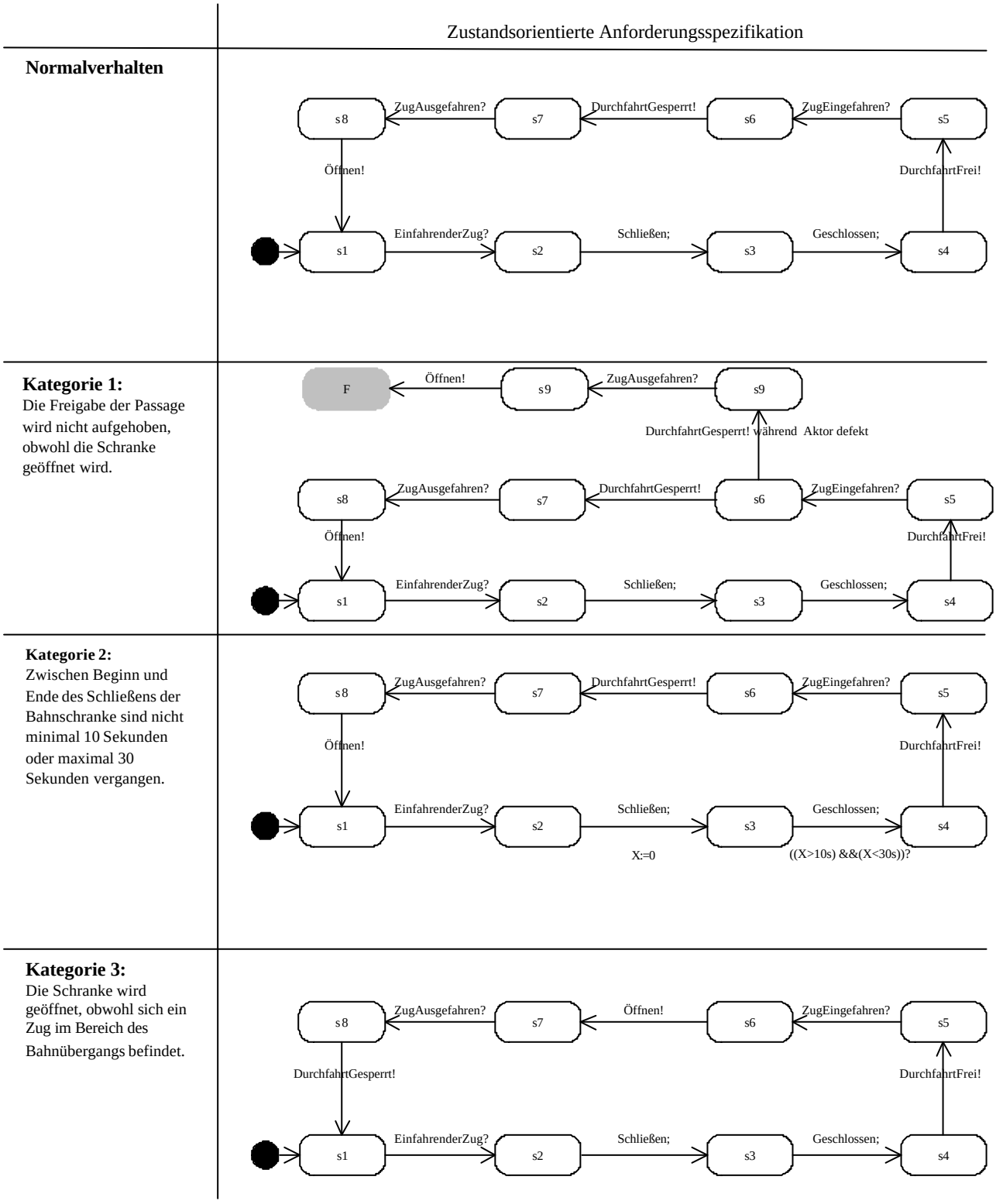


Abbildung 2-5 Spezifikationstechniken (zustandsorientiert) zur Beschreibung von Sicherheitsanforderungen

2.6 Analytische Qualitätssicherung

Aufgabe der analytischen Qualitätssicherung in der Anforderungsanalysephase ist die Prüfung der Qualität der Anforderungsspezifikation. Sie sollte nach /Lamsweerde 00,01c/ und /Robertson, Robertson 99/ die folgenden Eigenschaften aufweisen:

- Realisierbarkeit
- Konsistenz (Widerspruchsfreiheit)
- Verständlichkeit (z. B. Eindeutigkeit)
- Prüfbarkeit
- Adäquatheit
- Vollständigkeit

Die Qualitätssicherung dieser Eigenschaften erfolgt in der Anforderungsbewertungs- und der Anforderungsabnahmephase. Die Anforderungsbewertungsphase fokussiert dabei auf die Eigenschaften Realisierbarkeit, Konsistenz, Verständlichkeit und Prüfbarkeit. Bei der Realisierbarkeit wird geprüft, ob die Anforderungen mit den vorhandenen ökonomischen und technischen Mitteln umsetzbar sind. Für die ökonomische Realisierbarkeit werden Aufwands- und Ressourceschätzverfahren /Sneed 95/, /Antoniol et al. 95/ verwendet. Diese ermitteln näherungsweise die zu erwartenden Kosten bzw. den Ressourcenbedarf des Projektes. Die Erstellung von Prototypen kann darüber hinaus einen Aufschluss über die technische Realisierbarkeit einer Anforderungsspezifikation geben. Generell wird bei der Prüfung der Realisierbarkeit jedoch häufig auf Erfahrungswissen zurückgegriffen /Lamsweerde 00/.

Für die Prüfung der Qualitätseigenschaften Konsistenz und Widerspruchsfreiheit werden alle Anforderungen auf ihre gemeinsame technische Realisierbarkeit hin geprüft. Sind Anforderungen nicht gemeinsam realisierbar, so widersprechen sich diese Anforderungen /Heimdahl, Leveson 96/. In Abhängigkeit von der Anzahl der sich widersprechenden Anforderungen wird dies als binärer oder n-ärer Konflikt bezeichnet /Lamsweerde et al. 98/. Grundsätzlich entstehen Konflikte durch verschiedene Sichten und unterschiedliche Ansprüche und Erwartungen der Auftraggeber an das zu entwickelnde System /Easterbrook, Nuseibeh 96/, /Robinson, Volkov 97/. Zur Behebung der Konflikte müssen die sich widersprechenden Anforderungen als erstes identifiziert werden /Hunter, Nuseibeh 98/, /Heitmeyer et al. 96/. Anschließend werden die beteiligten Personen über den Konflikt informiert und eine gemeinsame Konfliktlösung gesucht.

Beim Prüfen der Verständlichkeit der Anforderungsspezifikation ist sicherzustellen, dass die Anforderungen durch alle beteiligten Personen auf gleiche Weise interpretiert werden. Dies setzt voraus, dass die verwendete Spezifikationsnotation beherrscht wird /Lamsweerde 00/. Des Weiteren sollten Fehlinterpretationen durch uneindeutige und unpräzise Anforderungen vermieden werden /Robertson, Robertson 99/. Daher sind synonyme und homonyme Begriffe in Anforderungsspezifikationen zu vermeiden oder in einem Glossar zu deklarieren.

Die Prüfbarkeit der Anforderungsspezifikation ist gegeben, wenn für jede Anforderung ein geeignetes Abnahmekriterium spezifiziert wurde. Mit diesen Abnahmekriterien ist in der Testphase feststellbar, ob das realisierte System auch tatsächlich die gestellten Anforderungen erfüllt.

In der Anforderungsabnahmephase werden schließlich die Adäquatheit und Vollständigkeit der Anforderungsspezifikation geprüft. Dies erfolgt durch die Validation der Anforderungen gegen die Ansprüche und Erwartungen der Auftraggeber. Stimmen die Anforderungen mit den Ansprüchen und Erwartungen der Auftraggeber überein, so ist die Anforderungsspezifikation adäquat. Sind des Weiteren alle Ansprüche und Erwartungen adäquat beschrieben, so ist die Anforderungsspezifikation vollständig.

2.7 Konstruktive Qualitätssicherung

Die konstruktiven Qualitätssicherungstechniken in der Anforderungsanalysephase forcieren die strukturierte Entwicklung und Formulierung der Anforderungsspezifikation. Genutzt werden dazu die im Folgenden dargestellten Techniken: Anforderungsverfolgung, Anforderungspriorisierung, Anforderungsschablonen und Zielgraphen. Nach /Balzert 01/ werden diese Techniken zum Anforderungsmanagement zusammengefasst, das in seiner Gesamtheit oder als einzelne konstruktive Qualitätssicherungstechniken durch Normen (z. B. /EN 50126/) und Qualitätsmanagement-Referenzmodelle (z. B. CMM /Paulk et al. 93/) gefordert wird.

Zielgraphen
Ein Zielgraph dient zur Strukturierung der Anforderungen und Ziele. Verwendet werden dazu attributierte Und/Oder-Graphen /Nilsen 71/. Diese stellen durch Oder- und Und-Verbindungen die Verfeinerungsbeziehungen zwischen Zielen und Anforderungen dar. Eine Und-Verbindung beschreibt dabei die Dekomposition eines Hauptziels in eine Menge von Unterzielen oder Anforderungen. Diese müssen erfüllt werden, um das Hauptziel zu erfüllen. Eine Oder-Verbindung beschreibt die Dekomposition eines Hauptziels in eine Menge von alternativen Unterzielen oder Anforderungen. Wird eine Alternative erfüllt, so ist das eine hinreichende Bedingung für die Erfüllung des Hauptziels. Oder-Verbindungen stellen somit Entscheidungspunkte während der Entwicklung dar, bei der sich das Entwicklungsteam für einen Oder-Pfad entscheiden muss. Dabei dienen Attribute der vom Ziel ausgehenden Kanten als Entscheidungshilfe.
Nach /Dardenne et al. 91/ besteht zwischen den Zielen und Anforderungen eines Projektes ein Zusammenhang analog zu der Implementierungsbeziehung zwischen Entwurf und implementierten Programmcode. Daher werden die Anforderungen durch Blattknoten im Zielgraphen repräsentiert. Dieser Zusammenhang wird beim Goal-Oriented Requirements Engineering (GORE) /Lamsweerde 01a/ genutzt, um die Vollständigkeit und Konsistenz von Anforderungen gegenüber der Menge der Ziele nachzuweisen und Konflikte zwischen Anforderungen durch Zielkonflikte zu identifizieren.

Anforderungsverfolgung
Die Anforderungsverfolgung ist eine während des gesamten Entwicklungsprozesses einsetzbare Qualitätssicherungstechnik /Jarke 98/, /Ramesh, Jarke 01/. Sie beschreibt nach /Gotel, Finkelstein 94/ die Entstehung und Realisierung einer Anforderung.
Für die Anforderungsentstehung sind die Abhängigkeiten zwischen den Ansprüchen und Erwartungen der Auftraggeber und den Anforderungen zu beschreiben. Dabei sollte eine Verfolgbarkeit in beide Richtungen möglich sein. Dadurch ergeben sich Vorteile bei der Identifikation von Inkonsistenzen /Lamsweerde et al. 98/, bei der Behebung von Anforderungskonflikten /Lamsweerde et al. 98/, /Heitmeyer et al. 96/, bei der Prüfung der Vollständigkeit /Lamsweerde 01c/ der Anforderungen und bei der Evolution der Anforderungen.
Für die Realisierung der Anforderungsverfolgung eignen sich die bereits beschriebenen Zielgraphen. Dazu werden den Ansprüchen und Erwartungen der Stakeholder Ziele zugewiesen und durch den Graphen die Abhängigkeiten zwischen diesen Zielen und den Anforderungen dargestellt.
Die Anforderungsrealisierung muss in den nachfolgenden Entwicklungsphasen beschrieben werden. Dazu werden Referenzen zwischen den Anforderungen und den Spezifikationselementen nachfolgender Entwicklungsphasen verwendet /Gericke, Liggesmeyer 02/, /Jarke 98/, /Heimdahl, Leveson 96/.

Anforderungspriorisierung
Aus wirtschaftlichen und technischen Gründen ist es oft schwer, alle Anforderungen zu realisieren. Daher müssen in einigen Fällen Anforderungen zurückgewiesen werden. Die Auswahl dieser zurückgewiesenen Anforderungen erfolgt jedoch oft ungezielt, da nach der Anforderungsdefinition alle Anforderungen eine gleichwertige Gewichtung besitzen. Aus diesem Grund ist es nach /Wiegers 99/ und /Karlsson 97/ erforderlich, die Anforderungen zu priorisieren. Mögliche Anforderungsprioritäten sind die Kundengewichtungen, die geschätzten Kosten, die Benefits sowie die mit der Realisierung verbundenen Risiken. Zur Identifikation der Kundengewichtungen ist es möglich, die Kunden nach einer absoluten Gewichtung zu fragen. Eine weitere Möglichkeit ist, die Anforderungen jeweils paarweise dem Kunden vorzulegen und ihn entscheiden zu lassen, welche Anforderung ihm wichtiger ist /Karlsson et al. 98/. Dies führt zu einer relativen Gewichtung der Anforderungen untereinander, ist jedoch bei umfangreichen Anforderungsspezifikationen mit einer großen kombinatorischen Komplexität verbunden. Zur Gewichtung der Anforderungen nach Kosten, Benefits und Risiken werden Aufwands- und Ressourceschätzverfahren /Sneed 95/, /Antoniol et al. 99/ verwendet. Als geeignete Technik zur Gewichtung der Anforderungen hat sich QFD (*quality function deployment*) als Verfahren bewährt /Karlsson et al. 98/, /Liggesmeyer 00/, /Brown 1991/.
Vorteile durch die Anforderungspriorisierung sind, neben der gezielten Selektion von zurückzuweisenden Anforderungen, zudem im Projektmanagement zu finden. Sie resultieren aus der Möglichkeit zu erkennen, welche Anzahl der Anforderungen einer Priorität bei einem Projektmeilenstein bereits umgesetzt sind. Dadurch ist möglich, einen Projektverzug frühzeitig zu erkennen und geeignete Maßnahmen einzuleiten.

Anforderungsschablonen
Für die strukturierte Spezifikation von Anforderungen werden in /Moreira et al. 02/, /Robertson, Robertson 00/ Anforderungsschablonen vorgeschlagen. Für softwareintensive technische Systeme sollte eine Anforderungsschablone den in Tabelle 2-4 dargestellten Aufbau haben.

Tabelle 2-4 Anforderungsschablone zur Spezifikation von Anforderungen für softwareintensive technische Systeme

Anforderungselement	Kurze Beschreibung
Bezeichner	Name der Anforderung oder des Ziels. Dieser Bezeichner muss projektweit eindeutig sein.
Typ	Funktionale Anforderung, Zuverlässigkeitsanforderung, Verfügbarkeitsanforderung, Sicherheitsanforderung, ...
Beschreibung	Kurze Beschreibung der Anforderung
Quelle	Bestimmung der Anforderungsquelle für die Anforderungsverfolgung
Auswirkung	Referenz auf relevante Elemente nachfolgender Spezifikationen (Architektur, Entwurf, Implementierung, Test)
Dekomposition	Spezifikation, wie ein Ziel in Teilziele oder Anforderungen dekomponiert werden kann. Verwendet werden dazu UND oder ODER Dekompositionen
Priorität	Beschreibt die Bedeutung der Anforderung mindestens auf einer Ordinalskala.
Konflikte	Referenziert andere Anforderungen, die mit der Anforderung in Konflikt stehen
Konstruktive Qualitätssicherung	Beschreibung und Spezifikation von geforderten konstruktiven Qualitätssicherungsmaßnahmen, welche zur Erfüllung der Anforderung oder des Ziels gefordert werden
Analytische Qualitätssicherung	Beschreibung von Prüfstrategien zum Nachweis der Erfüllung der Anforderung oder des Ziels im entwickelten System (Beispiel Testsequenzen)
Spezifikation	Eindeutige Beschreibung der Anforderung oder des Ziels in einer Anforderungsspezifikationsnotation oder -sprache

2.8 Zusammenfassung

In diesem Kapitel wurden ausgewählte Aspekte des Stands der Technik bei der Spezifikation und Qualitätssicherung der Anforderungsspezifikation vorgestellt. Besonders betrachtet wurde dabei die Spezifikation der für softwareintensive technische Systeme relevanten Qualitätsanforderungen Sicherheit, Verfügbarkeit, Zuverlässigkeit, Wartbarkeit und Echtzeit.
Als Ergebnis wurde gezeigt, dass sich diese Qualitätsanforderungen als Erweiterungen der funktionalen Anforderungsspezifikation definieren lassen. Zu diesen Erweiterungen zählen Zeitbedingungen, welche zur Spezifikation der Echtzeitanforderungen genutzt werden, und stochastische Maße und Kenngrößen, die zur Spezifikation von Zuverlässigkeits-, Wartbarkeits- und Verfügbarkeitsanforderungen verwendet werden. Für die Spezifikation der Sicherheitsanforderungen ist es erforderlich, die vom System ausgehenden, gefährlichen Fehlverhalten und deren tolerierbare Auftrittswahrscheinlichkeiten zu spezifizieren.

3 Qualitätseigenschaften im Architekturentwurf

Ausgehend von den in Kapitel 2 beschriebenen Spezifikationstechniken für die Qualitätsanforderungen Sicherheit, Zuverlässigkeit, Verfügbarkeit, Wartbarkeit und Echtzeitfähigkeit wird in diesem Kapitel ein Überblick über den derzeitigen Stand der Technik beim Architekturentwurf von softwareintensiven technischen Systemen unter Berücksichtigung dieser Qualitätseigenschaften gegeben. Dazu werden im Folgenden ein idealisiertes Vorgehensmodell für die Entwurfsphase, geeignete Spezifikationsnotationen für die Struktur- und Verhaltensspezifikationen sowie analytische und konstruktive Qualitätssicherungstechniken vorgestellt.

3.1 Idealisiertes Vorgehensmodell der Entwurfsphase

In dieser Arbeit wird für den Entwurf eines softwareintensiven technischen Systems das in Abbildung 3-1 dargestellte Vorgehensmodell bevorzugt. Dieses Vorgehensmodell wurde in Anlehnung an /Hofmeister et al. 99/, /Szyperski 98/, /Gomaa 00/, /Douglas 99/, /Bass et al. 98/ und /Bosch 00/ erstellt und beschreibt den Entwicklungsprozess der Architektur- und Entwurfsspezifikation eines softwareintensiven technischen Systems unter idealisierten Bedingungen.

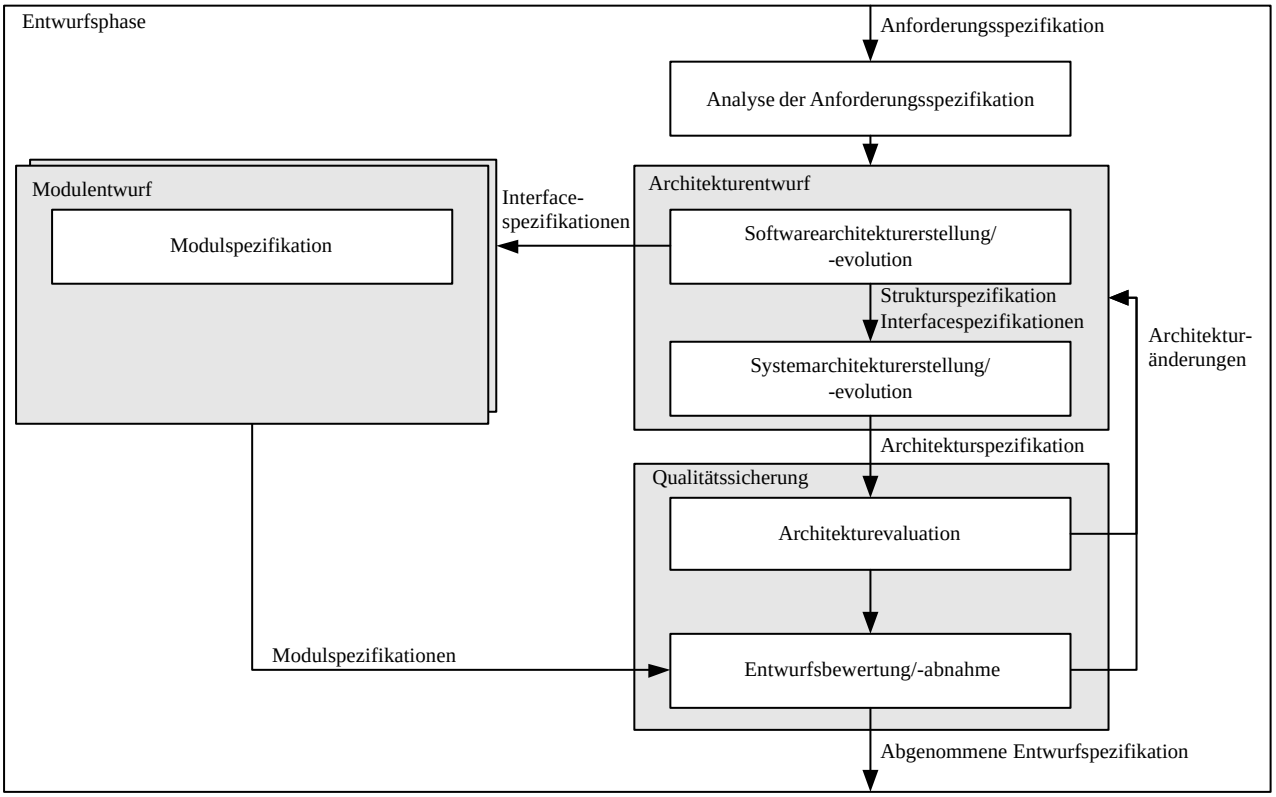


Abbildung 3-1 Idealisiertes Vorgehensmodell der Entwurfsphase

Zur besseren Verständlichkeit des Vorgehensmodells werden im Folgenden die einzelnen Teilphasen kurz vorgestellt und charakterisiert.

Analyse der Anforderungsspezifikation

In der ersten Teilphase des Entwurfs wird die Anforderungsspezifikation analysiert. Dadurch wird bei den beteiligten Softwarearchitekten ein gemeinsames Verständnis der einzelnen Anforderungen entwickelt. Darüber hinaus wird das Systemverständnis und das Verständnis der Schnittstellen zur Systemumgebung gefördert /Hofmeister et al 99/.

Werden bei der Analyse der Anforderungsspezifikation vergessene oder inkonsistente Anforderungen identifiziert, so ist eine Überarbeitung der Anforderungsspezifikation erforderlich. Daher ist es sinnvoll, die Analyse der Anforderungsspezifikation gemeinsam mit Entwicklern der Anforderungsspezifikation durchzuführen, um Rücksprachen zu erleichtern.

Softwarearchitekturerstellung

In der Softwarearchitekturerstellungsphase wird das zu entwickelnde Softwaresystem auf geeignete Weise in Softwarekomponenten dekomponiert. Das von außen beobachtbare Verhalten dieser Softwarekomponenten, also die Interaktionen der Komponente mit ihrer Umgebung, werden durch Interfacespezifikationen beschrieben /De Alfaro, Henzinger 01a/. Für die vollständige Spezifikation einer Softwarekomponente ist zudem eine Modulspezifikation notwendig /Szyperski 98/, welche das Verhalten einer Softwarekomponente detailliert und kompilierbar beschreibt. Diese Modulspezifikation wird im Modulentwurf erstellt.

Zusätzlich zu den Interfacespezifikationen ist für die Softwarearchitekturspezifikation auch eine Strukturspezifikation erforderlich. Diese Strukturspezifikation modelliert die Beziehungen zwischen den einzelnen Softwarekomponenten. Grundsätzlich wird dabei zwischen zwei Beziehungen unterschieden /Luckham et al. 95/:

- Kommunikationsbeziehungen
- Kompositionsbeziehungen

Durch Kommunikationsbeziehungen wird beschrieben, welche Komponenten im realisierten System miteinander interagieren. Durch Kompositionsbeziehungen hingegen wird dargestellt, auf welche Weise das System oder eine Komponente hierarchisch strukturiert ist, also wie eine Komponente aus Teilkomponenten aufgebaut ist und wie das entsprechende Interface durch Teilinterfaces zusammengesetzt ist. In Abbildung 3-2 ist eine solche Strukturspezifikation schematisch dargestellt, wobei die Kommunikationsbeziehungen durch horizontale und Kompositionsbeziehungen durch vertikale Pfeile repräsentiert werden.

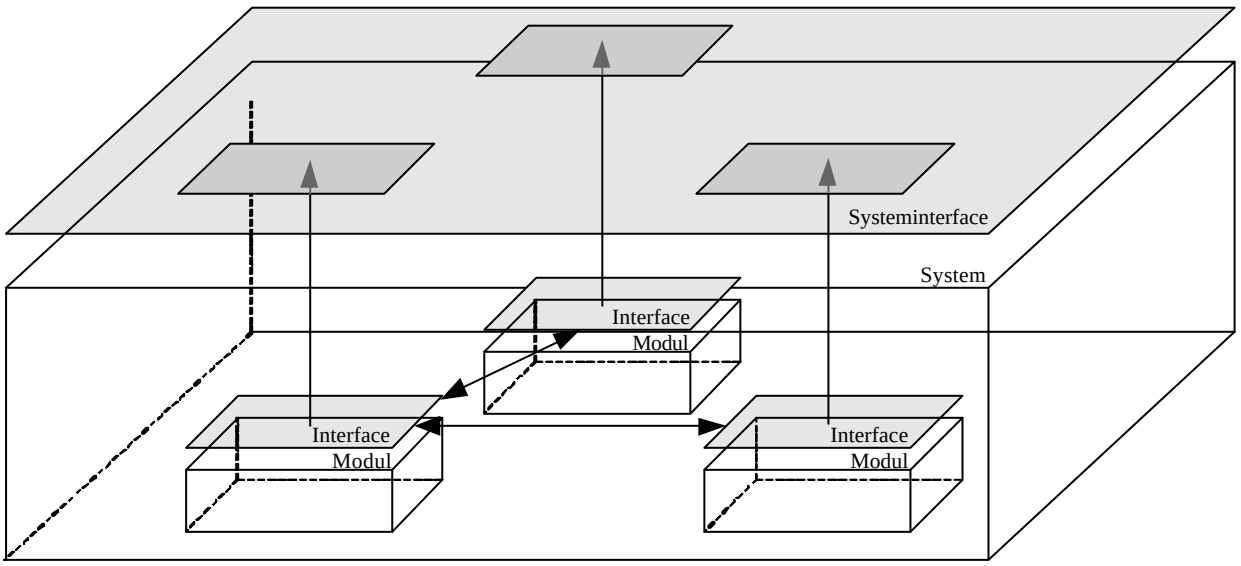


Abbildung 3-2 Einfache Strukturspezifikation

Für die Spezifikation einer Softwarearchitektur werden Architekturbeschreibungsnotationen verwendet, welche auch als ADL (*architecture description language*) bezeichnet werden /Bass et al 97/. Über die Zeit haben sich mehrere Architekturbeschreibungsnotationen etabliert. Diese fokussieren jeweils auf ein spezielles Anwendungsgebiet und sind daher oftmals nur bedingt für andere Anwendungsgebiete einsetzbar /Medvidovic, Taylor 00/. In Tabelle 3-1 wird ein Überblick über die geläufigen Architekturbeschreibungssprachen gegeben.

Tabelle 3-1 Überblick über die relevanten Architekturbeschreibungssprachen

Name	Anwendungsgebiet	Literatur
Aesop	Spezifikation von Architekturstilen	/Garlan et al. 95,96/
MetaH	Avionische Anwendungen (Flugsicherung, Navigation und Kontrolle)	/Vestal et al. 97/,/Binns et al. 96/
C2	Verteilte, dynamische und evolutionäre Systeme	/Medvidovic et al. 99/
Comm-Unity	Verteilte Systeme (Algebraische Spezifikation, Konnektoren)	/Wermelinger, Fiadeiro 99/, /Wermelinger et al. 01/, /Wermelinger, Fiadeiro 02/
UniCon	Beschreibung von Komponentenverschaltungen	/Shaw et al. 96/
Darvin	Formale Spezifikation von verteilten Systemen	/Magee et al. 95/
Wright	Modellierung und Analyse von nebenläufigen Systemen	/Allen, Garlan 97/
Rapide	Modellierung und Simulation von dynamischen Verhalten	/Luckham et al. 95/
SADL	Formale Beschreibung von Architekturverfeinerungen	/Moriconi et al .95/
ACME	Austauschformat zwischen den verschiedenen ADL	/Garlan et al. 97/

Während der Entwicklung des Systems ist die Softwarearchitekturspezifikation zumeist Änderungen unterworfen /Mens 99/. Diese werden notwendig, wenn sich die Anforderungen ändern oder durch die analytische Qualitätssicherung in den nachfolgenden Entwicklungsphasen Mängel in der Softwarearchitekturspezifikation erkannt werden. Die Veränderung der Softwarearchitekturspezifikation nach der Softwarearchitekturerstellungsphase wird als Softwarearchitekturevolution bezeichnet.

Systemarchitekturerstellung

Neben der Spezifikation von Softwarekomponenten und deren Zusammenwirken ist es für softwareintensive technische Systeme ebenfalls erforderlich, die benötigten Hardwarekomponenten zu berücksichtigen. Diese Hardwarekomponenten beeinflussen speziell Qualitätseigenschaften. Beispielweise wird die Echtzeitfähigkeit einer Softwarekomponente unter anderem von der Ausführungsgeschwindigkeit der ausführenden Hardwareplattform beeinflusst. Darüber hinaus hängt die Verfügbarkeit, Wartbarkeit oder Zuverlässigkeit des zu entwickelnden Systems von der Verfügbarkeit, Wartbarkeit oder Zuverlässigkeit der Sensoren, der Aktoren und der Hardwareplattform ab.

Während der Systemarchitekturerstellung ist es daher zunächst erforderlich, die genutzten Hardwareplattformen sowie die Sensoren und Aktoren auszuwählen. Anschließend werden die identifizierten Softwarekomponenten einer Hardwareplattform zugewiesen. Während dieser Phase muss darüber hinaus entschieden werden, ob die Softwarekomponenten auf einer einzigen Hardwareplattform oder verteilt ausgeführt werden. Bei einer verteilten Ausführung der Softwarekomponenten ist es zusätzlich notwendig, die Qualitätseigenschaften der Kommunikationskanäle zu betrachten.

Architekturevaluation

Die Architekturevaluation dient der analytischen Qualitätssicherung. Dabei wird geprüft, ob die Architekturspezifikation der Anforderungsspezifikation entspricht. Die Vorgehensweise der Prüfung unterscheidet sich dabei für funktionale Anforderungen und Qualitätsanforderungen.

Bei der Prüfung der funktionalen Anforderungen sind die Korrektheit und die Vollständigkeit der Architekturentwurfsspezifikation nachzuweisen. Dazu muss nach /Abadi, Lamport 91,95/ sichergestellt werden, dass die Softwarearchitekturspezifikation eine geeignete Verfeinerung der funktionalen Anforderungsspezifikation ist. Für den Nachweis werden die in der Strukturspezifikation enthaltenen Komponenten iterativ komponiert. Bei jedem Iterationsschritt werden dabei Komponenten aus der Strukturspezifikation ausgewählt, zwischen denen eine Kommunikationsbeziehung besteht. Für die ausgewählten Komponenten wird geprüft, ob ihre Interfaces kompatibel sind. Diese Prüfung wird als Kompatibilitätsanalyse bezeichnet und in Absatz 3.3.2 noch näher vorgestellt. Sind die ausgewählten Komponenten kompatibel, so wird durch Komposition eine neue Komponente erstellt. Die so entstandene Komponente ersetzt die einzelnen Komponenten in der Strukturspezifikation und kann in nachfolgenden Iterationsschritten erneut ausgewählt werden. Enthält die Strukturspezifikation nur noch eine Komponente, so wird die Interfacespezifikation

dieser Komponente gegen die funktionale Anforderungsspezifikation geprüft. Dies kann durch die Verfeinerungsprüfung (*refinement checking*) /Abadi, Lamport 91/, /Moriconi et al. 95/ erfolgen. Das theoretische Fundament für die Verfeinerungsprüfung ist in /De Alfaro, Henzinger 01a/ nachzulesen.
Die Prüfung der Architekturentwurfsspezifikation gegen die Qualitätsanforderungen wird in /Clements et al. 01/ als Softwarearchitekturanalyse (SAA) bezeichnet. In der vorliegenden Arbeit wird aber für eine bessere Abgrenzung zur Anforderungsanalysephase der Begriff Architekturevaluation benutzt. Für die einzelnen Qualitätsanforderungen werden dabei separate Evaluationstechniken verwendet. Diese Evaluationstechniken lassen sich in die folgenden Kategorien /Abowd et al. 97/ einordnen:
- Kompositionsbasierte Techniken
- Szenariobasierte Techniken
- Simulationsbasierte Techniken

Kompositionsbasierte Techniken entwickeln auf Basis der Strukturspezifikation und der Annahmen über die Qualitätseigenschaften der Komponenten dieser Strukturspezifikation quantitative Aussagen zu den Qualitätseigenschaften einer Softwarearchitektur. Beispiele für genutzte Annahmen sind die Wahrscheinlichkeit eines Fehlverhaltens oder die maximale Ausführungszeit einer Komponente. Ausgehend von diesen Annahmen werden in den Abschnitten 3.4-3.6 Techniken für die Evaluation von Sicherheits-, Wartbarkeits-, Zuverlässigkeits-, Verfügbarkeits- und Echtzeiteigenschaften dargestellt. Diese Aussagen der kompositionsbasierten Techniken sind korrekt, sofern die getroffenen Annahmen korrekt sind. Der Nachweis der Korrektheit der Annahmen muss daher in den nachfolgenden Phasen (beispielsweise in den Testphasen) erbracht werden. Sind die getroffenen Annahmen jedoch nicht korrekt, so ist es erforderlich, die Softwarearchitekturevaluation mit den neuen Annahmen zu wiederholen.
Szenariobasierte Techniken identifizieren anhand eines fiktiven Szenarios eine mögliche Interaktion mit dem zu entwickelnden System. Anschließend wird geprüft, ob durch die Softwarearchitektur dieses Szenario erwartungsgemäß behandelt wird /Hammer et al. 02/. Als allgemeine Methoden für die szenariobasierte Architekturevaluation haben sich SAAM (*Scenario-based Architecture Analysis Method*) und ATAM (*Architecture Tradeof Analysis Method*) etabliert /Clements 01/. Für die Bewertung von ökonomischen Gesichtspunkten wurde in /Asundi et al. 01/ die CBAM (*Cost Benefit Analysis Method*) als Erweiterung der SAAM und ATAM vorgeschlagen. Für die Analyse des Wartungsaufwands sollte ALMA (*Architecture-Level Modifiability Analysis*) /Bengtsson, Bosch 99/ verwendet werden.
Im Gegensatz zu den szenariobasierten und kompositionsbasierten Techniken arbeiten die simulationsbasierten Techniken nicht direkt mit der Softwarearchitektur, sondern erstellen daraus einen Prototyp des Systems. Mit diesem Prototyp werden Simulationen durchgeführt. Die experimentell gewonnenen Ergebnisse der Simulation werden anschließend verallgemeinert und bezüglich der Anforderungen geprüft /Abowd et al. 97/. Der Einsatz von simulationsbasierten Techniken wird dann erforderlich, wenn auf Basis der Softwarearchitekturspezifikation keine direkte Bewertung der Qualitätseigenschaften möglich ist.

Modulentwurf
Durch die Interfacespezifikation wird das extern beobachtbare Verhalten einer Komponente beschrieben. Für eine komplette Komponentenspezifikation sind jedoch noch weitere Aspekte, wie z. B. die Implementierung von Algorithmen und Datentransformationen, erforderlich. Daher ist im Modulentwurf die Interfacespezifikation durch eine Modulspezifikation zu erweitern. Diese Modulspezifikation stellt eine vollständige Beschreibung einer Komponente dar, welche nach der Kompilierung auf der Hardwareplattform ausführbar ist.

Entwurfsbewertung und Entwurfsabnahme
Die Entwurfsbewertungsphase dient der analytischen Qualitätssicherung der Entwurfsspezifikation. Dazu ist neben der Architekturevaluation auch die Konformität der Modulspezifikationen mit den Interfacespezifikationen zu prüfen. Dies kann ebenfalls durch eine Verfeinerungsprüfung /Abadi, Lamport 91/ erfolgen.
Entsprechen alle Modulspezifikationen ihren Interfacespezifikationen und wurde in der Architekturevaluation die Erfüllung der Anforderungen durch die Architekturspezifikation nachgewiesen, so kann die Entwurfspezifikation abgenommen werden.

3.2 Strukturspezifikation

Eine Strukturspezifikation beschreibt die Kompositions- und Kommunikationsbeziehungen zwischen den Komponenten. Zur Veranschaulichung der Konzepte einer Strukturspezifikation werden im Folgenden exemplarisch Komponenten- und Komponent-Konnektor-Modelle beschrieben. Als Komponenten-Modell wurde dazu mit dem ROOM-Struktur-Modell /Selic et al. 96/ ein in der Praxis verbreitetes Komponenten-Modell ausgewählt. Ähnliche komponentenbasierte Modelle werden in den ADL Rapide /Luckham et al. 95/, Darvin /Magee et al. 95/ und MetaH /Allen, Garlan 97/ verwendet. Die Komponent-Konnektor-Modelle werden am Beispiel des CCM /Hofmeister et al. 99/ erläutert. Sie werden aber auch in den ADL ACME /Garlan et al. 97/, C2 /Medvidovic et al. 99/ und UniCon /Shaw et al. 96/ eingesetzt.

3.2.1 Komponenten-Modelle

Die Grundlage der Komponenten-Modelle sind kommunizierende Aktoren (*actors*) /Agha, Kim 99/, welche nebenläufig agieren und die relevanten Informationen separat abspeichern. Im ROOM-Struktur-Modell werden diese Aktoren durch eine Kapsel /Selic et al. 94/ zu einer funktionalen und konzeptuellen Einheit zusammengefasst und die Einsicht in die interne Struktur verhindert.
Die Schnittstellen zur Umgebung einer Kapsel werden als Ports bezeichnet. Durch eine Verschaltung der Ports zweier Kapseln entsteht eine Kommunikationsbeziehung, über welche die Kapseln interagieren und Ereignisse austauschen. Werden die Ereignisse direkt an das Modul weitergeleitet, so wird der Port als End-Port bezeichnet.
Im ROOM-Struktur-Modell kann eine Kapsel in feingranularere Kapseln dekomponiert werden. Dadurch wird eine hierarchische Strukturierung der Architekturspezifikation ermöglicht. Zur Weiterleitung von Nachrichten an innere Komponenten werden mit den Relay-Ports spezielle Ports verwendet. Zusammenfassend ergibt sich das in Abbildung 3-3 dargestellte vereinfachte Meta-Modell einer ROOM-Strukturspezifikation.

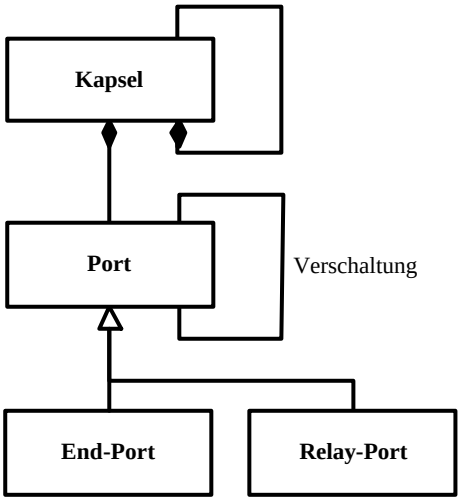


Abbildung 3-3 Vereinfachtes Meta-Modell des ROOM-Struktur-Modells

3.2.2 Komponent-Konnektor-Modelle

Komponent-Konnektor-Modelle führen wie in /Perry, Wolf 92/ und /Allen, Garlan 97/ gefordert mit dem Konnektoren ein weiteres Architekturelement ein. Durch dieses Architekturelement wird eine Separation von funktionalen Aspekten (Komponenten) und Kontroll- und Kommunikationsaspekten (Konnektoren) ermöglicht. Dies wirkt sich positiv auf die Verständlichkeit, Wartbarkeit, Flexibilität und Skalierbarkeit der Architektur aus /Garlan et al. 95/. Darüber hinaus wird die Wiederverwendbarkeit der einzelnen Komponenten gefördert /Medvidovic et al. 97/.

Ein Beispiel für ein Komponent-Konnektor-Modell ist das in /Hofmeister et al. 99/ vorgeschlagene CCM (component connector model), dessen Meta-Modell in Abbildung 3-4 dargestellt ist. In diesem Meta-Modell ist erkennbar, dass die Interfacespezifikationen zwischen Komponenten und Konnektoren differenziert werden. Einer Komponente wird zu diesem Zweck eine Menge von Ports und einem Konnektor eine Menge von Rollen zugeordnet. Des Weiteren ist erkennbar, dass ähnlich den Komponenten-Modellen eine hierarchische Dekomposition möglich ist. Dabei sind sowohl Komponenten als auch Konnektoren aus feingranulareren Komponenten und Konnektoren erstellbar.
Für die Beschreibung von Kommunikationsbeziehungen werden mit der Verschaltung und der Bindung zwei generelle Mechanismen verwendet. Die Verschaltung beschreibt eine Kommunikationsbeziehung zwischen einer Komponente und einem Konnektor. Dazu wird ein Port der Komponente mit einer Rolle des Konnektors verbunden.
Der Bindungsmechanismus wird bei einer hierarchischen Verschachtelung verwendet. Dabei wird ein Port einer inneren Komponente mit einem Port der umschließenden Komponente verbunden. Entsprechend kann der Bindungsmechanismus auch bei hierarchisch geschachtelten Konnektoren angewendet werden, in dem eine Rolle eines inneren Konnektors mit einer Rolle eines umschließenden Konnektors verknüpft wird.

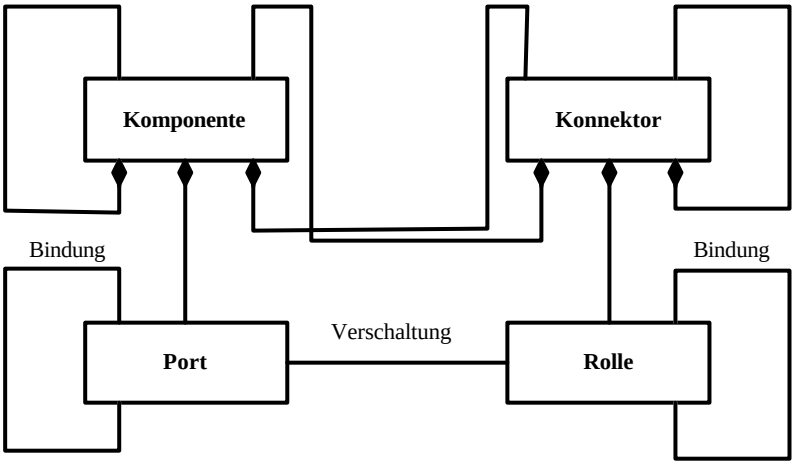


Abbildung 3-4 Metamodell eines Komponent-Konnektor-Modells nach /Hofmeister et al. 99/

3.3 Interfacespezifikation

Eine Interfacespezifikation beschreibt die Interaktionen einer Softwarekomponente mit ihrer Umgebung /Nierstrasz 93/. Diese Interaktion erfolgt über Ereignisse, welche das Auftreten von Nachrichten und Signalen beschreiben /Selic et al. 94/. In einer Interfacespezifikation wird zwischen Eingabeereignissen und Ausgabeereignissen unterschieden. Ein Eingabeereignis wird von der Umgebung erzeugt und von der Komponente beobachtet. Ein Ausgabeereignis dagegen wird von der Komponente erzeugt und an die Umgebung gesendet. In objektorientierten Programmiersprachen werden Eingabeereignisse durch Aufrufe von zur Verfügung gestellten Operationen und Ausgabeereignisse durch Aufrufe von Operationen fremder Objekte repräsentiert.
Zwischen den Ausgabe- und Eingabeereignissen bestehen kausale Abhängigkeiten. Das bedeutet, dass eine Komponente ein Ausgabeereignis nur dann erzeugt, wenn vorher bestimmte Eingabeereignisse von der Komponente beobachtet wurden. Die Gesamtheit aller kausalen Abhängigkeiten zwischen Eingabe- und Ausgabeereignissen wird als Verhaltensspezifikation oder Ausgabezusicherung (*output guarantee*) bezeichnet /Pratt 86/, /De Alfaro, Henzinger 01a/.
Ein Komponente verhält sich wie in der Verhaltensspezifikation beschrieben, wenn die als Eingabeprämissen *(input assumptions)* bezeichneten Beschränkungen bei der Benutzung der Komponente durch die Umgebung eingehalten werden /Abadi, Lamport 90/, /De Alfaro, Henzinger 01a/. Durch diese Beschränkungen wird sichergestellt, dass die Komponente die Ereignisse in korrekter Reihenfolge und mit korrektem Wertebereich der Parameter erhält. Eine korrekte Reihenfolge ist zum Beispiel immer dann von Bedeutung, wenn eine

Initialisierung einer Komponente vor ihrer Benutzung erforderlich ist. Als Eingabeprämisse in Bezug auf den korrekten Wertebereich sollte eine Komponente zur Berechnung von Quadratwurzeln nur mit positiven Zahlen versorgt werden. Diese Bedingungen müssen ebenfalls in die Interfacespezifikation integriert werden. Zusammengefasst ergibt sich aus der Verhaltensspezifikation und der Spezifikation der Beschränkungen der Eingabeereignisse das Protokoll einer Komponente /Hofmeister et al. 99/, /Selic et al. 94/.

3.3.1 Spezifikationsnotationen

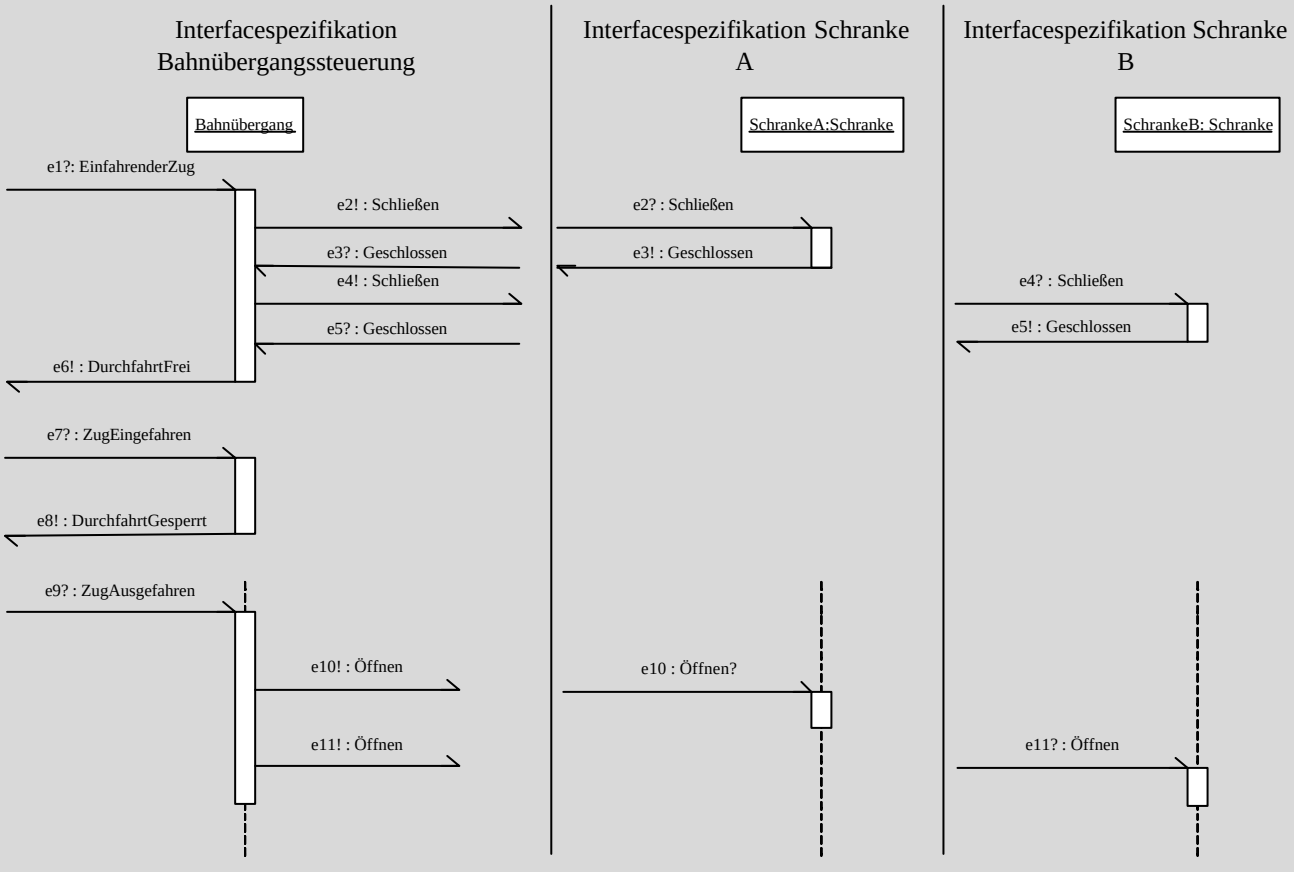
Zur Verdeutlichung der Vorgehensweise bei der Erstellung einer Interfacespezifikation werden im Folgenden mit den total-geordneten Ereignismodellen, den halb-geordneten Ereignismodellen, den Interfaceautomaten und den Prozessalgebren ausgewählte Spezifikationskonzepte dargestellt. Diese Konzepte werden in den bereits beschriebenen Architekturbeschreibungssprachen angewendet.

Total-geordnete Ereignismodelle

Ein total-geordnetes Ereignismodell spezifiziert das Interface einer Softwarekomponente durch einzelne lineare, exemplarische Ereignissequenzen (*Traces*). Innerhalb einer Ereignissequenz wird davon ausgegangen, dass ein Ereignis genau ein weiteres Ereignis auslöst. Total-geordnete Ereignismodelle eignen sich für die Spezifikation von sequenziellen Softwarearchitekturen /Lee 01/ und sind auf Grund ihrer Einfachheit in der Praxis sehr verbreitet /Booch et al. 99/. Als Spezifikationsnotation werden die bereits dargestellten Sequenzdiagramme verwendet.

Beispiel 3.1

Die Interfacespezifikation mit total-geordneten Ereignismodellen wird im Folgenden am bereits vorgestellten Beispiel eines eingleisigen Bahnübergangs veranschaulicht. Für die Interfaces von Bahnübergang, Schranke A und Schranke B sind exemplarische, total geordnete lineare Ereignissequenzen eingehender und ausgehender Nachrichten dargestellt. Die totale Ordnung der Ereignisse bzw. Nachrichten ergibt sich dabei aus ihrer Position auf der Lebenslinie der jeweiligen Komponente. Des Weiteren sind eingehende Nachrichten mit dem Postfix "?" und ausgehende mit dem Postfix "!" im Namen gekennzeichnet.



Halb-geordnete Ereignismodelle

Bei der Modellierung einer Interfacespezifikation in einem nebenläufigen oder reaktiven System ist die kausale Ordnung von zwei eintreffenden Ereignissen oft nicht bestimmbar /Andrews, Schneider 83/. Für die Spezifikation von Interfacespezifikationen mit solchen kausal unabhängigen Ereignissen werden in /Pratt 86/ *POMSET* (*partially ordered multi set of events*) vorgeschlagen, welche beispielsweise bei der ADL Rapide /Luckham et al 95/ angewandt werden. Ein *POMSET* spezifiziert die Interaktionen einer Komponente mit seiner Umgebung mit dem 4-Tupel $\langle V, \Sigma, \hat{o}, m \rangle$. Dabei ist V eine Menge von Knoten, Σ ist eine Menge von Nachrichten und Signalen, $\hat{o}$ ist eine partielle Ordnungsrelation über die Menge V und m ist eine Zuweisungsfunktion $m : V \to \Sigma$, welche jedem Knoten eine Nachricht oder ein Signal zuordnet. Durch die partielle Ordnungsrelation (Halbordnung) $\hat{o}$ werden die kausalen Abhängigkeiten zwischen den Knoten aus der Menge V spezifiziert. Die Ordnungsrelation muss für alle $x,y,z \in V$ die folgenden Eigenschaften besitzen:

- $x \hat{o} x$ (reflexiv)
- $x \hat{o} y$ und $y \hat{o} x$ impliziert x=y (antisymmetrisch)
- $x \hat{o} y$ und $y \hat{o} z$ impliziert $x \hat{o} z$ (transitiv)

Die Menge der Nachrichten und Signale Σ wird als Alphabet bezeichnet. Durch die Zuweisungsfunktion $m : V \to \Sigma$ wird jedem Knoten ein Element aus dem Alphabet zugeordnet. Die Knoten entsprechen somit Ereignissen. Durch die Zuweisungsfunktion ist es möglich, dass ein Element des Alphabets mehrfach in der Spezifikation auftritt, welches speziell in realen Systemen erforderlich ist.
Zur graphischen Darstellung eines *POMSET* eignet sich ein Hassediagramm. In diesem Diagramm werden die Elemente aus der Knotenmenge *V* als Punkte dargestellt. Zwei Knoten P_1 und P_2 sind dann voneinander kausal abhängig, wenn sie direkt oder transitiv durch eine Linie verbunden sind. Die Richtung der Ordnungsrelation wird durch die Position der Punkte im Diagramm bestimmt. Dabei gilt $P_1 \hat{o} P_2$, wenn der Punkt P_2 oberhalb des Punktes P_1 liegt.

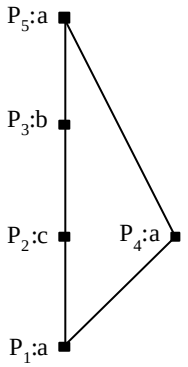


Abbildung 3-5 Hasse-Diagramm

In Abbildung 3-5 ist ein Hasse-Diagramm für das *POMSET* $\langle V_1, \Sigma_1, \hat{o}_1, m_1 \rangle$ dargestellt. In diesem *POMSET* ist:

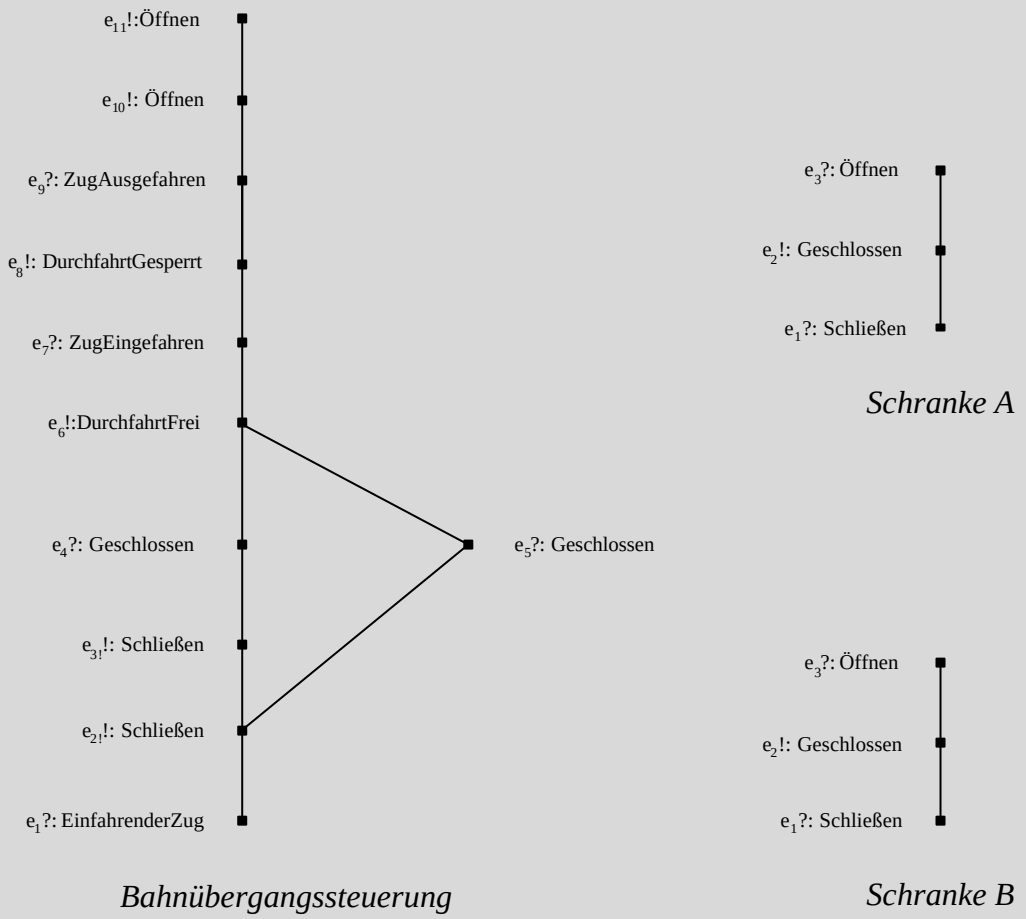
- $V_1 = \{P_1, P_2, P_3, P_4, P_5\}$
- $\Sigma_1 = \{a, b, c\}$
- $\hat{o}_1 = \{(P_1, P_2), (P_1, P_4), (P_2, P_3), (P_3, P_5), (P_4, P_5)\}$
- $m_1 = \{P_1 \to a, P_2 \to c, P_3 \to b, P_4 \to a, P_5 \to a\}$

Im Gegensatz zum linearen Ereignismodell beschreibt ein *POMSET* das Interface einer Komponente generisch /Pratt 86/. Das bedeutet, aus einer *POMSET*-Spezifikation lassen sich mehrere Ereignissequenzen erzeugen. Dies erfolgt durch Linearisierung /Luckham et al. 93/. Mit der Annahme, dass kausal unabhängige Ereignisse nicht gleichzeitig auftreten, würden sich aus dem *POMSET* $\langle V_1, \Sigma_1, \hat{o}_1, m_1 \rangle$ die Ereignissequenzen acbaa, acaba, aacba ergeben.

Beispiel 3.2

Die Interfacespezifikation des eingleisigen Bahnübergangsbeispiels mit POMSET kann durch das folgende Hassediagram charakterisiert werden. Die beiden Schranken werden parallel angesteuert. Daher können die

zwei Ereignisse e4? und e5? vom Typ "Geschlossen" in beliebiger Reihenfolge eintreffen, da sie nicht kausal voneinander abhängig sind.



Interfaceautomaten

Ein Interfaceautomat ist eine zustandsbasierte Spezifikation eines Komponenteninterfaces /Yellin, Strom 97/, welche nach /De Alfaro, Henzinger 01b/ durch ein 6-Tupel $P = \langle S, S^{init}, E^I, E^O, E^H, T \rangle$ wie folgt charakterisiert wird:

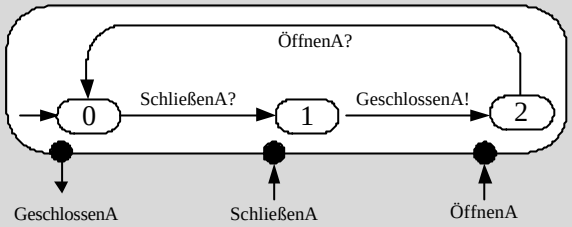
- S sind die möglichen Zustände des Automaten
- $S^{init} \subseteq S$ sind die Startzustände, mit $S^{init} \neq \emptyset$
- E^I sind die möglichen Eingabeereignisse, E^O sind die möglichen Ausgabeereignisse, E^H sind die möglichen internen Ereignisse, die Mengen sind disjunkt $E^I \cap E^O \cap E^H = \emptyset$ und ergeben zusammen die Menge der gesamten Ereignisse $E = E^I \cup E^O \cup E^H$
- $T \subseteq S \times E \times S$ beschreibt die Zustandsübergänge im Automaten; ein Element $\langle v, a, v' \rangle \in T$ definiert eine Reaktion auf ein Eingabeereignis, wenn $a \in E^I$ oder eine Ausgabereaktion, wenn $a \in E^O$ oder eine Reaktion auf ein internes Ereignis, wenn $a \in E^H$

Ein solcher Interfaceautomat kann in einem Zustand $v \in S$ nur bestimmte Eingabeereignisse $E^I(v)$ empfangen. Diese Menge wird durch die möglichen Zustandsübergänge T bestimmt. Das bedeutet, in Abhängigkeit vom Zustand werden Eingabeereignisse akzeptiert bzw. zurückgewiesen. Dadurch wird bei einem Interfaceautomaten die Reihenfolge der Eingabeereignisse beschränkt, wodurch die Spezifikation von Eingabeprämissen möglich ist.

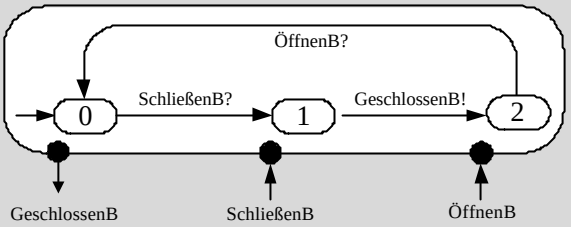
Des Weiteren kann ein deterministischer Interfaceautomat in einem Zustand $v \in S$ nur ein einziges Ausgabeereignis $e \in E^O(v)$ erzeugen. Da der Interfaceautomat durch eine Menge von Eingabeereignissen in den Zustand $v \in S$ gebracht wird, lassen sich somit die kausalen Abhängigkeiten zwischen den Eingabe- und Ausgabeereignissen spezifizieren. Damit eignet sich ein Interfaceautomat für die Verhaltensspezifikation der Komponente.

Beispiel 3.3

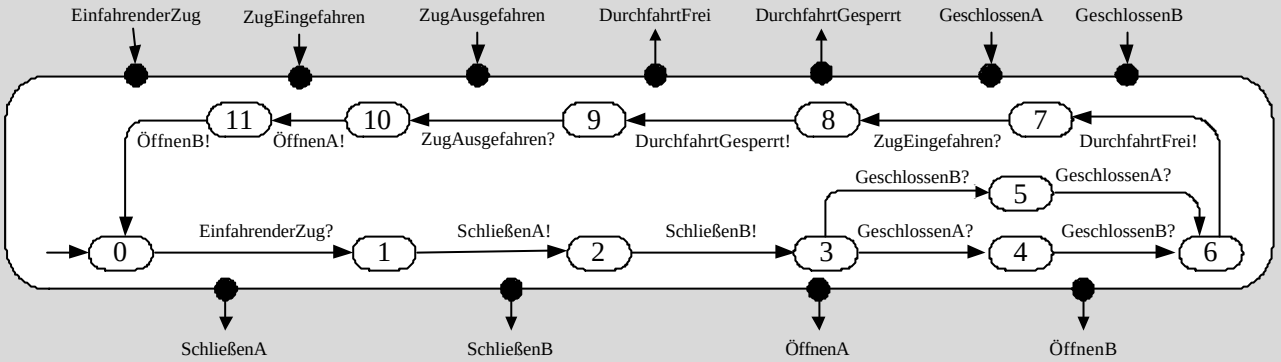
Die kausalen Abhängigkeiten zwischen Ein- und Ausgabeereignissen der Interfaces der Komponenten Schranke A, Schranke B und Bahnübergangssteuerung des bereits diskutierten Beispiels lassen sich durch Interfaceautomaten wie folgt beschreiben. Die verwendete Notation wurde dabei /De Alfaro, Henzinger 01b/ entnommen.



Interfacebeschreibung *Schranke A*



Interfacebeschreibung Schranke B



Interfacebeschreibung *Bahnübergangssteuerung*

Prozessalgebren

Eine Prozessalgebra spezifiziert das Interface einer Komponente durch einen nebenläufigen Prozess, welcher mit anderen Prozessen über Nachrichtenaustausch interagiert. Gebräuchliche Prozessalgebren sind CSP (*communicating sequential processes*) /Brookes et al. 84/ und das π Kalkül, welches auf der Theorie der CCS (*calculus of communicating systems* /Milner 89/) basiert. Im Folgenden werden die Konzepte der Prozessalgebren am Beispiel der CSP näher erläutert, da sie sich für die Architekturentwurfsspezifikation eignet /Allen, Garlan 97/ und unter anderem in der ADL Wright Anwendung findet.

In CSP werden synchron interagierende sequenzielle Prozesse spezifiziert. Synchron interagierend bedeutet, dass ein Ereignis nur gesendet werden kann, wenn gleichzeitig (synchron) ein anderer Prozess bereit zum Empfangen ist. Demnach erfolgt die Kommunikation in CSP über das Rendezvousprinzip /Hoare 85/. Des Weiteren wird angenommen, dass ein Prozess *P* nur durch eine vordefinierte Menge von Ereignissen *A* beeinflusst werden kann. Diese Menge *A* wird als Alphabet des Prozesses α *(P)=A* bezeichnet. Zusätzlich zum Alphabet der Prozesse existieren die Ereignisse *t* und $\surd$. Das Ereignis *t* repräsentiert eine interne Aktion eines Prozesses. Ein Prozess, der nur noch auf interne Ereignisse reagiert und mit seiner Umgebung nicht interagiert $\alpha(P)=\varnothing$ wird als STOP bezeichnet. Das Ereignis $\surd$ dient zur Modellierung der Terminierung eines Prozesses. Ein terminierender Prozess wird mit SKIP bezeichnet. Die CSP Syntax wird basierend auf den beiden Prozessen STOP und SKIP mit den in Tabelle 3-2 dargestellten Syntaxkonstrukten induktiv definiert.

Tabelle 3-2 CSP Syntaxkonstrukte

Name	Schreibweise	Bedeutung
Leerer Prozess	STOP	Spezifiziert einen Prozess, der mit seiner Umgebung nicht interagiert
Terminierender Prozess	SKIP	Spezifiziert einen Prozess, der nur noch auf das Ereignis $\surd$ reagiert

Ausgabeanweisung	$a! \rightarrow P$	Der Prozess erzeugt das Ausgabeereignis α und verhält sich anschließend wie in *P* spezifiziert.
Eingabeanweisung	$a? \rightarrow P$	Der Prozess empfängt das Eingabeereignis α und verhält sich anschließend wie in *P* spezifiziert.
Deterministische Auswahl	$P, \ Q$	Der Prozess verhält sich wie *P* oder *Q*. Die Entscheidung darüber ist von der Umgebung beeinflussbar. Dies kann beispielsweise über Wächterbedingungen /Dijkstra 75/ $a! \rightarrow P, \ b! \rightarrow Q$ erfolgen.
Nicht-deterministische Auswahl	$P \acute{o} Q$	Der Prozess verhält sich wie *P* oder *Q*. Die Entscheidung darüber ist nicht deterministisch, also von der Umgebung nicht beeinflussbar.
Sequenzielle Komposition	$P;Q$	Der Prozess führt zuerst die Spezifikation von *P* aus, und nachdem *P* das Ereignis √ erhält wird die Spezifikation von *Q* ausgeführt.
Parallele Komposition	$P \underset{A}{\|\|} Q$	Der Prozess verhält sich wie die parallele Ausführung der Prozesse *P* und *Q*. Die Ereignismenge A wird als Synchronisationsalphabet bezeichnet. Tritt ein Ereignis aus der Menge *A* auf, muss sowohl *P* als auch *Q* an der Kommunikation teilnehmen.
Interleaving	$P \ \|\|\| \ Q$	Der Prozess verhält sich wie die parallele Ausführung der Prozesse *P* und *Q*. Im Gegensatz zur parallelen Komposition muss jedoch nur ein Prozess an einer Interaktion teilnehmen.
Information Hiding	$P \setminus A$	Der Prozess verhält sich wie in *P* spezifiziert, jedoch sind die in der Menge *A* enthaltenen Ereignisse für die Umgebung nicht sichtbar und zugreifbar.

Beispiel 3.4

Die CSP-Prozessspezifikation für das Bahnübergangsbeispiel kann wie folgt angegeben werden:

Bahnübergang:=Bahnübergangssteuerung ||| SchrankeA ||| SchrankeB

Bahnübergangssteuerung:=LCC
LCC :=EinfahrenderZug?→CLC
CLC :=SchließenA! →SchließenB! →WFCLC
WFCLC :=(GeschlossenA?→ GeschlossenB?)||| (GeschlossenB?→ GeschlossenA?)→ WTLT
WTLT := DurchfahrtFrei! →ZugEingefahren? →DurchfahrtGesperrt!→ZugAusgefahren? →OLC
OLC :=ÖffnenA!→ÖffnenB! →LCC

SchrankeA:== LLC
LLC :=SchließenA? →GeschlossenA! →ÖffnenA? → LLC

SchrankeB:==RLC
RLC :=SchließenB? →GeschlossenB! →ÖffnenB? →RLC

Die Semantik von CSP kann durch das Trace-Modell und das Failures-Divergenz-Modell /Brookes et al. 84/ beschrieben werden. Das Trace-Modell beschreibt einen CSP-Prozess durch die Menge aller bildbaren Ereignissequenzen (*traces*) T*(P)* dieses Prozesses. Eine einzelne Ereignissequenz dieser Menge wird wie folgt notiert:

$$trace(P) = \langle a_1, a_2, ...a_n \rangle \ \ a_i \in \left(a(P) \cup \{\ddot{O}\} \right) \wedge a_n = \ddot{O}$$

Für die Ermittlung von T*(P)* wird aus der Prozessspezifikation ein Zustands-Transitions-Automat $A(P) = \langle S, s_o, E, \rightarrow \rangle$ gebildet. In diesem Automaten ist *S* die Menge der möglichen Zustände, s_0 der Startzustand, *E* die Menge der möglichen Ereignisse und $\rightarrow \subseteq S \times \{E \cup \{Ö, t\}\} \times S$ ist eine Zustandsübergangsfunktion. Für die Bildung von *A(P)* werden in /Brookes et al. 84/ induktiv Regeln für die CSP-Syntaxkonstrukte gegeben. Aus dem Zustands-Transitions-Graphen *A(P)* wird die bildbare Sprache *L(P)* erzeugt, welche der Menge aller bildbaren Ereignissequenzen T*(P)* entspricht. Die Ermittlung von T*(P)* lässt sich auch direkt, ohne Umweg über den Automaten *A(P)*, aus der CSP- Spezifikation ermitteln. Dazu wird ebenfalls induktiv vorgegangen, indem zunächst die Ereignissequenzmengen für die einfachen Prozesse STOP und SKIP bestimmt werden.

$$T(STOP) = \{\varnothing\}$$

$$T(SKIP) = \{\varnothing, \surd\}$$

Die Ereignissequenzmenge für die Eingabe- und Ausgabeanweisung ergibt sich aus der leeren Menge und der Konkatenation des Ereignisses *a*! oder *a*? und T*(P)*

$$T(a? \rightarrow P) = \{\varnothing\} \cup \{a?\}^{o} T(P)$$

$$T(a! \rightarrow P) = \{\varnothing\} \cup \{a!\}^{o} T(P)$$

Für alle zweistelligen CSP- Operatoren *op* existieren Operatoren op_{trace}, für die gilt:

$$T(P op Q) = T(P) op_{traces} T(Q)$$

Für die Beschreibung der einzelnen Operatoren op_{trace} wird auf /Brookes et al. 84/ verwiesen.
Auf Basis der Ereignissequenzen lassen sich die kausalen Abhängigkeiten zwischen den Ereignissen spezifizieren. Ein Ausgabeereignis a_j ist dabei von allen Vorgängerereignissen $a_{i<j}$ in einer Ereignissequenz kausal abhängig. Somit eignen sich CSP-Prozesse für die Verhaltensspezifikation von Komponenten. Für die Spezifikation der korrekten Reihenfolge der Eingabeereignisse kann mit Hilfe des Graphen *A(P)* diejenige Teilmenge des Alphabets ermittelt werden, welche in einem Zustand akzeptiert wird. Die Menge der nicht akzeptierten Ereignisse in einem Zustand wird als Refusalmenge *R(P)* bezeichnet. Zusammen mit der Zustandsübergangsfunktion $\rightarrow$ kann somit auch die Reihenfolge der Eingabeereignisse spezifiziert werden.
Eine weitere semantische Fundierung von CSP besteht in dem Failures-Divergenz-Modell. Dieses beschreibt einen CSP-Prozess *P* durch das Tupel $\langle a(P), F(P), D(P) \rangle$. Dabei entspricht $a(P)$ dem Alphabet des Prozesses *P*. Durch Φ*(P)* wird die Menge der Ereignissequenzen charakterisiert, welche durch einen Prozess zurückgewiesen wird. Sie wird aus dem Kreuzprodukt aus der Menge der Ereignissequenzen und der dazugehörigen Refusalmenge gebildet:

$$F(P) = T(P) \times R(P)$$

Die Menge Δ *(P)* beschreibt dagegen die Ereignissequenzen, die zu einer endlosen Folge von internen Ereignissen führen. Vorteile besitzt das Failures-Divergenz-Modell bei der Prüfung von Eigenschaften des Prozesses. So kann bestimmt werden, ob die Protokolle zweier Prozesse *P* und *Q* kompatibel sind. Dies wird in Abschnitt 3.3.2 noch näher erläutert. Darüber hinaus kann geprüft werden, ob ein Prozess *P* eine korrekte Verfeinerung eines Prozesses *Q* ist. Erforderlich ist dies beispielweise bei der Korrektheitsprüfung einer Modulspezifikation gegen seine Interfacespezifikation. Für Prozesse mit endlich vielen Zuständen sind diese Eigenschaften mit dem Werkzeug FDR (*failures divergence refinement*) nachzuweisen /Roscoe 95/.

3.3.2 Kompatibilitätsanalyse

Zwei Komponenten mit den Interfacespezifikationen *A* und *B* können verschaltet werden, wenn ihre Protokolle kompatibel sind. Das bedeutet, beide Interfacespezifikationen halten die Eingabeprämissen der jeweils anderen ein. Für den Nachweis der Kompatibilität wird üblicherweise von beiden Interfacespezifikationen die Menge aller bildbaren Ereignissequenzen generiert /Chakrabarti et al. 02/. Anschließend ist für jede Ereignissequenz einzeln zu prüfen, ob die Eingabeprämissen der jeweils anderen Interfacespezifikation verletzt werden.

Bei der Failures-Divergenz-Semantik von CSP-Prozessen /Brookes et al. 84/ kann zusätzlich zur der Einhaltung der Eingabeprämissen auch die Freiheit von Verklemmungen nachgewiesen werden. Dazu ist zu prüfen, ob keine Ereignissequenz des einen Prozesses von dem anderen Prozess zurückgewiesen wird oder diesen in einen Divergenzzustand bringt. Formal lässt sich dies durch die folgenden beiden Bedingungen ausdrücken:

$$\forall t_A \in T\ (P), f_B \in F\ (Q), d_B \in D: (Q): t_A \neq f_B \wedge t_A \neq d_B$$
$$\forall t_B \in T\ (Q), f_A \in F\ (P), d_A \in D: (P): t_B \neq f_A \wedge t_B \neq d_A$$

Die Kompatibilitätsanalyse anhand von Ereignissequenzen ist jedoch vor allem bei komplexen Spezifikationen mit großem Aufwand verbunden. Darüber hinaus ist sie nur bei Interfacespezifikationen anwendbar, welche eine endliche Menge von Ereignissequenzen beschreiben. Für Interfacespezifikationen mit komplexen oder unendlichen bildbaren Mengen von Ereignissequenzen ist es daher erforderlich, die Protokollkompatibilität auf Basis der generischen Interfacespezifikationen nachzuweisen. Im Folgenden wird dazu ein für Interfaceautomaten anwendbares Verfahren vorgestellt. Diese Vorgehensweise lässt sich mit geringen Modifikationen auch auf *POMSET*- und *CSP*- Spezifikationen übertragen, da sich diese auf Zustandstransitionsautomaten abbilden lassen /Milner 89/, /Brookes et al. 84/, /Pratt 86/.
Für die Kompatibilitätsanalyse müssen jeweils zwei Interfaceautomaten *A* und *B* komponiert und die Ausgabeereignisse eines Automaten auf die Eingabeereignisse des anderen Automaten abgebildet werden. Die so zwischen den beiden Automaten ausgetauschten Ereignisse werden zur Menge $shared(A,B)$ vereinigt, welche wie folgt bestimmt wird:

$$shared(A,B) = \left(E_A^I \cap E_B^O\right) \cup \left(E_B^I \cap E_A^O\right)$$

Mit Hilfe der Menge $shared(A,B)$ lässt sich ein Produktautomat $A \otimes B$ der beiden Interfaceautomaten *A* und *B* wie folgt bilden:

$$S_{A\otimes B} = S_A \times S_B$$
$$S_{A\otimes B}^{init} = S_A^{init} \times S_B^{init}$$
$$E_{A\otimes B}^I = E_A^I \cup E_B^I \setminus shared(A,B)$$
$$E_{A\otimes B}^O = E_A^O \cup E_B^O \setminus shared(A,B)$$
$$E_{A\otimes B}^H = E_A^H \cup E_B^H \cup shared(A,B))$$
$$T_{A\otimes B} = \begin{aligned} &\left\{\left((v,u),a,(v',u)\right) \mid (v,a,v') \in T_A \wedge a \notin shared(A,B) \wedge u \in S_B\right\} \cup \\ &\left\{\left((v,u),a,(v,u')\right) \mid (u,a,u') \in T_B \wedge a \notin shared(A,B) \wedge v \in S_A\right\} \cup \\ &\left\{\left((v,u),a,(v',u')\right) \mid (v,a,v') \in T_A \wedge (u,a,u') \in T_B \wedge a \in shared(A,B)\right\} \end{aligned}$$

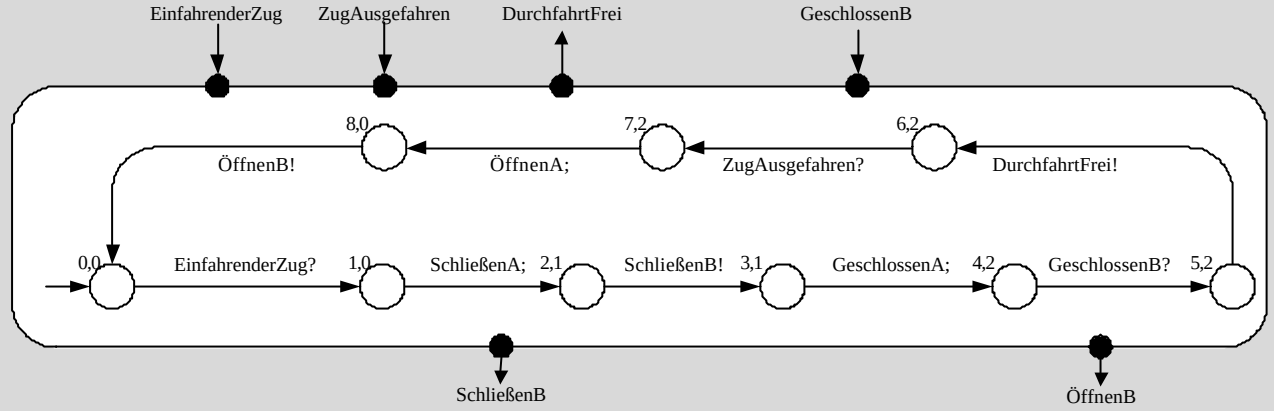
Durch den gebildeten Produktautomaten wird das Verhalten einer Komponente beschrieben, welche aus zwei Komponenten mit den Interfaceautomaten *A* und *B* komponiert ist /De Alfaro, Henzinger 01b/. Als Voraussetzung müssen jedoch alle nicht in $shared(A,B)$ enthaltenen Ereignisse zwischen beiden Automaten disjunkt sein, da dies für die deterministische Identifizierung der Zustandsübergänge im Produktautomaten erforderlich ist.
Sind die Protokolle der Interfaceautomaten *A* und *B* inkompatibel, so existieren im Produktautomaten Fehlerzustände, in denen der Automat keine weiteren Eingabeereignisse akzeptiert. Diese Zustände werden durch die Menge $illegal(A,B)$ identifiziert, welche wie folgt bestimmt wird:

$$illegal(A,B) = \left\{(v,u) \in S_A \times S_B \mid \exists a \in shared(A,B).\left(a \in E_A^O(v) \wedge a \notin E_B^I(u) \vee a \in E_B^O(u) \wedge a \notin E_A^I(v)\right)\right\}$$

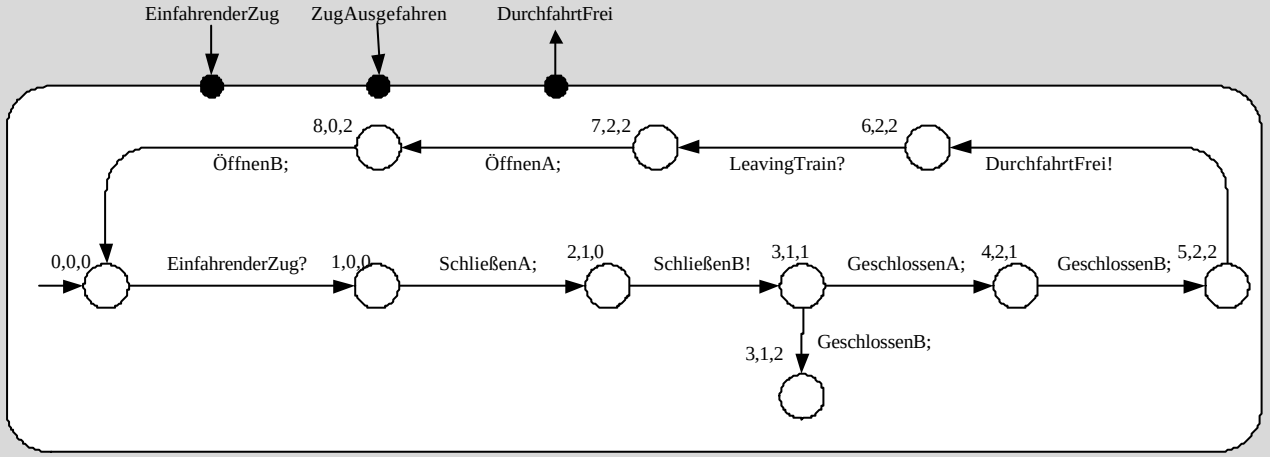
Diese Fehlerzustände sollten bei einer Komposition vermieden werden. Sind sie jedoch unumgänglich, so ist es erforderlich, die Umgebung der Komponente so zu beschränken, dass diese Fehlerzustände nicht erreicht werden. Dies kann über zusätzliche Eingabeprämissen erfolgen.

Beispiel 3.5

Zur Veranschaulichung der Kompatibilitätsanalyse mit Interfaceautomaten werden die im Beispiel 3.3 dargestellten Interfaceautomaten des Bahnübergangsbeispiels verwendet. Dabei wird jedoch die Spezifikation der Bahnübergangssteuerung durch Wegnahme des Zustands 5 verändert. Dadurch ergibt sich im Kompositionsautomaten ein Fehler im Zustand (3,1,1), da dieser auf die Eingabe GeschlossenB? nicht reagieren kann und somit in den Zustand (3,1,2) läuft.



Bahnübergangssteuerung ↳ SchrankeA



(Bahnübergangssteuerung ↳ SchrankeA) ↳ SchrankeB

3.3.3 Integration von Echtzeit-Eigenschaften

Für die Spezifikation von Annahmen bezüglich der Echtzeiteigenschaften einer Interfacespezifikation sind neben den kausalen Abhängigkeiten auch die temporalen Abhängigkeiten zu beachten. Dafür müssen die Auftrittszeitpunkte der Ereignisse zusätzlich mit Zeitbedingungen beschränkt werden, welche die Echtzeitanforderungen der Anforderungsspezifikation korrekt verfeinern. Für die Spezifikation dieser Zeitbedingungen wurden einige der vorgestellten Interfacespezifikationskonzepte auf geeignete Weise erweitert. Diese Erweiterungen werden im Folgenden vorgestellt.

Für die Spezifikation von Echtzeiteigenschaften in einem *POMSET* werden in /Luckham, Vera 93/ zeitannotierte *POMSET* (*timed POMSET*) vorgeschlagen. Diese weisen jedem Ereignis einen Auftrittszeitpunkt zu. Dadurch ist neben der kausalen Ordnung auch die temporale Ordnung bestimmbar. Die Spezifikation von Zeitbedingungen kann z. B. über das in Rapide /Lukham et al. 93,95/ verwendete Ereignismuster *during(n,m)* erfolgen, bei dem die Auftrittszeitpunkte eines Ereignisses mit der oberen Schranke *n* und der unteren Schranke *m* beschränkt werden.

Interfaceautomaten werden in /de Alfaro et al. 02/ mit den zeitannotierten Interfaceautomaten (*timed interfaceautomata*) für die Spezifikation von Echtzeitbedingungen erweitert. Sie bieten die Möglichkeit, den Zeit-

punkt eines Zustandsüberganges durch eine Menge von Zeitgebern C zu beschränken. Die Spezifikation der Zeitbedingung erfolgt ähnlich den in Abschnitt 2.3 beschriebenen zeitannotierten Automaten /Alur, Dill 94/.
Zeitannotierte CSP (*timed CSP*) /Reed, Roscoe 88/, /Schneider 95/ führen mit $WAIT\ t$ und $a^t \rightarrow P$ zwei weitere Syntaxkonstrukte ein. Dabei repräsentiert $WAIT\ t$ einen Prozess, welcher t Zeiteinheiten wartet und anschließend terminiert. Dadurch kann die untere Schranke einer Zeitbedingung wie folgt spezifiziert werden: $WAIT\ t; P$. Beim Syntaxkonstrukt $a^t \rightarrow P$ bietet der Prozess das Ereignis a für t Zeiteinheiten an und verhält sich anschließend wie in P spezifiziert. Dadurch lässt sich auch die obere Schranke einer Zeitbedingung spezifizieren. Für eine gemeinsame Spezifikation von unterer und oberer Schranke einer Zeitbedingung wird in /Pardo et al. 01/ das Syntaxkonstrukt $a^{\langle t_1, t_2 \rangle} \rightarrow P$ vorgeschlagen. Dabei entspricht $\langle t_1, t_2 \rangle$ dem Intervall, in dem der Prozess das Ereignis zur Verfügung stellt.

3.3.4 Integration von Wartbarkeits-, Zuverlässigkeits- und Verfügbarkeitseigenschaften

Für die Spezifikation von Wartbarkeits-, Zuverlässigkeits- und Verfügbarkeitsannahmen in Interfacespezifikationen ist es erforderlich, das Verhalten in Abhängigkeit von stochastischen Attributen zu beschreiben. In CSP kann dies beispielsweise durch das in /Lowe 95/ vorgeschlagene, erweiterte Syntaxkonstrukt für die nicht-deterministische Auswahl $P\ {}_p\ddot{o}\ {}_q Q$ spezifiziert werden. In diesem Konstrukt beschreibt p die Wahrscheinlichkeit, mit der sich der Prozess wie in P spezifiziert und q die Wahrscheinlichkeit, mit der sich der Prozess wie in Q spezifiziert verhält.
Für die Spezifikation von Zuverlässigkeitsannahmen entspricht P beispielsweise der Spezifikation des korrekten Verhaltens und Q der Spezifikation des Verhaltens nach einem Ausfall. Somit ist q die Ausfallwahrscheinlichkeit. Für die Spezifikation von Wartbarkeitsannahmen werden P und Q vertauscht. Somit charakterisiert q die Reparaturwahrscheinlichkeit.
Für die Erweiterung von Interfaceautomaten werden probabilistische Zustandsübergänge eingeführt. Diese Zustandsübergänge treten in Abhängigkeit von einer spezifizierten Wahrscheinlichkeit $p \in P$ auf. Ein probabilistischer Übergang von einem Zustand $s \in S$ in den Zustand $s' \in S$ wird durch die Transitionsrelation T wie folgt spezifiziert:
$T \subseteq S \times E \times S \times P$

3.3.5 Integration von Sicherheitseigenschaften

Für die Integration von Sicherheitsannahmen in Interfacespezifikationen sind diejenigen Fehlverhalten einer Komponente zu beschreiben, welche zu Gefährdungen führen. Diese Fehlverhalten lassen sich nach /Bondavalli, Simoncini 90/ und /McDermid, Pumfrey 94/ den folgenden Kategorien zuordnen:
- Ein Ereignis erfolgt außerhalb des spezifizierten Zeitfensters (zu früh oder zu spät)
- Ein Ereignis wird unterlassen
- Die Ereignisse treten in einer fehlerhaften Reihenfolge auf
- Ein Ereignis wird erzeugt, aber nicht gefordert

Für die Spezifikation der Fehlverhalten der ersten beiden Kategorien werden die in Abschnitt 3.3.3 und 3.3.4 dargestellten Spezifikationstechniken verwendet. Für die Spezifikation der Fehlverhalten der letzten beiden Kategorien muss die Interfacespezifikation um Ereignissequenzen erweitert werden, welche die Komponente nicht offenbaren darf. Dies kann zum Beispiel über die direkte Beschreibung der Ereignissequenzen oder über die Markierung von fehlerhaften Zuständen oder Zustandsübergängen erfolgen /Leveson 95/.
Jedes dieser Fehlverhalten muss zusätzlich mit Annahmen über dessen Auftrittswahrscheinlichkeit annotiert werden.

3.4 Nachweis von Sicherheitseigenschaften

Zum Nachweis der Sicherheitseigenschaften eines softwareintensiven technischen Systems werden Gefährdungsanalysetechniken verwendet /Liggesmeyer 00/. Diese Gefährdungsanalysetechniken lassen sich auch in der Architekturentwurfsphase zur Bestimmung der Sicherheitseigenschaften einer Architektur einsetzen. Dazu muss überprüft werden, ob ein nach der Architekturspezifikation erstelltes System die in der Risiko-

analyse ermittelten Sicherheitsanforderungen erfüllt. Zu diesem Zweck müssen die Fehlverhalten der Architekturelemente identifiziert werden, welche zu Gefährdungen führen. Darüber hinaus werden die kausalen Zusammenhänge zwischen den Fehlverhalten und den Gefährdungen analysiert. Ausgehend von Annahmen über die Auftrittswahrscheinlichkeiten der Fehlverhalten kann somit innerhalb der Gefährdungsanalyse die Wahrscheinlichkeit einer Gefährdung ermittelt und die Erfüllung der Sicherheitsanforderungen geprüft werden.
Oft werden in der Gefährdungsanalyse allerdings bisher unerkannte Gefährdungen identifiziert. Diese bilden dann eine zusätzliche Eingabe für eine neue Risikoanalyse, bei der die Auswirkungen solcher Gefährdungen geprüft werden. Stellen sich diese als relevant für das System heraus, werden sie in die Sicherheitsanforderungen aufgenommen. Dadurch entsteht ein zyklischer Prozess zwischen Gefährdungs- und Risikoanalyse, welcher sich dann stabilisiert, wenn die Sicherheitsanforderungen hinreichend beschrieben sind und in der Gefährdungsanalyse keine weiteren Gefährdungen identifiziert werden.
Die einzelnen Aktivitäten der Gefährdungsanalyse und der Zusammenhang zwischen Gefährdungsanalyse und Risikoanalyse sind im Sanduhrmodell aus der /EN 50129/ und /Braband 98/ erkennbar. Dieses Modell ist in Abbildung 3-6 dargestellt.

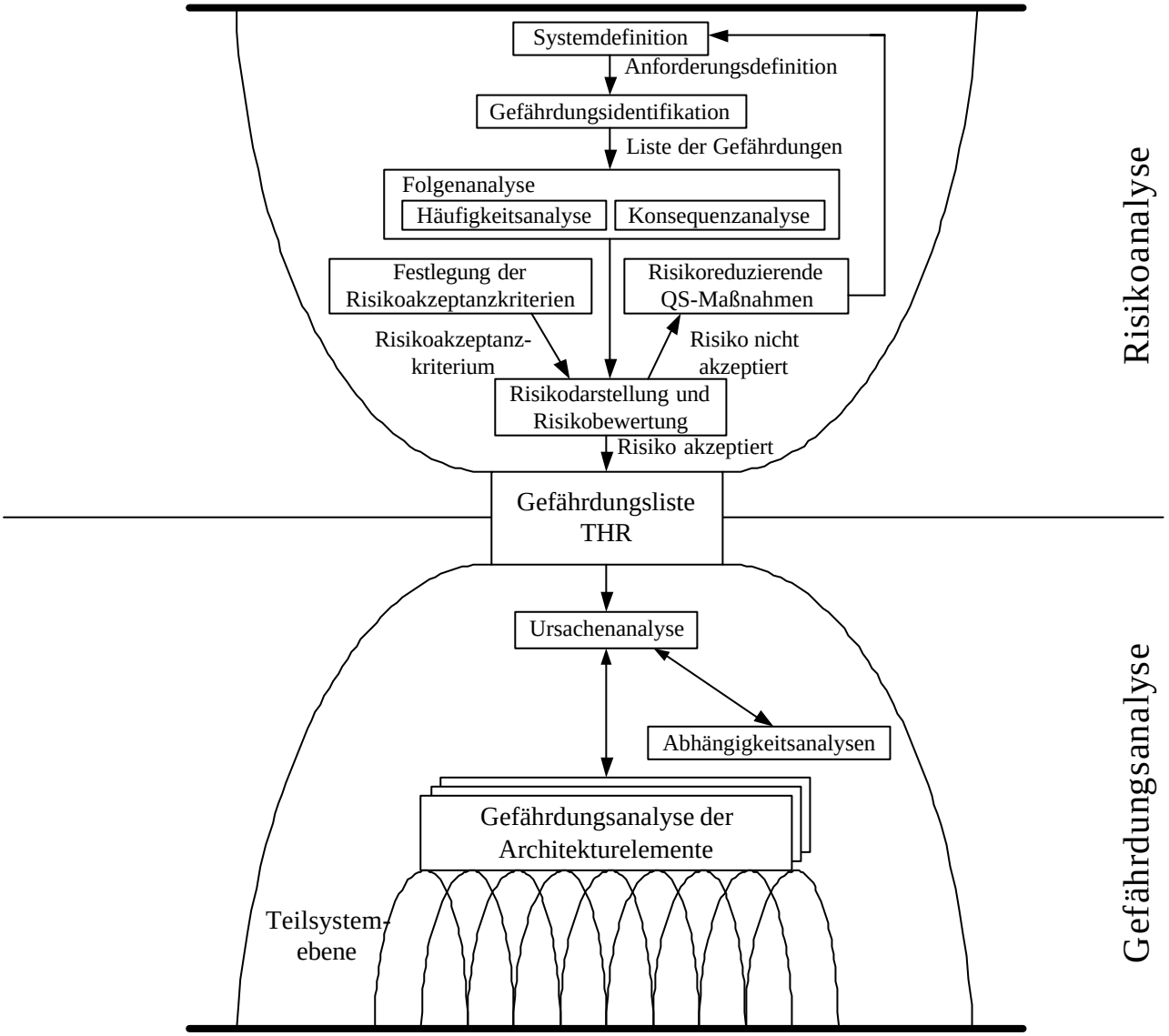


Abbildung 3-6 Zusammenhang zwischen Gefährdungs- und Risikoanalyse

Die einzelnen Schritte der Gefährdungsanalyse werden im Folgenden zum besseren Verständnis kurz beschrieben.

Ursachenanalyse
Innerhalb der Ursachenanalyse werden für jede Gefährdung die relevanten Fehlverhalten der Architekturelemente und deren Einfluss auf die Gefährdung identifiziert. Oft sind für eine Gefährdung jedoch mehrere Fehlverhalten verantwortlich. Daher empfiehlt es sich, den Prozess der Analyse in zwei Teilschritten zu vollziehen. Zum einen sollte eine qualitative Ursachenanalyse durchgeführt werden, bei der die Wirkzusammenhänge zwischen den Fehlverhalten der Architekturelemente und den Systemgefährdungen identifiziert werden. Zum anderen werden ausgehend von diesen kausalen Abhängigkeiten und den Annahmen über die Wahrscheinlichkeiten der relevanten Fehlverhalten – die Wahrscheinlichkeiten der Gefährdungen ermittelt. Dieser Schritt kennzeichnet den quantitativen Teil der Ursachenanalyse.

Abhängigkeitsanalyse
Durch die Abhängigkeitsanalyse werden diejenigen Fehlverhalten identifiziert, welchen eine gemeinsame Ursache zugrunde liegt. Diese Ursachen werden als CCF (common-cause-failure) bezeichnet und müssen bei der quantitativen Ursachenanalyse gesondert berücksichtigt werden /Mauri 00/.

Beispiel 3.6

Ein Beispiel für einen CCF ist ein Fehlverhalten der Hardwareplattform, z. B. aufgrund einer Unterbrechung der Stromzufuhr. Dieses führt zu einem Fehlverhalten aller Softwarekomponenten, die auf der Hardwareplattform ablaufen.

In /Mauri 00/ und /Pumfrey 99/ werden mit der ZSA (*zonal safety analysis*), der PRA (*particular risk analysis*) und der CMA (*common mode analysis*) spezielle Techniken der Identifizierung derartiger Fehlverhalten benannt und dokumentiert. Vorrangig wird jedoch bei der Abhängigkeitsanalyse auf Erfahrungswissen zurückgegriffen. Dieses ist für zahlreiche Anwendungsgebiete bereits in zusammengestellten Checklisten verfügbar /SAE-APR 4761/.

Gefährdungsanalyse der Architekturelemente
Für die einzelnen Architekturelemente werden darüber hinaus weitere Gefährdungsanalysen durchgeführt. Diese erfolgen rekursiv entsprechend den Kompositionsbeziehungen der Strukturspezifikation, bis die Architekturelemente nicht mehr weiter dekomponierbar sind. Durch diese Vorgehensweise werden die elementaren Ursachen für die relevanten Fehlverhalten identifiziert. Ausgehend von den Annahmen über die Auftrittswahrscheinlichkeiten aller elementaren Ursachen werden dann die Auftrittswahrscheinlichkeiten der Fehlverhalten ermittelt.

3.4.1 Einordnung von Gefährdungsanalysetechniken

Die Techniken der Gefährdungsanalyse lassen sich nach /Fenelon, McDermid 93/ in eine der vier Kategorien einordnen, welche in Abbildung 3-7 dargestellt sind.

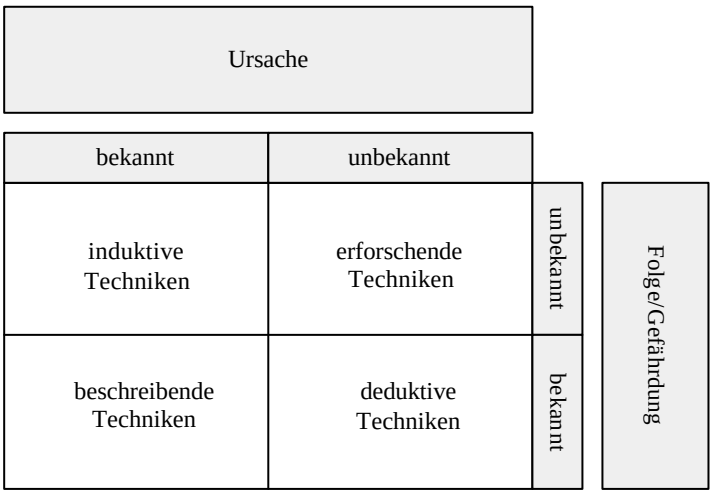
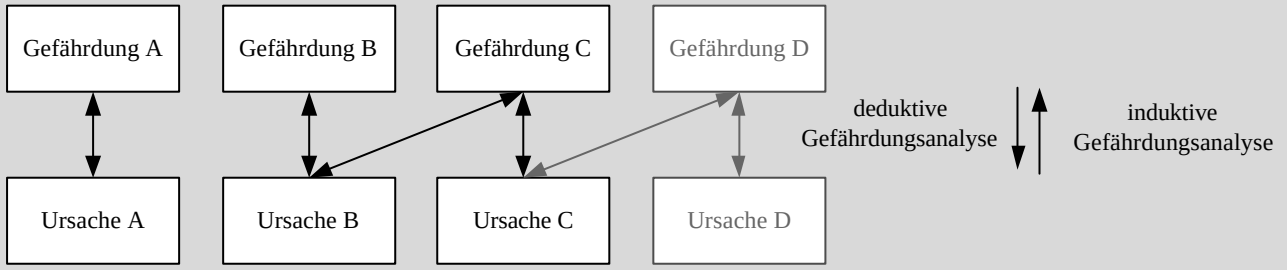


Abbildung 3-7 Einordnungsschema für Gefährdungsanalysetechniken

Erforschende Gefährdungsanalysetechniken werden angewendet, wenn sowohl alle Ursachen als auch alle Gefährdungen unbekannt sind. Dadurch ist ein strukturierter Ablauf der Gefährdungsanalyse jedoch nicht möglich und Gefährdungsanalysetechniken aus dieser Kategorie sind in der Praxis wenig verbreitet /Fenelon, McDermid 93/.
Bei den beschreibenden Techniken sind dagegen sowohl die Gefährdungen als auch deren Ursachen bekannt. Ihr Potenzial liegt daher in der Darstellung und Aufarbeitung der Wirkzusammenhänge zwischen Ursachen und Gefährdungen. Für die primäre Aufgabe der Gefährdungsanalyse, die Wirkzusammenhänge zwischen Ursachen und Gefährdungen zu identifizieren, sind beschreibende Techniken daher ebenfalls nur bedingt geeignet.
Als relevante Techniken werden lediglich die Techniken der induktiven als auch der deduktiven Kategorie angesehen /Leveson 95/. Dabei gehen induktive Techniken von bekannten Ursachen aus und untersuchen die daraus entstehenden Gefährdungen, wohingegen deduktive Techniken mögliche Ursachen für bekannte Gefährdungen untersuchen. In die Kategorie der induktiven Risikoanalysetechniken lassen sich mit der IF-FMEA und HAZOPS zwei im Folgenden beschriebene Techniken einordnen. Zu den deduktiven Risikoanalysetechniken zählt die Fehlerbaumanalyse.
Für eine vollständige Gefährdungsanalyse sollten sowohl induktive als auch deduktive Techniken kombiniert werden /Fenelon, Hebbron 94/. Erforderlich ist dies, da bei komplexen Systemen sowohl die Menge der Gefährdungen als auch die Menge der Ursachen für die Gefährdungen nicht vollständig erfassbar ist /Lutz, Woodhouse 97/. Durch eine Kombination von induktiven und deduktiven Techniken ist es jedoch möglich, die Anzahl der identifizierten Gefährdungen und Ursachen zu steigern.

Beispiel 3.7

Durch die Risikoanalyse sind die drei Gefährdungen A, B und C bekannt. Anhand einer deduktiven Analyse werden für diese Gefährdungen die Ursachen A, B und C wie in der Abbildung dargestellt ermittelt. Wird auf diese Ursachen eine induktive Gefährdungsanalysetechnik angewendet, so kann mit D eine zusätzliche Gefährdung identifiziert werden. Diese kann wiederum weitere Ursachen besitzen, welche sich durch eine erneute deduktive Analyse ermitteln lassen.



3.4.2 Fehlerbaumanalysen

Die Fehlerbaumanalyse ist eine in der Praxis oft angewandte Technik zur Gefährdungsanalyse. Nach /Liggesmeyer 00/ stellt sie eine modulare, hierarchische und formale Technik dar, die qualitative und quantifizierte Ergebnisse hinsichtlich der Gefährdungswahrscheinlichkeiten eines Systems liefert. Fehlerbäume sind unabhängig von der Art der Systemkomponenten anwendbar /Birolini 99/ und werden für Software- wie Hardwarekomponenten genutzt. Sie eignen sich daher zur Analyse von softwareintensiven technischen Systemen.
Für die Fehlerbaumanalyse wird eine Notation verwendet, welche eine wohldefinierte Syntax und Semantik besitzt. Diese Notation wird in den Normen /DIN 25424/ bzw. /IEC 61025/ beschrieben. Nach diesen Normen besteht ein Fehlerbaum aus Verknüpfungs- und Zusatzsymbolen. Die Verknüpfungssymbole sind logische Operatoren, die eine Menge von Eingaben auf eine Ausgabe abbilden. Daher ist jeder Fehlerbaum syntaktisch eine hierarchische, boolesche Funktion. In der Abbildung 3-8 sind Verknüpfungssymbole nach der /DIN 25424/ dargestellt.

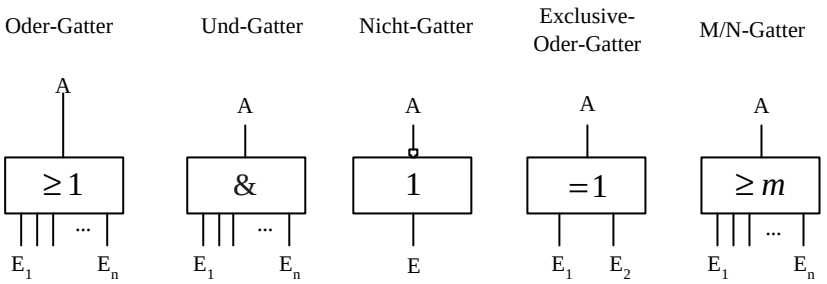


Abbildung 3-8 Verknüpfungssymbole nach /DIN 25424-1/ und /DIN 25424-2/

Neben den Verknüpfungssymbolen werden für den syntaktischen Aufbau eines Fehlerbaumes auch Zusatzsymbole benötigt. Zusatzsymbole sind nach der /DIN 25424-1/ zum Beispiel Eingänge, Kommentare und administrative Symbole.

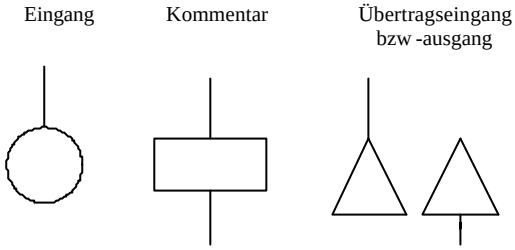


Abbildung 3-9 Zusatzsymbole eines Fehlerbaumes nach /DIN 25424-1/

3.4.3 HAZOPS und HAZOPS-basierte Techniken

Ursprünglich wurde HAZOPS (*HAZard and OPerational Study*) zur Bestimmung von Fehlverhalten und deren Auswirkungen in chemischen Anlagen verwendet. Ein Merkmal dieser Anlagen ist, dass sie Flüsse von chemischen Substanzen beinhalten, die im Gefährdungsfall reagieren und Unfälle verursachen. Durch die HAZOPS werden die Flüsse der chemischen Substanzen strukturiert auf mögliche Abweichungen von den spezifizierten Flusseigenschaften untersucht. Dazu werden Leitworte verwendet, welche den Zustand des Flusses beschreiben. Beispiele für diese Leitworte sind unter anderem: kein Fluss, zuviel, zuwenig, mehr als, ein Teil von. Für jede Abweichung wird untersucht, welche Ursachen dieser Abweichung zugrunde liegen und welche möglichen Gefährdungen von dieser Abweichung ausgehen.
Die Anwendung von HAZOPS wurde auch für Software- und Computersysteme untersucht. Dabei wurden mit den CHAZOP (*Computer HAZOP*) /Burns, Pitblado 93/, SHAZOPS (*Software HAZOPS*) /McDermid, Pumfrey 94/, SHARD (*Software Hazard Analysis and Resolution in Design*) /Pumfrey 99/ und PES HAZOP (*Programmable Electronic Systems HAZOP*) /Def Std 00-58/ unterschiedliche software- und computerbezogene Techniken entwickelt. Von diesen Techniken eignet sich SHARD besonders für die Analyse von objektorientierten Architekturmodellen /Pumfrey 99/. Daher wird sie im Folgenden beispielhaft für die anderen Techniken kurz vorgestellt und erläutert.
Die SHARD untersucht auf Basis der Strukturspezifikation in einem Zyklus alle relevanten Ereignissequenzen zwischen den Architekturelementen auf mögliche Abweichungen von dem gewünschten Verhalten. Zur Durchführung von SHARD sind daher grundlegende Kenntnisse der Architekturelemente und der Informationsflüsse zwischen ihnen erforderlich. Dazu stellt der Systemarchitekt die Architektur kurz vor und beschreibt die Intention der einzelnen Architekturelemente. Danach wird für jede Ereignissequenz eine Abweichung von dem gewünschten Verhalten anhand der ausgewählten Leitworte untersucht. Die Untersuchungen in /McDermid, Pumfrey 94/ haben gezeigt, dass die Leitworte der herkömmlichen HAZOP in Bezug auf programmierbare elektronische Systeme nicht geeignet sind. Daher mussten neue Leitworte für die Untersuchung dieser Systeme gefunden werden. Nach /McDermid, Pumfrey 94/ stellten sich die folgenden fünf als geeignet heraus:

- wert-fehlerhaft
- frühzeitig
- verspätet

– unerlaubt, unerwartet
– unterlassen

Mit diesen fünf Leitworten werden Abweichungen der Ereignissequenz vom spezifizierten Verhalten identifiziert. Diese Abweichungen werden anschließend auf ihre Effekte und Ursachen hin untersucht. Sind die Ursachen einer fehlerhaften Ereignissequenz bereits hinreichend genau identifiziert und beschrieben, werden die Effekte begutachtet. Dabei wird die Frage beantwortet, welchen Einfluss die fehlerhafte Ereignissequenz auf die gesamten Ausgaben des Systems hat. Somit kann festgestellt werden, wie sich die Fehlverhalten auf die funktionalen und die Sicherheitsanforderungen auswirken und welche Fehlverhalten zu eventuellen Gefährdungen führen. Führt eine fehlerhafte Ereignissequenz de facto zu einer Gefährdung, muss darüber hinaus untersucht werden, wie stark der Einfluss auf die Gefährdung ist. Für alle fehlerhaften Ereignissequenzen, die einen Einfluss auf eine potenzielle, vom System ausgehende Gefährdung besitzen, müssen darüber hinaus die Fehlererkennungs- und Fehlerbehebungs-Mechanismen untersucht werden. Ausgehend von diesen Mechanismen kann das Team Vorschläge für den Entwurf machen, um die Wahrscheinlichkeit der resultierenden Gefährdungen zu reduzieren.

Die Ergebnisse der SHARD lassen sich in einer Tabelle darstellen. Die Spalten der Tabelle repräsentieren hierbei die in der SHARD identifizierten Informationen:

– Fehlerhafte Ereignissequenz
– Leitwort
– Abweichung
– Mögliche Ursachen
– Effekte
– Erkennungs- und Sicherungsmechanismen
– Vorschläge für den Entwurf

Ausgehend von den Veröffentlichungen zu den verschiedenen HAZOPS-basierte Techniken und der SHARD wurde die in Abbildung 3-10 dargestellte Vorgehensweise zur Anwendung der HAZOPS in softwareintensiven technischen Systemen erstellt. Diese beschreibt zusammenfassend die Hauptaktivitäten der computer- und softwareorientierten HAZOPS.

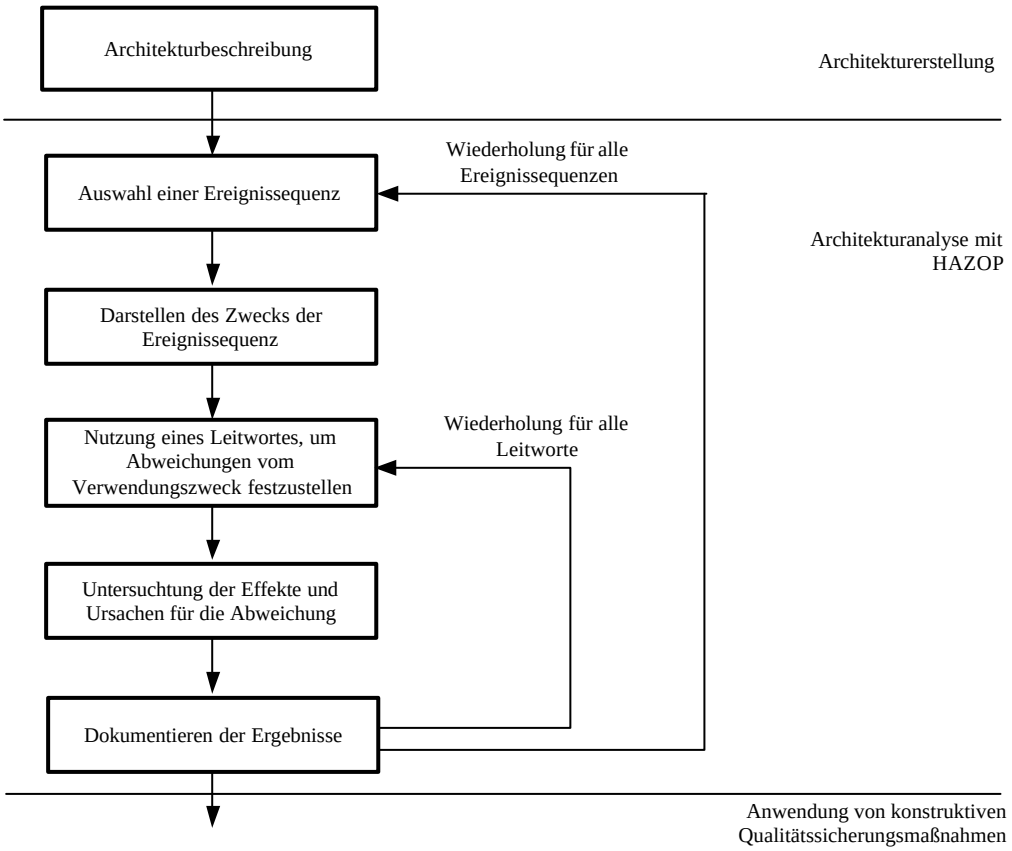


Abbildung 3-10 Vorgehensweise bei der Gefährdungsanalyse mit HAZOPS

3.4.4 IF-FMEA

Die IF-FMEA (Interfacefokussierte FMEA) /Papadopoulos et al. 01/ ist eine induktive Gefährdungsanalysetechnik, welche auf der systemorientierten FMECA /DIN 25448/ basiert. In ihrer Vorgehensweise ähnelt sie den HAZOPS-basierten Techniken. Angewendet wird die IF-FMEA auf die Interfacespezifikation eines Architekturelementes, da sich dort Fehlverhalten offenbaren. Für die Identifikation der relevanten Fehlverhalten wird deshalb jede ausgehende Ereignissequenz mit den bereits dargestellten Leitworten untersucht. Für jedes identifizierte Fehlverhalten werden anschließend ebenfalls die Auswirkungen auf die vom System ausgehenden Gefährdungen ermittelt.
Die IF-FMEA untersucht jedoch nicht nur die von den Architekturelementen ausgehenden Fehlverhalten, sondern auch das Verhalten des Architekturelementes bei fehlerhaften Eingaben. Dies ist deshalb entscheidend, da ein Fehlverhalten eines Architekturelementes entweder auf ein internes Fehlverhalten oder auf eine fehlerhafte Eingabesequenz zurückgeführt werden kann /Papadopoulos 00/. In der IF-FMEA werden daher auch die eingehenden Ereignissequenzen betrachtet. Bei einer gleichzeitigen Betrachtung der eingehenden und ausgehenden Ereignissequenzen sind somit Aussagen über die Behandlung, die Erzeugung, die Transformation und die Fortpflanzung von Fehlverhalten möglich.
Die Ergebnisse der IF-FMEA werden im Allgemeinen tabellenartig dokumentiert. Dazu werden alle relevanten Fehlverhalten der ausgehenden Ereignissequenzen und ihre Wahrscheinlichkeiten für ein Architekturelement aufgelistet und beschrieben. Für jedes Fehlverhalten werden zusätzlich die Ursachen in zwei Spalten mit logischen Formeln dokumentiert. Die eine Spalte enthält dabei die internen Fehler und die andere Spalte die fehlerhaften Eingabesequenzen. Für die Berechnung der Wahrscheinlichkeit einer fehlerhaften Ereignissequenz sind die Wahrscheinlichkeiten der internen Fehler und eingehenden fehlerhaften Ereignissequenzen sowie die logischen Verknüpfungen zwischen diesen zu betrachten.
Die Ergebnisse der IF-FMEA sind auch in der Fehlerfortpflanzungs- und Fehlertransformations–Notation (*failure propagation transformation notation* FPTN) dokumentierbar. Diese ist eine grafische Notation zur Darstellung der fehlerhaften Ereignissequenzen in Systemen mit einer komplexen internen Struktur. Ursprünglich wurde sie in /Fenelon, McDermid 93/ beschrieben und in /Fenelon et al. 94/ weiterentwickelt.
Ein Architekturelement wird in der FPTN grafisch als Viereck mit abgerundeten Ecken dargestellt. Jedem dargestellten Architekturelement werden fehlerhafte eingehende und ausgehende Ereignissequenzen zugewiesen, welche Instanzen eines Fehlverhaltenstyps sind. Verwendet werden die folgenden Typen:

- *zs* Ereignis erfolgt außerhalb des spezifizierten Zeitfensters und ist zu spät
- *zf* Ereignis erfolgt außerhalb des spezifizierten Zeitfensters und ist zu früh
- w Ereignis ist außerhalb des spezifizierten Wertebereiches
- *u* Ereignis wurde unterlassen
- *ng* Ereignis wurde erzeugt aber nicht gefordert

Wird einem Bezeichner der Buchstabe *e* angehängt/beigefügt, so zeigt dies, dass die fehlerhafte Ereignissequenz von der Komponente erkannt wird.
Die Beziehungen zwischen den fehlerhaften ein- und ausgehenden Ereignissequenzen werden anhand von logischen Regeln beschrieben. Diese Regeln sind Transformationsregeln, Fortpflanzungsregeln, Behandlungsregeln und Generierungsregeln. Sie werden im Rumpf der FPTN-Komponente festgehalten. Zusätzlich zu den Informationen über die fehlerhaften Ereignissequenzen enthält die FPTN noch Informationen zu den einzelnen Architekturelementen. Diese Informationen dienen der Identifizierung des Architekturelementes, der Typbestimmung und der Bestimmung der zugewiesenen Sicherheitsanforderungsstufe. Diese Informationen werden im Kopf einer FPTN-Komponente abgebildet.
In Abbildung 3-11 wird eine Architekturkomponente in der FPTN dargestellt. Diese Architekturkomponente besitzt vier fehlerhafte eingehende Ereignissequenzen (A,B,C,X) und drei fehlerhafte ausgehende Ereignissequenzen (D,E,F). Für die fehlerhafte Ereignissequenz D wird eine Fortpflanzungsregel und für die fehlerhafte Ereignissequenz E eine Transformationsregel angegeben. Die fehlerhafte Ereignissequenz F wird dagegen nicht durch eine Transformations- oder Fortpflanzungsregel, sondern durch eine Generierungsregel erzeugt.
Die fehlerhaften eingehenden Ereignissequenzen A, B und C sind hierbei die Ursache für die fehlerhaften, ausgehenden Ereignissequenzen. Sie werden in der Transformationsregel und Fortpflanzungsregel als Eingabe benutzt. Der Fehlerfluss X wird von der Komponente jedoch nicht weitergegeben, da er von der Komponente erkannt und durch einen geeigneten Mechanismus verhindert wird.

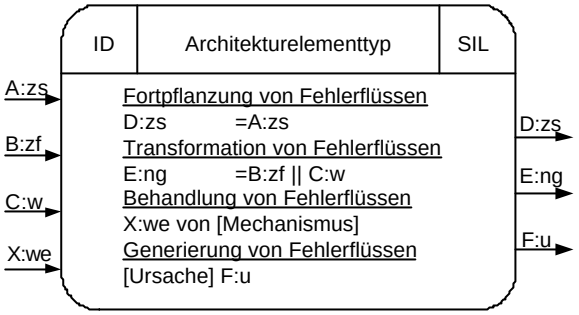


Abbildung 3-11 Darstellung einer Architekturkomponente in der FPTN

In der Fehlerfortpflanzungs- und Fehlertransformations-Notation sind Architekturelemente ähnlich wie in den Architekturspezifikationen aus feingranulareren Elementen aufgebaut. Die Fehlerflüsse eines Elementes werden in diesem Fall an die enthaltenen Elemente weitergeleitet. Der Aufbau der Kompositionshierarchie der FPTN entspricht somit der Hierarchie der Strukturspezifikation der untersuchten Architektur.

3.4.5 HiP-HOPS

HiP-HOPS (*hierarchically performed hazard origin and propagation studies*) ist eine Gefährdungsanalyse-Methode /Papadopoulos 00/, /Papadopoulos et al. 01/, welche aufbauend auf der Strukturspezifikation des Systems die Ursachen der Gefährdungen untersucht. Durch HiP-HOPS werden die Fehlverhalten und deren Auswirkungen in einem komplexen System von der Systemebene bis zur Ebene der elementaren Systembestandteile untersucht. Dazu wird auf der Systemebene mit der funktionsbezogenen Fehleranalyse eine Technik der Risikoanalyse verwendet. Diese dient zur Identifizierung der grundlegenden Gefährdungen und zur Darstellung der zu diesen Gefährdungen führenden funktionalen Fehlverhalten des Systems. Anschließend wird das System, wie bereits dargestellt, in mehrere Architekturelemente aufgeteilt, wodurch ein hierarchisches Modell des Systems entsteht. Die Architekturelemente werden danach mit der IF-FMEA auf mögliche lokale Fehlverhalten hin untersucht. Diese werden für jedes Architekturelement in einer Tabelle dargestellt. Sind für alle Architekturelemente die möglichen Fehlverhalten bekannt, wird untersucht, wie sich diese Fehlverhalten auf das gesamte System auswirken. Dabei müssen die identifizierten funktionalen Fehlverhalten des Systems und die lokalen Fehlverhalten der Architekturelemente in Beziehung zu einander gebracht werden. Basierend auf der hierarchischen Struktur werden dazu systematisch generierte Fehlerbäume verwendet. Für jedes funktionale Fehlverhalten wird genau ein Fehlerbaum erzeugt, welcher die lokalen Fehlverhalten der Architekturelemente auf die globalen Fehlverhalten des Systems abbildet.

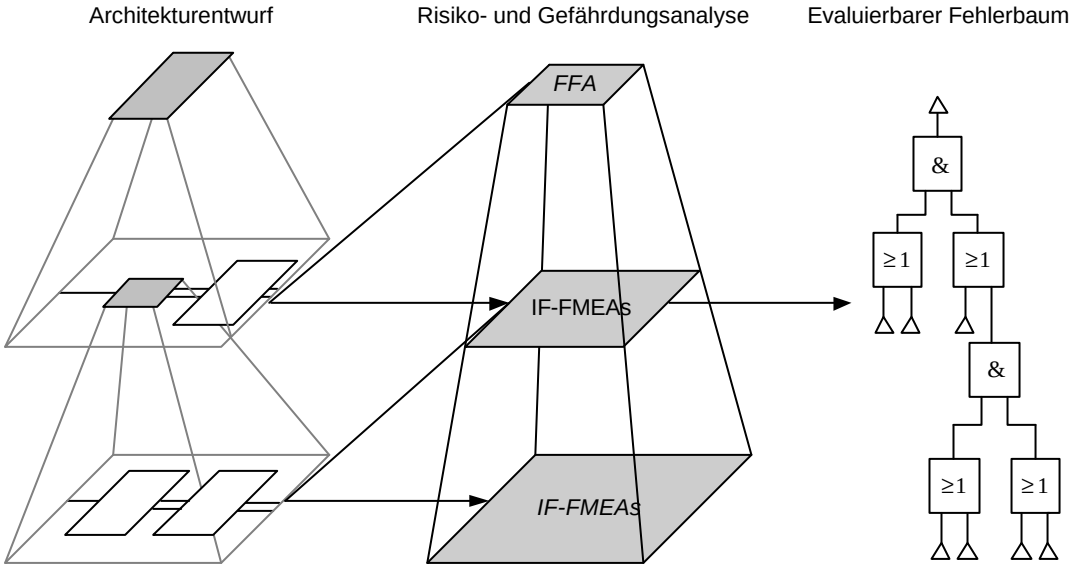


Abbildung 3-12 Überblick über die Hip-HOPS Methode /Papadopoulos 00/

Der Hauptvorteil von HiP-HOPS ist der nahtlose Übergang zwischen dem Architekturentwurf und der Gefährdungsanalyse. Es besteht eine direkte Abbildung der im Architekturentwurf entwickelten Systemstruktur und deren Verhältnis zu den analysierten Komponenten der Gefährdungsanalyse.
Die hierarchische Strukturierung des Systems wird in der Praxis oft in einer späteren Phase der Systementwicklung verändert. Dadurch decken sich die Ergebnisse der Gefährdungsanalyse nicht immer mit dem realen System. Bei HiP-HOPS wird die Gefährdungsanalyse jedoch automatisch bei einer Veränderung der hierarchischen Strukturierung überarbeitet. Dadurch ist die Gefährdungsanalyse stets deckungsgleich mit dem Verhalten des Systems.

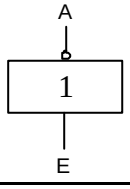
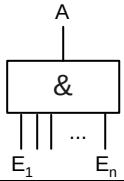
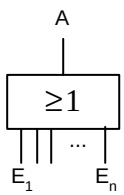
3.5 Nachweis von probabilistischen Eigenschaften

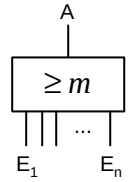
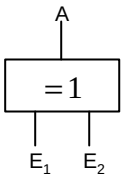
Zur analytischen Qualitätssicherung und zum Nachweis von probabilistischen Eigenschaften, wie z. B. Zuverlässigkeits-, Wartbarkeits- und Verfügbarkeitseigenschaften, im Architekturentwurf werden Annahmen über die stochastischen Eigenschaften der enthaltenen Komponenten getroffen. Ausgehend von diesen Annahmen wird durch komponierende Techniken auf die Eigenschaften des gesamten Systems geschlossen. Im Folgenden werden mit der Fehlerbaumanalyse, der Analyse von Zuverlässigkeits-Blockdiagrammen und der Markov-Analyse in der Praxis verbreitete Techniken vorgestellt /Liggesmeyer 00/.

3.5.1 Fehlerbaumanalyse

Fehlerbäume sind neben der Gefährdungsanalyse auch für die Analyse von stochastischen Eigenschaften einsetzbar. Insbesondere eignen sie sich für die Analyse von Zuverlässigkeitseigenschaften. Dazu werden zunächst die Teilkomponenten identifiziert, welche zur Erfüllung einer funktionalen Anforderung notwendig sind. Anschließend wird mit dem Fehlerbaum dargestellt, wie sich ein Ausfall einer solchen Teilkomponente auf das Gesamtsystem auswirkt. Sind die Zuverlässigkeiten aller Teilkomponenten bekannt, so kann anschließend anhand des Fehlerbaumes die Zuverlässigkeit ermittelt werden, mit der das System die funktionalen Anforderungen erfüllt. Zur Auswertung der Systemzuverlässigkeit aus einem Fehlerbaum sind zusätzlich noch Berechnungsvorschriften nötig. Diese sind Bestandteil der /DIN 25424/ und für die jeweiligen Verknüpfungssymbole in Tabelle 3-3 dargestellt.

Tabelle 3-3 Berechnungsvorschriften für die Zuverlässigkeit von Fehlerbäumen

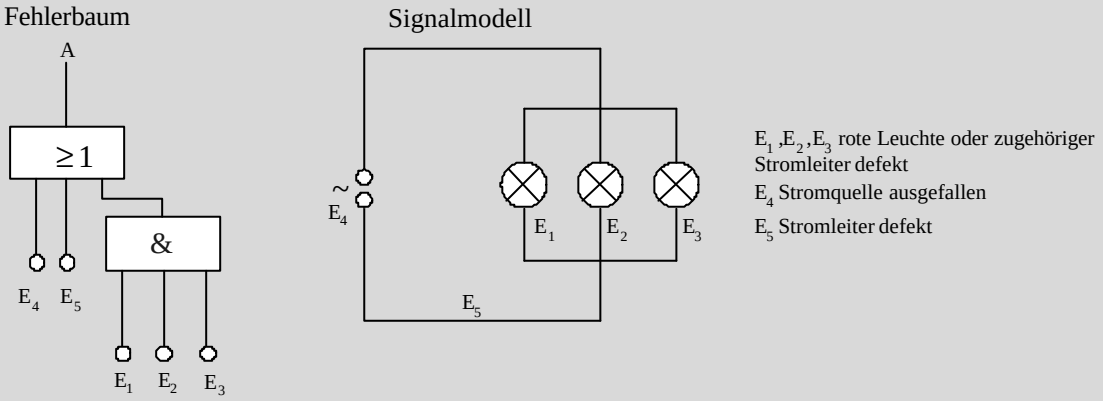
Name	Verknüpfungssymbol	Berechnungsvorschrift
Nicht-Verknüpfung		$R_{S[NICHT]}(t) = 1 - R_i(t) = F_i(t)$
Und-Verknüpfung		$R_{S[UND]}(t) = 1 - \prod_{i=1}^{n} (1 - R_i(t))$
Oder-Verknüpfung		$R_{S[ODER]}(t) = \prod_{i=1}^{n} R_i(t)$

M/N-Verknüpfung		Komponenten mit identischen Ausfallwahrscheinlichkeiten $R_{S[M/N]}(t) = \sum_{i=M}^{N} \binom{N}{i} R(t)^i (1-R(t))^{N-i}$ Komponenten mit unterschiedlichen Ausfallwahrscheinlichkeiten $R_{S[M/N]}(t) = \prod_{i=M}^{N} R_{S[M-1/i-1]}(t) + R_i(t) - R_i(t) R_{S[M-1/i-1]}(t)$
Exklusiv-Oder-Verknüpfung		$R_{S[EX-ODER]}(t) = 1 - \sum_{i=1}^{n} (1 - R_i(t))$

Beispiel 3.8

Zur Illustration der Zuverlässigkeitsanalyse mit einem Fehlerbaum soll die Zuverlässigkeit der Funktion bestimmt werden, mit der das Verbot einer Zugdurchfahrt durch ein rotes Leuchtsignal angezeigt wird. Zur Erhöhung der Zuverlässigkeit besitzt dieses Signal drei redundante Lampen. Das System und der dazugehörige Fehlerbaum wird in der Abbildung schematisch modelliert. Mit den Eingaben E_1 bis E_5 werden die dazugehörigen Überlebenswahrscheinlichkeitsfunktionen $R_1(t)$ bis $R_5(t)$ der einzelnen Komponenten verknüpft. Aus dem modellierten Fehlerbaum ergibt sich die folgende Berechnungsformel für die Systemzuverlässigkeit $R_S(t)$:

$$R_S(t) = R_4(t) \cdot R_5(t) \cdot \left[1 - \prod_{i=1}^{3} (1 - R_i(t))\right]$$



3.5.2 Zuverlässigkeits-Blockdiagramme

Zuverlässigkeits-Blockdiagramme /IEC 61078/ (engl. *reliability block diagrams*, auch RBD abgekürzt) gestatten die Ermittlung der Systemzuverlässigkeit basierend auf der Komponentenzuverlässigkeit /Liggesmeyer 00/. Sie besitzen einen ähnlichen Aufbau und eine ähnliche Aussagekraft wie die Fehlerbäume und sind für die Zuverlässigkeitsanalyse ineinander überführbar /Birolini 99/. Die Überführungsvorschriften für Serien- und Parallelschaltungen sind in Abbildung 3-13 dargestellt.

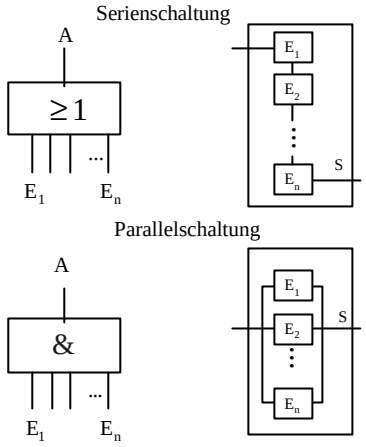
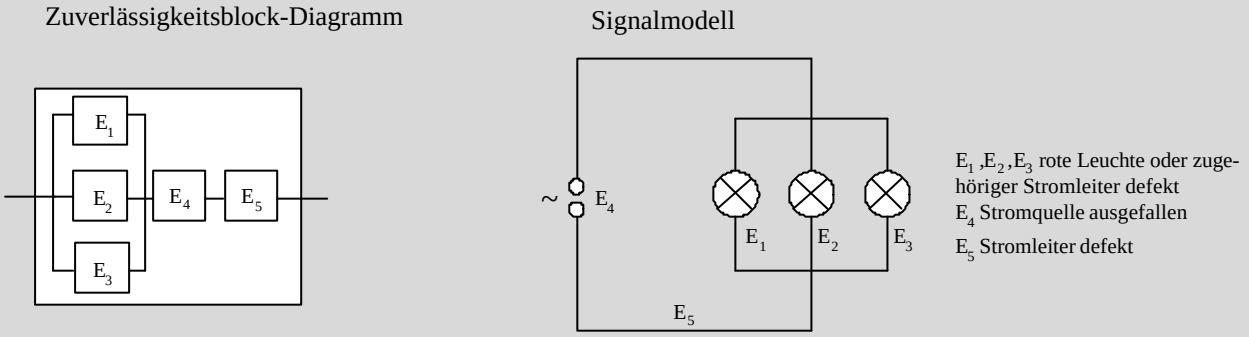


Abbildung 3-13 Überführungsvorschriften

Beispiel 3.9

Für das in der Fehlerbaumanalyse verwendete Beispiel ergibt sich demnach für das Signal das dargestellte Zuverlässigkeits-Blockdiagramm. Die Berechnungsvorschrift für die Bestimmung der Zuverlässigkeit ist in beiden Modellen identisch.



3.5.3 Markov-Analysen

Markov-Analysen werden unter anderem zur Analyse der Verfügbarkeit von softwareintensiven technischen Systemen /Liggesmeyer 00/ verwendet. Grundlage der Analyse sind hierbei Markov-Modelle, welche Zustandsautomaten beschreiben, deren Zustandsübergänge mit Wahrscheinlichkeiten annotiert sind. Für die Verfügbarkeitsanalyse mit Markov-Modellen beschreiben die Zustände entweder das intakte System oder das System, welches vollständig bzw. bei dem eine Komponente ausgefallen ist. Die Zustandsübergangswahrscheinlichkeiten beschreiben Annahmen über die Ausfall- und Reparaturwahrscheinlichkeit einer Komponente oder des gesamten Systems. Für die Bestimmung der Verfügbarkeit ist die Aufenthaltswahrscheinlichkeit in den Zuständen, welche das intakte System beschreiben, zu bestimmen.
In Abbildung 3-14 ist ein einfaches Markov-Modell dargestellt. Dieses beschreibt ein System, welches im Zustand P_0 intakt und im Zustand P_1 ausgefallen ist.

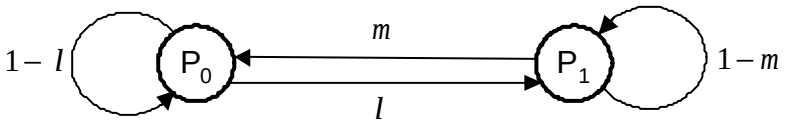


Abbildung 3-14 Markov-Modell

Die Verfügbarkeit *A(t)* der Betrachtungseinheit entspricht dabei der Aufenthaltswahrscheinlichkeit, mit der das System zum Zeitpunkt *t* im Zustand P_0 ist. Demnach gilt:

$$A(t) = P_0(t)$$

Die Summe der Aufenthaltswahrscheinlichkeiten beider Zustände ist eins.

$$1 = P_0(t) + P_1(t)$$

Die Zustandsübergänge werden durch die Ausfallrate und die Reparaturrate beschrieben. Das Markov-Modell aus der Abbildung 3-14 hat die folgende Zustandsübergangsmatrix:

$$P = \begin{array}{c} \\ 0 \\ 1 \end{array} \begin{array}{c} \begin{array}{cc} 0 & 1 \end{array} \\ \begin{pmatrix} 1-\lambda & \lambda \\ \mu & 1-\mu \end{pmatrix} \end{array}$$

Für die Bestimmung der Verfügbarkeit ist es erforderlich, die Aufenthaltswahrscheinlichkeit im Punkt P_0 zu bestimmen. Dies erfolgt durch Lösung des sich aus der Zustandsübergangsmatrix ergebenden Differenzialgleichungssystem:

$$P_0(t+dt) = P_0(t)(1-\lambda\, dt) + P_1(t)\mu\, dt$$

$$P_1(t+dt) = P_0(t)\,\lambda\, dt + P_1(t)(1-\mu\, dt)$$

Der Lösungsweg des Differenzialgleichungssystems wird in /Liggesmeyer 00/ und /MIL HDBK 338B/ beschrieben. Als Ergebnis gilt für die Aufenthaltswahrscheinlichkeiten des Systems im Zustand P_0:

$$P_0(t) = \frac{\mu}{\lambda+\mu} + \left(c - \frac{\mu}{\lambda+\mu} e^{-(\lambda+\mu)t} \right)$$

Die Konstante *c* ist dabei die Aufenthaltswahrscheinlichkeit des Systems bei seiner Inbetriebnahme im Zustand P_0. Aus der allgemeinen Berechnungsvorschrift lassen sich für den Zustand P_0 die folgenden beiden Spezialfälle ableiten:

$$P_0(t=0) = \frac{\mu}{\lambda+\mu} + \left(c - \frac{\mu}{\lambda+\mu} \right) = c \quad und \quad \lim_{t\to\infty} P_0(t) = \frac{\mu}{\lambda+\mu}$$

Für eine unendliche Systemlaufzeit ergibt sich die Verfügbarkeit des Systems wie in Absatz 2.4 dargestellt aus der *MTBF* und der *MTTR* nach der folgenden Berechnungsvorschrift:

$$A_s = \lim_{t\to\infty} A(t) = \lim_{t\to\infty} P_0(t) = \frac{\mu}{\lambda+\mu} \; mit \quad \lambda = \frac{1}{MTBF} \quad \mu = \frac{1}{MTTR}$$

$$A_s = \frac{\frac{1}{MTTR}}{\frac{1}{MTBF} + \frac{1}{MTTR}} = \frac{\frac{1}{MTTR}}{\frac{MTBF + MTTR}{MTBF \cdot MTTR}} = \frac{MTBF}{MTBF + MTTR}$$

3.6 Nachweis der Echtzeitfähigkeit

Grundlage für die Analyse der Echtzeitfähigkeit sind Annahmen über die Echtzeiteigenschaften der Komponenten. Diese werden, wie in Abschnitt 3.3.3 dargestellt, in die Interfacespezifikation integriert. Auch wenn alle Annahmen korrekt und miteinander vereinbar sind, ist es dennoch möglich, dass das System seine Echtzeitanforderungen nicht einhält. Grund dafür sind Wartezeiten bei der gemeinsamen Nutzung der Hardwareplattform und anderer Ressourcen. Es ist daher nachzuweisen, dass durch die Zuweisung der Hardwareplattform und der gemeinsamen Ressourcen die Echtzeitanforderungen nicht verletzt werden. Dies erfolgt durch die Schedulinganalyse.

Grundsätzlich werden nach /Hüsener 95/ statische und dynamische Schedulingalgorithmen unterschieden. Statische Schedulingalgorithmen weisen dabei den Komponenten zur Entwurfszeit den Ausführungszeitraum auf der Hardwareplattform zu. Daher ist es nicht möglich, Komponenten dynamisch zur Laufzeit zu erzeugen. Dynamische Schedulingalgorithmen ordnen dagegen einer Komponente auf Basis ihrer Priorität zur Laufzeit den Ausführungszeitraum zu. Dieser Ausführungszeitraum kann bei präemptiven Scheduling-Algorithmen auch unterbrochen werden, wenn eine Komponente mit einer höheren Priorität die Hardwareplattform benötigt. Im Folgenden wird die Schedulinganalyse am Beispiel der Rate-Monotonic-Analysis für dynamische Schedulingalgorithmen und der Analyse von zyklischen Modellen für statische Schedulingalgorithmen erläutert.

3.6.1 Rate-Monotonic-Analysis

Das Rate-Monotonic-Scheduling ist ein preemtives Verfahren zur Vergabe des Ausführungszeitraumes bei einer statischen Anzahl von zyklischen Prozessen. Die Vergabe erfolgt dabei auf Basis der Prozesspriorität P_i, welche indirekt proportional zur Periodendauer T_i des Prozesses ist. Zur Vereinfachung der Analyse ist beim Rate-Monotonic-Scheduling auch die Periodendauer äquivalent zur geforderten Zeitschranke D_i. Zusätzlich wird für jeden Prozess P_i eine Annahme über die maximale Ausführungszeit C_i getroffen. Basierend auf dieser Annahme halten nach /Liu, Layland 73/ alle Prozesse ihr Zeitschranken ein, wenn gilt:

$$\sum_{q=1}^{n} \frac{C_q}{T_q} \leq n\left(2^{\frac{1}{n}} - 1\right)$$

Dieser Test ist hinreichend, aber nicht notwendig. Das bedeutet, wenn die Prozesse der Architektur den Test nicht bestehen, ist es nicht zwingend zu schlussfolgern, dass ein Prozess seine Zeitschranke verletzt. Daher wird dieser Test in /Hüsener 95/ auch als pessimistisch bezeichnet.

Eine exakte Aussage über die Einhaltung der Zeitschranke eines Prozesses ist mit dem in /Sha, Goodenough 90/ vorgestellten Test möglich. Dieser Test berechnet die Auslastung der Hardwareplattform. Ist sie kleiner als eins, so hält der Prozess seine Zeitschranke ein. Die Auslastung der Hardwareplattform ergibt sich dabei aus der Summe der eigenen Auslastung und der Auslastung von Prozessen mit einer höheren Priorität. Somit halten alle Prozesse ihre Zeitschranken ein, wenn für jeden Prozess P_i gilt:

$$\frac{C_i}{T_i} + \min_{0 \leq t \leq D_i}\left(\sum_{q=1}^{i-1} \frac{C_q}{t}\left\lceil \frac{t}{T_q} \right\rceil\right) \leq 1$$

Dieser Test ist für Prozesse anwendbar, welche außer der Hardwareplattform auf keine weiteren gemeinsamen Ressourcen zugreifen. Enthält die Architektur jedoch solche gemeinsam genutzte Ressourcen, ist es notwendig den Test zu erweitern, da durch diese Ressourcen ein Prozess auch von Prozessen mit einer niedrigeren Priorität geblockt werden kann. Die maximale Blockierungszeit B_i ist daher für jeden Prozess P_i anzugeben, und die damit verbundene Auslastung ist wie folgt in den Test zu integrieren:

$$\frac{C_i}{T_i} + \min_{0 \leq t \leq D_i}\left(\sum_{q=1}^{i-1} \frac{C_q}{t}\left\lceil \frac{t}{T_q} \right\rceil\right) + \frac{B_i}{T_i} \leq 1$$

Die Ermittlung der maximalen Blockierungszeit B_i wird durch das *priority inheritance* /Sha et al. 90/ oder *priority ceiling* Protokoll /Sha, Goodenough 90/ erleichtert. Diese Protokolle stellen sicher, dass ein Prozess durch maximal einen niederprioren Prozess geblockt wird. Dadurch ergibt sich die maximale Blockierungszeit B_i aus der maximalen Ausführungszeit in einer von P_i gemeinsam genutzten Ressource. Des Weiteren lösen diese Protokolle auch das *priority inversion* Problem /Goodenough, Sha 88/.

Für die Integration von sporadischen Prozessen wird in /Sprunt et al. 89/ ein sporadischer Server vorgeschlagen. Dieser sporadische Server ist ein periodischer Prozess, welcher sporadische Prozesse zu einem Cluster zusammenfasst und diese periodisch aufruft. Die Periodendauer muss dazu kürzer sein als die minimale Ankunftszeit der Ereignisse der sporadischen Prozesse.

3.6.2 Analyse von statischen zyklischen Modellen

Statische Schedulingalgorithmen definieren bereits zur Entwurfszeit den Start- und Endzeitpunkt des Ausführungszeitraums eines periodischen Prozesses P_i. Dazu wird bei Systemen mit endlicher Laufzeit ein Schedulingplan erstellt. Für Systeme mit unendlicher Laufzeit werden hingegen zyklische Modelle vorgeschlagen /Shaw 01/. Diese erstellen einen Scheduling-Plan für jeweils einen Zyklus und duplizieren diesen. Bei der Analyse des Schedulingplans ist dabei nachzuweisen, dass die Zeitschranken nicht verletzt werden. Dazu ist jeder Prozess in einem Zyklus mindestens so oft periodisch aufzurufen wie der ganzzahlig aufgerundete Quotient aus der Zykluszeit und der Periodendauer des Prozesses:

$$n_i \geq \left\lceil \frac{Z}{T_i} \right\rceil$$

Der Schedulingplan eines zyklischen Modells kann jedoch auch so erstellt werden, dass a priori alle Zeitbedingungen eingehalten werden. Die dazu verwendeten Verfahren werden in /Shaw 01/ dargestellt.

3.7 Konstruktive Qualitätssicherungstechniken

Konstruktive Qualitätssicherungstechniken in der Entwurfsphase dienen der strukturierten Entwicklung der Entwurfsspezifikation. Dabei sollte die Entwurfsspezifikation so entwickelt werden, dass a priori Fehler vermieden werden.
Dazu ist es erforderlich, die Verständlichkeit der Entwurfsspezifikation, die Wiederverwendbarkeit und die Sicherheitseigenschaften einer Architekturspezifikation zu verbessern.

3.7.1 Förderung der Verständlichkeit der Entwurfsspezifikation

Die Förderung der Verständlichkeit der Entwurfsspezifikation kann unter anderem durch eine geeignete Systemmodularisierung, eine geeignete Modularisierung der Interfacespezifikationen, eine geeignete Abstraktion sowie durch Anforderungsverfolgung erreicht werden.

Geeignete Systemmodularisierung
Bei der Entwicklung der Architekturspezifikation wird das System in Komponenten zerlegt. Dies wird als Systemmodularisierung bezeichnet. Eine Systemmodularisierung wird als geeignet bezeichnet, wenn die resultierenden Komponenten

- überschaubare Einheiten sind.
- unabhängig voneinander entwickelbar, testbar und änderbar sind.
- minimale Schnittstellen miteinander haben.
- wiederverwendbar sind.
- hierarchisch komponierbar sind.

Für die Bewertung der Modularisierungsqualität eignen sich die beiden Maße Kopplung und Bindung /Balzert 01/.
Die Kopplung betrachtet dabei die Interaktionen zwischen den Komponenten. Je enger die Kopplung zwischen verschiedenen Komponenten ist, desto komplizierter und schwerer verständlich wird ein System. Des Weiteren werden Änderungen durch eine hohe Kopplung erschwert /Briand et al. 00/, /Chidamber, Kemerer 94/.
Bindung ist dagegen das Maß für den inneren Zusammenhang einer Komponente. Je stärker die Bindung in einer Komponente ist, um so verständlicher ist die Komponente und um so besser lässt sich diese Komponente realisieren /Bieman, Kang 98/, /Chidamber, Kemerer 94/.
Zusammenfassend sollte für eine geeignete Systemmodularisierung und eine verständliche Architekturspezifikation also eine lose Kopplung zwischen den Komponenten und eine starke Bindung innerhalb der Komponenten angestrebt werden.

Geeignete Modularisierung der Interfacespezifikationen
Die Komplexität der Interfacespezifikationen steigt mit der Komplexität der modellierten Komponente. Für die Reduktion der Komplexität werden die Interfacespezifikationen durch Modularisierung in kleinere Teile aufgeteilt. Diese Teile werden wie bereits dargestellt als Ports bezeichnet. Sie charakterisieren die speziellen Interaktionen mit einer anderen Komponente. Daher wird die Modularisierung der Interfacespezifikationen dann als geeignet bezeichnet, wenn ausschließlich diejenigen Ereignissequenzen durch einen Port beschrieben werden, welche für die Interaktion mit der entsprechenden Komponente erforderlich sind /Selic et al. 94/.

Geeignete Abstraktion
Abstraktion ist ein geeignetes Mittel zur Reduktion der Komplexität. In der Softwarearchitektur-Erstellungsphase werden deshalb Softwarearchitektursichten verwendet, welche ausschließlich einen Aspekt einer Softwarearchitektur beschreiben und andere Aspekte ausblenden. Durch die Aufgliederung in Sichten ist es möglich, spezielle Aspekte verständlich darzustellen und die Softwarearchitektur an die unterschiedlichen Anforderungen der Interessengruppen anzupassen. Eine Sicht ist demnach dann ein geeignetes Abstraktionsmittel, wenn sie einen bestimmten Aspekt auf geeignete Weise darstellt und darüber hinaus allen Anforderungen der entsprechenden Interessengruppen gerecht wird.

Die Erstellung von Softwarearchitektursichten wird durch die /IEEE 1471/ empfohlen. In /Kruchten 95/, /Soni et al. 95/ und /Bass et al. 98/ werden dazu Vorschläge und Modelle für die Erstellung der Sichten gegeben.

Anforderungsverfolgung
Durch die Anforderungsverfolgung in der Architekturentwurfsphase werden die Beziehungen zwischen den Anforderungen und den Elementen der Architekturspezifikation dargestellt /Jarke 98/. Dabei sollte nach /Gotel, Finkelstein 94/ die Anforderungsverfolgung bidirektional und vollständig realisiert werden. Das bedeutet, es müssen zu jeder Komponente die relevanten Anforderungen und zu jeder Anforderung die relevanten Komponenten ermittelbar sein. Die Identifikation der relevanten Komponenten einer Anforderung wird dabei als Vorwärtsverfolgbarkeit bezeichnet. Ihre Vorteile ergeben sich bei der analytischen Qualitätssicherung /Gericke, Liggesmeyer 02/. So kann unter anderem geprüft werden, ob durch die Architekturspezifikation alle Anforderungen berücksichtigt wurden. Zudem wird erkennbar, welche Auswirkungen die Änderungen einer Anforderung auf die Architekturspezifikation besitzen. Die Anforderungsverfolgung von den Komponenten zu den relevanten Anforderungen wird als Rückwärtsverfolgbarkeit bezeichnet. Vorteile ergeben sich bei der Verständlichkeit und Änderbarkeit der Architekturspezifikation, da durch die Identifikation der relevanten Anforderungen die Intention bei der Erstellung der Komponente ersichtlicher wird.

3.7.2 Förderung der Wiederverwendung

Durch die Förderung der Wiederverwendbarkeit wird die Nutzung von bewährten und getesteten Konzepten ermöglicht. Dazu wird die Verwendung von Verhaltenstypen und Typhierarchien, Architekturmustern und Architekturstilen sowie Produktlinienarchitekturen empfohlen.

Verhaltenstypen und Typhierarchien
Typisierungen bieten die Möglichkeit, Spezifikationselemente innerhalb einer Spezifikation wiederzuverwenden /Meyer 97/ und somit eine mehrfache redundante Beschreibung zu vermeiden. In der Softwarearchitekturerstellungsphase werden Typen in Form von Verhaltenstypen /Lee, Xiong 02/ verwendet. Diese Verhaltenstypen sind Interfacespezifikationen, welche das Verhalten einer Menge von Komponenten beschreiben. Zur Veranschaulichung entspricht im objektorientierten Ansatz ein Verhaltenstyp einer Klasse und eine Komponente einem Objekt eines Verhaltenstyps.
Ein Verhaltenstyp kann weiterhin Aspekte der Interfacespezifikation von anderen Verhaltenstypen übernehmen. Dies wird als Vererbung bezeichnet. Die Vererbungsbeziehungen zwischen den Typen werden in Typhierarchien dargestellt. Der Vorteil von Typhierarchien und Vererbung liegt in der Wiederverwendung. Gleiche Aspekte werden in einer Spezifikation nur einmal definiert. Dadurch wird Redundanz vermieden und die Wartbarkeit erleichtert.

Architekturmuster und Architekturstile
Architekturmuster und Architekturstile dokumentieren Erfahrungswissen und bewährte Konzepte in der Architekturentwurfsphase. Ein Architekturmuster beschreibt dazu nach /Buschmann et al. 96/ ein wiederkehrendes Entwurfsproblem und ein generisches Schema zur Lösung dieses Problems. Im Lösungsschema werden dabei spezielle Komponenten sowie die Interaktionen zwischen diesen Komponenten dargestellt. Ein Beispiel für ein Architekturmuster ist das *Model-View-Controller*-Muster /Buschmann et al. 96/.
Im Gegensatz zum Architekturmuster beschreibt ein Architekturstil ein Interaktions-Schema zwischen Komponenten, ohne auf diese speziell einzugehen /Shaw, Garlan 96/. Ein Architekturstil beschreibt somit keine konkrete Architektur, sondern nur die Kooperationen zwischen den Komponenten. Für die Spezifikation werden Daten- und Kontrollflussbeschränkungen angegeben /Abowd et al. 95/. Beispiele für Architekturstile sind *Pipes-Filter*, *Client-Server* und *Blackboard*.
Die Vorteile von Architekturmustern und Architekturstilen liegen neben den ökonomischen Aspekten vor allem in der Vermeidung von Fehlern /Gamma et al. 95/. Diese resultiert aus der Verwendung von bereits bewährten Konzepten.

Produktlinienarchitekturen
Produktlinienarchitekturen /Jazayeri et al. 00/, /Binns et al. 96/ werden verwendet, wenn mehrere Produkte aus einem Anwendungsgebiet stammen und dadurch ähnliche Architekturen besitzen. Die gemeinsamen Bestandteile dieser Architekturen werden zu einem Referenzmodell zusammengefasst, welches in der Literatur

auch als Referenzarchitektur /Hofmeister et al. 99/ oder als anwendungsspezifische Architektur /Binns et al. 96/ bezeichnet wird. Diese Referenzmodelle stellen einen Lösungsansatz dar, der sich nach den Erfahrungen der Systemarchitekten in einem speziellen Anwendungsgebiet bewährt hat.
Wird nun ein neues System in dem Anwendungsgebiet entwickelt, so wird die Softwarearchitektur nicht von Grund auf neu entworfen, sondern es wird das Referenzmodell an das spezielle Problem angepasst. Dieser Vorgang wird auch als Ableitung bezeichnet /Jazayeri et al. 00/.
Werden bei der Entwicklung der konkreten Architektur Fehler im Referenzmodell entdeckt, so wirkt sich die Behebung der Fehler auf alle konkreten Architekturen aus. Des Weiteren entstehen bei der Entwicklung der konkreten Architekturen oft neue Anforderungen für das Referenzmodell. Werden diese in das Referenzmodell integriert, so wirkt sich dies ebenfalls auf alle konkreten Architekturen aus. Dadurch wird das Referenzmodell stetig verbessert /Bosch 01/.

3.7.3 Förderung von Sicherheitseigenschaften

Die Förderung der Sicherheitseigenschaften der Softwarearchitektur kann durch eine Separation von kritischen und nicht-kritischen Verhalten sowie durch ausgewogene sicherheitsfördernde Maßnahmen erreicht werden.

Separation von kritischen und nicht-kritischen Verhalten
In sicherheitskritischen Systemen sollte kritisches und nicht-kritisches Verhalten in unterschiedlichen Komponenten lokalisiert sein /Leveson 95/. Dadurch wird die Gefahr reduziert, dass ein Fehlverhalten einer nicht-sicherheitskritischen Komponente Auswirkung auf das sicherheitskritische Verhalten des Systems hat. Weiterhin ist es aus technischen und ökonomischen Gründen oft nicht möglich, alle Komponenten einer Architektur mit dem gleichen Aufwand zu entwickeln und zu testen /Liggesmeyer 00/. Für die Förderung der Sicherheitseigenschaften ist daher eine Fokussierung auf die kritischen Komponenten erforderlich.
Für die Bestimmung der Kritikalität einer Komponente in einer späteren Entwicklungsphase werden daher z. B Sicherheitsanforderungsstufen (SIL) /EN 50126/ verwendet. In Abhängigkeit von dieser SIL werden durch die /EN 50128/ spezielle analytische und konstruktive Qualitätssicherungsmaßnahmen gefordert.

Ausgewogene sicherheitsfördernde Maßnahmen
In softwareintensiven technischen Systemen gehen sicherheitskritische Fehlverhalten von Software- oder Hardwarekomponenten aus. Die Fehlverhalten der Softwarekomponenten beruhen dabei auf systematischen Fehlern in der Entwicklung. Die Fehlverhalten von Hardwarekomponenten dagegen beruhen zusätzlich noch auf zufälligen Fehlern und auf Materialschwächen oder Verschleißerscheinungen. Der Einsatz von sicherheitsfördernden Maßnahmen sollte beide Kategorien von Fehlverhalten ausgewogen beheben /Braband 98/. Es ist nicht sinnvoll, in einem System eine Kategorie der sicherheitskritischen Fehlverhalten besonders zu verbessern, wenn die andere dabei vernachlässigt wird. Daher muss in der Qualitätssicherung ein gesundes Verhältnis zwischen Maßnahmen zur Fehlervermeidung bei der Systementwicklung und zur Beherrschung von Hardwareausfällen angestrebt werden.

3.8 Zusammenfassung

In diesem Kapitel wurden ausgewählte Aspekte des Stands der Technik für die Spezifikation und Evaluation von Qualitätseigenschaften in der Architekturentwurfsphase beschrieben. Dazu wurde gezeigt, wie sich Aussagen über die Qualitätseigenschaften in die Strukturspezifikation und Interfacespezifikationen integrieren lassen und wie auf Basis dieser Aussagen eine Evaluation der Qualitätseigenschaften Sicherheit, Verfügbarkeit, Wartbarkeit, Zuverlässigkeit und Echtzeitfähigkeit im Architekturentwurf durchgeführt wird. Für die Architekturevaluation werden dazu Fehlerbaumanalysen, HAZOPS, HiP-HOPS und IF FMEA basierte Techniken, Markovanalysen, Zuverlässigkeitsblockdiagrammanalysen sowie Schedulinganalysen als etablierte Techniken vorgeschlagen.

4 Strukturorientierte Optimierung der Qualitätseigenschaften im Architekturentwurf

In diesem Kapitel wird ein Verfahren zur Verbesserung von Qualitätseigenschaften in der Architekturentwurfsphase beschrieben. Für dieses Verfahren wird zunächst das zugrundeliegende Prozessmodell spezifiziert und anschließend ein Automatisierungsansatz vorgestellt, welcher eine automatische Durchführung des Prozesses ermöglicht. Die Vorteile der Automatisierung liegen vor allem im ökonomischen und qualitativen Bereich. Für den Nachweis der Neuartigkeit des Automatisierungsansatzes werden, ähnliche bereits publizierte Ansätze untersucht und eine Abgrenzung zu diesen Ansätzen vorgenommen.

4.1 Der Transformationsprozess

Für die Verbesserung der Qualitätseigenschaften im Architekturentwurf wird ein in /Bosch, Mollin 99/ vorgeschlagener zyklischer Prozess verwendet. Die Voraussetzung für die Prozessanwendung ist eine funktional korrekte Architekturspezifikation, welche auf Basis einer Anforderungsspezifikation erstellt wurde. Diese funktional korrekte Architekturspezifikation wird mit Architekturevaluationen auf die Erfüllung der Qualitätsanforderungen geprüft. Wird dabei erkannt, dass Qualitätsanforderungen verletzt werden, so wird die Architektur, z. B. durch die Anwendung von Architekturmustern, transformiert. Diese Architekturtransformationen sollten ausgewählte Qualitätseigenschaften positiv beeinflussen, ohne jedoch die funktionalen Eigenschaften zu verändern. Dadurch ist die Architekturspezifikation nach der Anwendung einer Architekturtransformation weiterhin funktional korrekt. Nach jeder Architekturtransformation wird die Architekturspezifikation erneut evaluiert und der Erfolg der Architekturtransformation geprüft. Dazu müssen die Evaluationsmodelle neu erstellt oder zumindest angepasst werden. Dies ist erforderlich, da sich durch Architekturtransformationen die Strukturspezifikation und somit die Grundlage der Evaluationsmodelle verändert. Wird bei einer Architekturevaluation erkannt, dass alle Qualitätsanforderungen erfüllt sind, so terminiert der Prozess. Anderenfalls wird eine neue Architekturtransformation ausgewählt und der Zyklus wiederholt.

Da Anforderungsspezifikationen auch nicht-realisierbare oder konfliktbehaftete Anforderungen enthalten, welche erst bei der Architekturevaluation erkannt werden, ist der vorgeschlagene Prozess noch zu erweitern. Diese Erweiterung beinhaltet einen Prozessschritt, durch den die Anforderungsspezifikation geändert und somit jene nicht-realisierbaren oder konfliktbehafteten Anforderungen aus der Anforderungsspezifikation entfernt werden. Zusammenfassend ergibt sich für die Verbesserung der Qualitätseigenschaften im Architekturentwurf der in Abbildung 4-1 dargestellte Prozess.

Die Architekturtransformationen, welche in dem Prozess angewendet werden, besitzen die folgenden Charakteristika /Bosch, Mollin 99/, /Grunske 03c/:

- Die exakte Beeinflussung der zu verbessernden Qualitätseigenschaften kann vor der Anwendung der Architekturtransformationen nicht bestimmt werden, da es auf den Kontext der Architekturtransformation ankommt, auf welche Weise sich die Qualitätseigenschaften einer bestimmten Architektur verändern.
- Eine Architekturtransformation kann eine Qualitätseigenschaft verschlechtern, welche in Konflikt mit der zu verbessernden Qualitätseigenschaft steht.

Aufgrund dieser Charakteristika ist eine gezielte Auswahl der Architekturtransformationen nicht möglich. Daraus folgt, dass Architekturtransformationen probeweise durchgeführt werden. Ist eine Architekturtransformation nicht erfolgreich, so muss diese zurückgenommen und eine andere Architekturtransformation probiert werden. Dadurch wird grundsätzlich ein Suchbaum aufgespannt, der als Knoten die einzelnen erstellten Architekturspezifikationen und als Kanten die dazugehörigen Architekturtransformationen enthält. Der Prozess entspricht somit einer Optimierung und wird im Folgenden als SOQA (Strukturorientierte Optimierung der Qualitätseigenschaften im Architekturentwurf) bezeichnet.

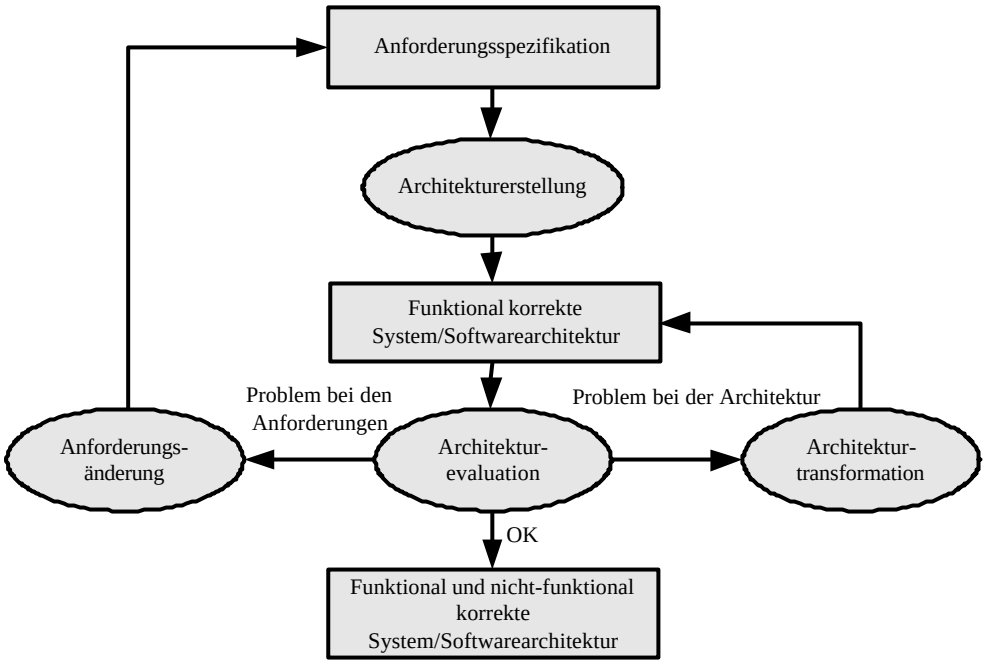


Abbildung 4-1 Transformationsprozess zur Verbesserung der Qualitätseigenschaften einer Architekturspezifikation

4.2 Automatisierungsansatz

In diesem Abschnitt wird ein Automatisierungsansatz vorgestellt, welcher die automatische Auswahl und Anwendung von Architekturtransformationen sowie die automatische Architekturevaluation innerhalb des SOQA-Prozesses ermöglicht.

4.2.1 Motivation

Aus einer Automatisierung des SOQA-Prozesses ergeben sich ökonomische und qualitative Vorteile. Die ökonomischen Vorteile resultieren aus einem kleineren personellen und zeitlichen Aufwand bei der Prozessdurchführung durch die computerunterstützte Durchführung des Prozesses. Der Aufwand wird dabei im Speziellen für die Auswahl und Anwendung der Architekturtransformationen sowie für die Architekturevaluation reduziert. Als Resultat werden die Kosten gesenkt und die Entwicklungszeit für eine Softwarearchitektur, welche ihre Qualitätsanforderungen erfüllt wird, verkürzt.
Die qualitativen Vorteile ergeben sich aus der Vergrößerung der Tiefe und Breite des Suchbaumes bei einer automatischen Prozessdurchführung und der damit verbundenen Wahrscheinlichkeit, eine bessere Softwarearchitektur zu finden. Die größere Suchtiefe ergibt sich aus einer größeren Anzahl an Transformationen, welche durch einen automatisierten und computerunterstützten Prozess durchführbar sind. Die Vergrößerung der Suchbaumbreite ergibt sich aus einer größeren Anzahl an Transformationsoperatoren, welche für die Anwendung zur Verfügung stehen. Der Grund dafür ist ein größeres Expertenwissen, da die Architekturtransformationen das Erfahrungswissen von mehreren Softwarearchitekten repräsentieren. Bei einer manuellen Anwendung kann im Gegensatz dazu oft nur auf das Erfahrungswissen eines Softwarearchitekten zurückgegriffen werden.
Zweitens wird die Qualität der resultierenden Softwarearchitektur erhöht, da durch die Formalisierung und Automatisierung der Architekturtransformationen Fehler vermieden werden. Die vermeidbaren Fehler sind in einer fehlerhaften Durchführung einer manuellen Architekturtransformation zu identifizieren. Dabei können z. B die funktionalen Eigenschaften verändert werden, womit die Softwarearchitektur nicht mehr der funktionalen Anforderungsspezifikation entspräche. Zudem werden durch die Automatisierung Fehler bei der Erstellung der Evaluationsmodelle und bei der Durchführung der Architekturevaluation vermieden. Diese Fehler verfälschen die Evaluationsergebnisse entweder positiv oder negativ. Bei einer positiven Verfälschung wird der Prozess frühzeitig beendet, ohne dass die Softwarearchitekturspezifikation ihre Qualitäts-

anforderungen erfüllt. Bei einer negativen Verfälschung werden auf Basis der Evaluationsergebnisse weitere falsche oder unnötige Transformationen durchgeführt.

4.2.2 Anforderung an die Realisierung

Die Umsetzung des automatisierten SOQA-Prozesses erfordert eine geeignete Architekturbeschreibungssprache und geeignete qualitätsverbessernde Architekturtransformationsoperatoren, welche auf eine Architekturspezifikation in der Architekturbeschreibungssprache anwendbar sind. Im Folgenden wird beschrieben, welche Anforderungen sowohl an die Architekturbeschreibungssprache als auch an einen Transformationsoperator gestellt werden.
Die Architekturbeschreibungssprache muss grundsätzlich zur Durchführung von automatisierten Architekturtransformationen und automatisierten Architekturevaluationen geeignet sein. Dieses sind die beiden Hauptanforderungen an die Architekturbeschreibungssprache. Aus der Eignung für eine automatisierte Architekturevaluation ergeben sich weitere Anforderungen. So ist es für eine Architekturevaluation notwendig zu bestimmen, ob die Architektur ihre Anforderungen einhält. Daher ist für jedes Architekturelement eine Bestimmung der relevanten Anforderungen notwendig. In der anderen Richtung müssen für alle Anforderungen die relevanten Architekturelemente bestimmbar sein. Daraus ergibt sich die Notwendigkeit, eine bidirektionale Anforderungsverfolgung in die Architekturbeschreibungssprache zu integrieren.
Die Qualitätseigenschaften eines Systems werden nicht nur durch die Software bestimmt, sondern auch durch die Hardware, auf der die Software ausgeführt wird. Dadurch ist es erforderlich, sowohl Hardware- als auch Softwarebestandteile in einer Architektur zu spezifizieren.
Die Anforderungen an die Transformationsoperatoren des automatisierten SOQA-Prozesses ergeben sich aus der Prozessbeschreibung. Dabei ist es erforderlich, dass ein Transformationsoperator automatisch auf eine Architekturspezifikation anwendbar ist. Diese Anwendung muss gegebenenfalls auch rückgängig gemacht werden können, falls sich dadurch eine Verschlechterung der Qualitätseigenschaften ergibt. Für jeden Transformationsoperator müssen Vor- und Nachbedingungen spezifiziert werden, um die gezielte Auswahl eines geeigneten und anwendbaren Transformationsoperators zu ermöglichen. Des Weiteren dürfen sich durch die Anwendung des Transformationsoperators die funktionalen Eigenschaften der Architekturspezifikation nicht verändern.
Zusammenfassend ergeben sich für den automatisierten SOQA-Prozess die folgenden Anforderungen für die Architekturbeschreibungssprache (A1 und A2) und die darauf anwendbaren Transformationsoperatoren (A3):

- A1: Die Architekturbeschreibungssprache soll die Möglichkeit bieten, Architekturtransformationen durchzuführen.
- A2: Die Evaluation der Qualitätseigenschaften des zu entwickelnden Systems soll auf Basis der Architekturbeschreibungssprache möglich und automatisierbar sein.
- A2.1: Die Hardware- und Softwarebestandteile eines softwareintensiven technischen Systems sollen gemeinsam beschreibbar sein.
- A2.2: Es soll eine bidirektionale Anforderungsverfolgung zwischen Architekturbeschreibungssprache und Anforderungsspezifikation möglich sein.
- A2.3: Die Architekturelemente lassen sich mit modularen Evaluationsmodellen für die Qualitätseigenschaften Sicherheit, Verfügbarkeit, Zuverlässigkeit, Wartbarkeit und Echtzeitfähigkeit annotieren.
- A3: Ein Transformationsoperator muss auf eine Softwarearchitektur automatisch anwendbar sein.
- A3.1: Für jeden Transformationsoperator müssen die Bedingungen spezifiziert werden, bei denen der Operator anwendbar ist.
- A3.2: Für jeden Transformationsoperator muss der erwartete Einfluss auf die Qualitätseigenschaften spezifiziert werden.
- A3.3: Ein Transformationsoperator darf die funktionalen Eigenschaften der Softwarearchitektur nicht verändern.
- A3.4: Die Anwendung der Transformationsoperatoren muss reversibel sein.

4.2.3 Skizzierung der Realisierung

Ausgehend von den Anforderungen des automatisierten SOQA-Prozesses ist für die technische Realisierung eine Formalisierung und Automatisierung der Architekturtransformation und –evaluation erforderlich. Grundlage ist daher eine Architekturspezifikation in einer formalen Architekturbeschreibungssprache. Daher wurde die Architekturbeschreibungssprache COOL (*componentbased object oriented language*) entwickelt, welche als formale Grundlage hierarchische typisierte Hypergraphen verwendet. Dabei wird jedes Element einer Architekturspezifikation als ein Element eines hierarchischen typisierten Hypergraphen spezifiziert. Als Konsequenz lassen sich Architekturtransformationen als automatisierbare Graphtransformationsregeln in der Kategorie der hierarchischen typisierten Hypergraphen darstellen. Für die Spezifikation der Vor- und Nachbedingungen werden die Graphtransformationsregeln durch prädikatenlogische Ausdrücke erweitert. Der Nachweis, dass ein Transformationsoperator die funktionalen Eigenschaften der Softwarearchitektur nicht verändert, wird nicht für jeden Transformationsoperator einzeln erbracht, sondern es wird ein Algorithmus angegeben, welcher die Verhaltensäquivalenz der Architektur vor und nach der Anwendung des Transformationsoperators nachweist. Die automatisierte Architekturevaluation erfolgt durch modellbasierte Evaluationstechniken. Dafür werden die einzelnen Elemente der Architekturspezifikation mit modularen Evaluationsmodellen annotiert, welche ausschließlich den Einfluss des Architekturelementes auf die Qualitätseigenschaften beschreiben. Dadurch ist eine kompositionsbasierte Evaluation der Qualitätseigenschaften der gesamten Softwarearchitektur möglich.
Die konkrete Realisierung der Architekturbeschreibungssprache und der Transformationsoperatoren wird in den Kapiteln 5 und 6 beschrieben.

4.3 Verwandte Ansätze

In diesem Abschnitt werden die bestehenden Ansätze zur automatischen Transformation von Spezifikationen im Softwareentwicklungsprozess vorgestellt. Dabei werden jeweils die Zielstellungen und die relevanten Bestandteile der Realisierungen der einzelnen Ansätze beschrieben. Anschließend werden die Ansätze tabellarisch gegenübergestellt und mit dem Automatisierungsansatz des automatisierten SOQA-Prozesses verglichen.
Zur besseren Strukturierung werden die verwandten Ansätze in zwei Kategorien unterteilt. Die eine Kategorie enthält allgemeine Ansätze zur automatischen Transformation von Softwarespezifikationen. Die andere Kategorie enthält Ansätze, welche sich im Speziellen mit der automatischen Transformation von Architekturspezifikationen befassen. Dabei ist anzumerken, dass die allgemeinen Ansätze zur Transformation von Softwarespezifikationen oft auch die Grundlagen zur Architekturtransformation mit einschließen. Daher sind diese Ansätze für diese Arbeit ebenso relevant wie die speziellen Ansätze zur Architekturtransformation.

4.3.1 Ansätze zur Transformation von Softwarespezifikationen

In der Literatur wurden die folgenden Ansätze zur automatischen oder automatisierbaren Transformation von Softwarespezifikationen beschrieben.

Ansatz von T. Mens, T. D'Hondt, N. van Eetvelde, S. Demeyer und D. Janssens
Tom Mens stellt in seiner Dissertation /Mens 99/ und in den darauf aufbauenden Arbeiten /Mens, Van Eetvelde 03/, /Mens et al. 02/, /Mens, D'Hondt 00/ und /Mens 00/ einen generellen Ansatz zur Evolution und Transformation der Modelle des objektorientierten Softwareentwicklungsprozesses vor. Ziel ist die Beschreibung eines einheitlichen Formalismus für die Evolution im Softwareentwicklungsprozess. Verwendet werden dazu Graphen und Graphtransformationsregeln. Dabei stellen Graphen einen modell-unabhängigen Spezifikationsformalismus und Graphtransformationsregeln einen modell-unabhängigen Transformationsformalismus dar.
Für die Spezifikation mit Graphen werden im Besonderen hierarchische typisierte Graphen betrachtet. Diese werden primär für die Spezifikation von UML-Modellen verwendet /Mens 99/, /Mens, D'Hondt 00/. Zur Beschreibung der Abbildung von Architekturspezifikationen und Programmcodespezifikationen auf hierarchische typisierte Graphen wird in /Mens 99/ und /Mens et al. 02/ ebenfalls ein Ansatz vorgeschlagen.

Die Transformationenregeln beschreiben verhaltenserhaltende Transformationen /Mens, van Eetvelde 03/, /Mens et al. 02/. Sie realisieren somit den *Refactoring*-Ansatz aus /Fowler 99/ und /Demeyer 03/. Für die Spezifikation der Transformationsregeln werden bedingte Graphtransformationsregeln, also mit Vorbedingungen erweiterte Regeln, verwendet. Dadurch ist zum einen eine gezielte Auswahl der Regeln möglich und zum anderen können Konflikte bei der parallelen und sequenziellen Regelanwendung erkannt werden. Die Erkennung von Konflikten wird durch ein spezielles Vertragskonzept /Meyer 97/ erleichtert, welches Wiederverwendungs- oder Evolutionsverträge /Mens 99/ enthält.
Zur praktischen Validation des Ansatzes wurde in /Mens, van Eetvelde 03/ ein Prototyp beschrieben, welcher in PROGRES /Schürr et al. 95/ und Fujaba /Niere, Zündorf 00/ realisiert ist. Dieser Prototyp beherrscht einfache optimierende Transformationsregeln, welche auf Java-Quellcode anwendbar sind und die Verständlichkeit verbessern.

Ansatz von U. Assmann
In /Assmann 00/ wird ein Transformationssystem zur Performanz-Optimierung von Quellcode beschrieben. Grundlage dieses Transformationssystems ist die Abbildung des Quellcodes auf typisierte Graphen. Diese Graphen werden dazu verwendet, redundante und unnötige Funktionsaufrufe zu identifizieren und mit Graphtransformationsregeln zu entfernen. Das funktionale Verhalten des Programms bleibt durch die Transformationen erhalten. Demnach beschreibt dieser Ansatz ähnlich wie der Ansatz in /Mens, Van Eetvelde 03/, /Mens et al. 02/ eine automatisierbare Anwendung von *Refactoring*-Regeln /Fowler 99/. Die praktische Anwendung des Graphtransformationssystems wird durch das Werkzeug OPTIMIX sowie einigen Fallstudien belegt.
Besonderes Augenmerk wird in /Assmann 00/ auf die parallele Anwendung von Transformationsregeln und die Findung einer Terminierungsbedingung gelegt. Für die Gewährleistung der parallelen Regelanwendung wird für die Regeln ein Abhängigkeitsgraph erzeugt. Dieser Abhängigkeitsgraph spezifiziert, welche Regeln parallel kombinierbar sind und welche Regeln bei einer parallelen Anwendung zu Konflikten führen. Für die Bestimmung der Terminierung des Transformationssystems werden mit der *termination by edge accumulation* und der *termination by subtraction* zwei Terminierungsbedingung angegeben.

Ansatz von G. Taentzer
Gabriele Taentzer beschreibt in /Taentzer 99/ und /Taentzer 01/ einen Ansatz zur visuellen Spezifikation des Verhaltens und der Evolution von verteilten Systemen. Dabei wird jedoch die Evolution zur Laufzeit des Systems betrachtet. In diesem Punkt unterscheidet sich der Ansatz von den bisher vorgestellten Ansätzen.
Für die Beschreibung der Konfiguration des Systems und für die Beschreibung des lokalen Verhaltens der einzelnen Komponenten werden graphbasierte Spezifikationstechniken verwendet. Als Notation wird dazu die Modellierungssprache UML genutzt, welche an einigen Stellen erweitert wird. Für die Beschreibung der Laufzeitevolution wird die verteilte Graphtransformation verwendet /Ehrig et al. 88/, welche auf dem algebraischen Ansatz /Ehrig 73/ aufbaut.
Die praktische Anwendbarkeit wird durch die Realisierung eines Prototypen in AGG /Taentzer et al. 99/, sowie durch die Anwendung des Ansatzes in Fallstudien mit einem verteilten Versionsmanagementsystem /Taentzer 99/ und einer verteilten Datenbank /Taentzer et al. 99/ gezeigt.

Ansatz von A. Agrawal, T. Levendovszky, J. Sprinkle, F. Shi und G. Karsai
Der Ansatz der Mitarbeiter des *Institute for Software Integrated Systems (ISIS)* an der *Vanderbilt University* in Nashville fokussiert auf die Modelltransformation /Agrawal et al. 02,03/ im Rahmen des *MDA*-Projektes (*model driven architecture*) der OMG /OMG 03/. Unter Modelltransformation wird dabei zum einen die Codegenerierung aus einem Modell und zum anderen die Überführung eines Modells in einen anderen Modelltyp verstanden. Ersteres wird als vertikale und zweiteres als horizontale Modelltransformation bezeichnet /Agrawal et al. 03/. Zur Durchführung der Modelltransformation wird das Modell zunächst in einer Graphstruktur spezifiziert. Dazu wird für jeden Modelltypen ein Meta-Modell bereitgestellt, welches eine eindeutige Zuordnung zwischen den Elementen des Modells und den Graphelementen zulässt.
Basierend auf der Graphstruktur ist eine Überführung des Modells in den Zielmodelltypen mit Hilfe von Graphtransformationsregeln möglich. Als Beispiel wird in /Agrawal et al. 03/ die Modelltransformation von Statecharts /Harel 88/ in endliche Zustandsmaschinen präsentiert.
Das Besondere am vorgestellten Ansatz sind neben der Transformation zwischen verschiedenen Modelltypen vor allem die Transformationsregeln. Diese ermöglichen es, hierarchische und strukturgenerische Graphen zu identifizieren und zu transformieren. Dies erhöht die Mächtigkeit des Graphtransformationssystems.

Ansatz von R. France, S. Ghosh, E. Song, D.-K. Kim
In /France et al. 03/ wird ein Transformationsansatz vorgestellt, welcher sich ebenfalls auf das *MDA*-Projekt (*model driven architecture*) der OMG /OMG 03/ bezieht. Dabei werden horizontale Modelltransformationen betrachtet, welche die funktionalen Eigenschaften der Spezifikation nicht beeinflussen, aber zur einer Verfeinerung der Architektur führen. Als Beispiel für diese Modelltransformationen werden die GoF-Muster /Gamma 95/ verwendet. Für die Anwendung der Transformationen wird jedoch explizit kein formaler Ansatz präsentiert.

4.3.2 Ansätze zur Transformation von Architekturspezifikationen

In der Literatur wurden die folgenden Ansätze zur automatischen oder automatisierbaren Transformation von Architekturspezifikationen identifiziert.

Ansatz von R. A. Riemenschneider, M. Moriconi, X. Qian
Der Ansatz in /Moriconi et al. 95/ beschreibt ein Verfahren zur schrittweisen Verfeinerung von abstrakten Softwarearchitekturen zu konkreteren, implementierungsnaheren Softwarearchitekturen unter Bewahrung der funktionalen Korrektheit. Dieser Ansatz wird in /Riemenschneider 99/ erneut diskutiert und ergänzt. Grundlage ist die Beschreibung einer Architekturspezifikation mit prädikatenlogischen Formeln. Verwendet wird dazu die Prädikatenlogik erster Ordnung. Für die Verfeinerung einer Architekturspezifikation werden Muster (*refinement patterns*) verwendet, welche als *interpretation mapping* spezifiziert werden. Ein solches *interpretation mapping* beschreibt dabei die Abbildung der Formeln der Theorie der abstrakten Architektur auf Formeln der Theorie der verfeinerten Architektur.
Der Fokus der Arbeit liegt auf der Korrektheit der Architekturverfeinerungen. Dabei wird für jedes Verfeinerungsmuster ein Beweis geführt, welcher die funktionale Korrektheit allgemein, d.h. unabhängig von einer konkreten Anwendung nachweist. Dadurch wird das Ziel *correctness by construction* der formalen Methoden auch für Softwarearchitekturen anwendbar.
Zum Nachweis der praktischen Realisierbarkeit werden in /Moriconi et al. 95/ eine Reihe von korrekten Verfeinerungsmustern vorgestellt. Diese wurden auf ein reales System (200 KLOC, Fortran 77, Stromversorgungssektor) angewendet.

Ansatz von D. Le Metayer
Daniel Le Metayer beschreibt in /Le Metayer 96,98/ primär einen Ansatz zur Formalisierung von Architekturstilen. Die Idee seines Ansatzes ist, eine Softwarearchitektur als Graph zu beschreiben, bei dem die Knoten aktive Komponenten und die Kanten die Kommunikationsbeziehungen zwischen diesen Komponenten beschreiben. Ausgehend von der Spezifikation von Softwarearchitekturen als Graphen wird ein Architekturstil als eine Menge von Graphen betrachtet, welche sich durch eine Graphgrammatik erzeugen lassen.
In den Arbeiten /Le Metayer 96,98/ werden grundsätzlich noch keine Architekturtransformationen beschrieben. Sie stellen jedoch den Ausgangspunkt für die im Folgenden vorgestellten Ansätze dar.

Ansatz von D. Hirsch, U. Montanari und P. Inverardi
Dan Hirsch, Ugo Montanari und Paola Inverardi beschreiben in /Hirsch, Montanari 99,00/ und /Hirsch et al. 99/ einen Ansatz zur Architekturrekonfiguration von verteilten Systemen. Im Speziellen werden Kommunikationssysteme und deren grundsätzliche Komponenten, wie z. B. Client, Server, Router und Bridges betrachtet. Ziel der Rekonfigurationen ist, die Architektur zu verfeinern und dabei einen Architekturstil zu befolgen. In diesem Punkt ähnelt der Ansatz der Arbeit von Le Metayer / Le Metayer 98/.
Für die Modellierung der Softwarearchitektur werden Hypergraphen verwendet. Dabei entsprechen Komponenten Hyperkanten und die Kommunikation zwischen den Komponenten erfolgt über Knoten. Als Formalismus zur Architekturrekonfiguration werden kontextfreie Hyperkantenersetzungen genutzt. Diese Hyperkantenersetzungen spezifizieren das Regelsystem einer Graphgrammatik, welches, wie in / Le Metayer 98/ dargestellt, einen Architekturstil beschreibt.

Ansatz von H. Fahmy und R. C. Holt
Ziel des in /Holt 98/ und /Fahmy, Holt 00/ vorgeschlagenen Ansatzes ist, die Komplexität und Verständlichkeit von Softwarearchitekturen zu verbessern. Dazu werden einfache Transformationen präsentiert, welche vor allem die Kopplungs- und Bindungseigenschaften verbessern. Diese Transformationen werden in

/Fahmy, Holt 00/ als bedingte Graphtransformationsregeln dargestellt und prototypisch in PROGRES /Schürr et al. 95/ realisiert. In /Holt 98/ wird für die Repräsentation der gleichen Regeln ein anderer Ansatz gewählt. Dabei wird die Architektur als Graph unter Verwendung von algebraischen Tarski-Operatoren spezifiziert. Die Transformationen der Architektur werden durch algebraische Relationen beschrieben, welche durch einen Relationsinterpreter mit dem Namen GROK /Holt 02/ umgesetzt werden.
Die praktische Anwendbarkeit des Ansatzes wird in /Holt 98/ anhand einiger Fallstudien (250-300 KLOC COBOL- und C-Programme) untermauert.

Ansatz von M. Wermelinger, A. Lopes, J. L. Fiadeiro
Ziel der Arbeiten von Michel Wermelinger, Antónia Lopes und José Luiz Fiadeiro ist die Formalisierung der dynamischen Architekturrekonfiguration zur Laufzeit eines Systems /Wermelinger, Fiadeiro 99/, /Wermelinger et al. 01/ und /Wermelinger, Fiadeiro 02/. Besonders betrachtet werden dabei verteilte Systeme.
Für die Spezifikation einer Architektur wird ein algebraischer Ansatz und die Architekturbeschreibungssprache CommUnity verwendet. Die möglichen Veränderungen der Architektur zur Laufzeit werden durch einfache Transformationen beschrieben. Als Beispiel für solche Transformationen wird das Hinzufügen, das Entfernen und das Verfeinern von Komponenten sowie die Zuweisung von Kommunikationsvariablen dargestellt. Für die Anwendung dieser Transformationen wird der *double-pushout* Ansatz /Ehrig 79/ verwendet. Neben den einfachen Transformationen wird in /Wermelinger et al. 01/ eine Möglichkeit präsentiert, komplexe Transformationen zu spezifizieren. Diese komplexen Transformationen beschreiben die strukturierte Ausführung der einfachen Transformationen ähnlich einer Programmiersprache.

Ansatz von C. Ermel, R. Bardohl und J. Padberg.
In /Ermel et al. 01/ und /Padberg 03/ wird ein formales Rahmenwerk für die Architekturevolution vorgestellt. Grundlage für dieses Rahmenwerk ist die gemeinsame Spezifikation der Softwarearchitektur und deren Implementierung (Verhaltensspezifikation) als Graph. Die Spezifikation der Evolution erfolgt mit dem HLR (*high level replacement system*) /Ehrig et al. 93/ und *double-pushout*-Graphtransformationsregeln.
Die Umsetzung des Ansatzes wurde mit Hilfe des Werkzeuges GenGED /Bardohl et al. 03/ und einem akademischen Beispiel gezeigt.

Tabelle 4-1 Gegenüberstellung der relevanten Softwareevolutionsansätze

Merkmale / Ansätze	Allgemeines		Spezifikationsnotation und Transformation				Prozess- und Werkzeugunterstützung			Anwendung
	Veröffentlichungstermin	Intention des Ansatzes	Spezifikations-Formalismus	Spezielle Merkmale des Spezifikations-Formalismus	Transformations-formalismus	Spezielle Merkmale des Transformations-formalismus	Auswahl der Transformationen	Evaluation des Transformationserfolgs	Werkzeugeinsatz	
T. Mens, et al.	99, 00, 02, 03	Entwicklung der formalen Grundlagen für die Evolution von objektorientierten Modellen	Graphen	Hyperkanten, Typisierung, Hierarchisierung	HGE	SPO und DPO Typvariabilität, Vorbedingungen	Eingeschränkte Auswahl über Vorbedingungen		Prototyp realisiert in Fujaba /Niere, Zündorf 00/ und PROGRES /Schürr et al. 95/	UML Modelle, in Ansätzen Architekturspezifikationen und Quellcode
U. Assman	00	Performanz Optimierung von Programmen	Graphen	Typisierung	GE	DPO, Parallelität der Regeln, Terminierungskriterien	Durch LCM-Analyse (lazy code motion)	Siehe Auswahl der Transformation	OPTIMIX	Quellcode
G. Taentzer	99, 01	Beschreibung eines visuellen Konzeptes für die Modellierung und Evolution von verteilten Systemen	Graphen	Typisierung	GE	Verteilte Graphtransformationen			Prototyp realisiert in AGG /Taentzer et al. 99/	UML (mit Erweiterungen)
A. Agrawal et al.	03	Modelltransformation und Codegenerierung im Rahmen des MDA-Projektes der OMG	Graphen	Typisierung Hierarchisierung	GE	SPO, Strukturvariabilität			GME	UML Modelle (GReAT)
R. France	03	Überblick über die Anwendung von Entwurfsmustern und Refactoring im Rahmen des MDA-Ansatzes	Nichtformalisiert		Nichtformalisiert	DPO ähnlicher Ansatz				UML-Modelle, Quellcode

Legende:
SPO Single Pushout Ansatz
DPO Double Pushout Ansatz
(H)KE (Hyper-)Kantenersetzung
KE Knotenersetzung
(H)GE (Hyper-)Graphsetzung

Tabelle 4-2 Gegenüberstellung der relevanten Architekturevolutionsansätze

Merkmale / Ansätze	Allgemeines		Spezifikationsnotation und Transformation				Prozess- und Werkzeugunterstützung			Anwendung
	Veröffentlichungstermin	Intention des Ansatzes	Spezifikations-Formalismus	Spezielle Merkmale des Spezifikations-Formalismus	Transformations-formalismus	Spezielle Merkmale des Transformations-formalismus	Auswahl der Transformationen	Evaluation des Transformations-erfolgs	Werkzeug-einsatz	
R. A: Riemenschneider et al.	95, 99	Beschreibung von korrekten Architekturverfeinerungen	Prädikatenlogik		Theorie Interpretationen (*interpretation mapping*)	Nachweis der Regelkorrektheit *(correctness by construction)*				Architekturspezifikationen
D. Le Métayer	96, 98	Formalisierung von Architekturstilen durch Graphgrammatiken	Graphen	Hyperkanten	Vorgeschlagen werden HKE, KE und GE.					Architekturspezifikationen
H. Fahmy, R. C. Holt	98, 00, 01	Verbesserung der Verständlichkeit einer Architekturspezifikation	Graphen+Tarski Relationen	Typisierung, Hierarchisierung	GE	SPO			Prototyp realisiert in PROGRES /Schürr et al. 95/	Architekturspezifikationen (*box-and-line diagrams*)
M. Wermelinger	99, 01, 02	Formalisierung der dynamischen Architekturrekonfiguration zur Laufzeit eines Systems	Graphen (Algebraische Spezifikationen)	Typisierung	GE	DPO				Architekturspezifikationen (Grundlage ist die ADL CommUnity)
D. Hirsch et al.	99, 00	Rekonfiguration von verteilten Systemen unter Beibehaltung eines Architekturstils	Graphen	Hyperkanten, Typisierung	HKE	SPO				Architekturspezifikationen für Kommunikationssystem
J. Padberg et al.	01, 03	Erstellung eines generellen Frameworks zur Spezifikation von Modellevolutionen	Graphen (Algebraische Spezifikationen)	Typisierung	GE	DPO (HLR)				Architekturspezifikationen (Kombiniert mit den Implementierungsspezifikationen)

Legende:

SPO Single Pushout Ansatz
DPO Double Pushout Ansatz
HLR High Level Replacement System
(H)KE (Hyper-)Kantenersetzung
KE Knotenersetzung
(H)GE (Hyper-)Graphsetzung

4.3.3 Abgrenzung zu den bestehenden Ansätzen

Ausgehend von den Ergebnissen der Analyse der verwandten Ansätze ist erkennbar, dass in den letzten Jahren ein großer Aufwand in die Formalisierung und Automatisierung von Software- und Architekturtransformationen investiert wurde. Dabei wurden hauptsächlich Graphen und Graphtransformationssysteme genutzt. In diesem Punkt besteht Übereinstimmung zwischen den bestehenden Ansätzen und dem Automatisierungsansatz des SOQA-Prozesses. Außerdem besteht Übereinstimmung mit den Ansätzen in /Mens 99/, /Assmann 00/ und /Fahmy, Holt 00/ in dem Punkt, dass diese ebenfalls verhaltenserhaltende Transformationen verwenden.
Unterschiede ergeben sich vor allem im Anwendungsgebiet, da keiner der bisherigen Ansätze auf Architekturspezifikationen für softwareintensive technische Systeme angewendet wurde. Dadurch ergeben sich vor allem Unterschiede bei der Architekturbeschreibungssprache und der Abbildung der einzelnen Architekturelemente auf die Graphelemente der Spezifikation. Ein weiterer Unterschied ist das Ziel, die Qualitätseigenschaften Sicherheit, Verfügbarkeit, Zuverlässigkeit, Wartbarkeit und Echtzeitfähigkeit zu verbessern. Ein ähnliches Ziel wurde bei keinem existierenden Ansatz identifiziert.
Neuerungen beim Automatisierungsansatz des SOQA-Prozesses ergeben sich bei der Auswahl der Transformationen und bei der Überprüfung des Erfolgs einer Transformation. Dabei werden die Transformationen auf Basis der Evaluationsergebnisse ausgesucht. Nur wenn die in den Nachbedingungen spezifizierten Qualitätsanforderungen nicht erfüllt werden, wird ein Transformationsoperator ausgewählt. Daraus ergibt sich ebenfalls eine Terminierungsbedingung für den Transformationsprozess, welcher terminiert, wenn keine Transformationsoperatoren mehr anwendbar sind. Dies ist der Fall, wenn eine Architekturspezifikation mit den optimalen Qualitätseigenschaften gefunden wurde oder wenn alle Qualitätsanforderungen erfüllt sind. Die Überprüfung des Erfolgs einer Transformation wird ebenfalls durch die Architekturevaluation bestimmt. Dabei werden die Qualitätseigenschaften, also die Ergebnisse der Architekturevaluation, vor und nach der Anwendung der Architekturtransformation verglichen. Haben sich die Qualitätseigenschaften bei der Anwendung der Architekturtransformation verbessert, so war die Architekturtransformation erfolgreich.

4.4 Zusammenfassung

Die Zielsetzung dieser Forschungsarbeit ist die Entwicklung eines Verfahrens, welches die Qualitätseigenschaften eines softwareintensiven technischen Systems in der Architekturentwurfsphase bestimmt, verbessert und optimiert.
Dazu wurde in diesem Kapitel der SOQA-Prozess vorgestellt, welcher die strukturorientierte Optimierung der Qualitätseigenschaften einer Architekturspezifikation ermöglicht und somit der Zielsetzung der Dissertation entspricht. Des Weiteren wurde ein Ansatz für die automatische und computerunterstützte Durchführung des Prozesses präsentiert. Dieser Automatisierungsansatz schlägt Hypergraphen zur Spezifikation einer Softwarearchitektur und Hypergraphtransformationen zur Spezifikation von Architekturtransformationen vor.
Für eine vollständige Realisierung des Verfahrens ist eine geeignete Architekturbeschreibungssprache und ein geeigneter Formalismus zur Spezifikation und Anwendung von Transformationsoperatoren erforderlich. Diese Aspekte werden in den folgenden beiden Kapiteln beschrieben.

5 Die Architekturbeschreibungssprache COOL

In diesem Kapitel wird eine Architekturbeschreibungssprache vorgestellt, welche den Anforderungen des automatisierten SOQA-Prozesses entspricht. Die Idee der Architekturbeschreibungssprache ist es, Hypergraphen als einheitliches Konzept zur Spezifikation einer Architektur zu verwenden. Dadurch lassen sich Architekturtransformationen als Hypergraphtransformationsregeln spezifizieren und automatisch auf Architekturen in der Architekturbeschreibungssprache anwenden.
Im Folgenden werden daher zunächst die Grundlagen der Hypergraphtheorie vorgestellt. Dabei werden insbesondere die Typisierung und die Hierarchisierung betrachtet. Anschließend wird gezeigt, wie sich eine Architekturbeschreibungssprache inklusive der Evaluationsmodelle auf dieses Konzept abbilden lässt.

5.1 Einführung in die Hypergraphtheorie

5.1.1 Allgemeine Hypergraphen

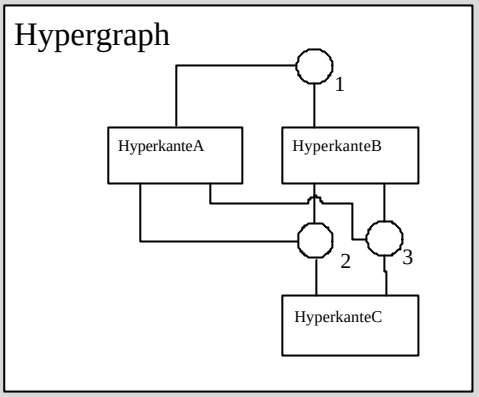
Hypergraphen sind eine Verallgemeinerung von Graphen, bei denen eine Kante mit mehr als zwei Knoten verbindbar ist /Berge 89/. Dadurch lassen sie sich universeller einsetzen /Habel 92/ und sind speziell für die Beschreibung von komplexen Strukturen, wie z. B Softwarearchitekturen, geeignet /Grunske 03c, 03e/. Formal werden Hypergraphen nach /Habel 92/ wie folgt definiert.

Definition 5.1 Allgemeine Hypergraphen
Sei L_V eine Menge von Knotentypen (*node types*) und L_E eine Menge von Hyperkantentypen (*edge types*), dann wird ein Hypergraph G aus der Menge der bildbaren Hypergraphen G über L_V und L_E durch das Tupel $\langle V, E, att, lab \rangle$ beschrieben. Dabei repräsentiert V eine Menge von Knoten und E eine Menge von Hyperkanten. Die Vereinigung von Knoten und Hyperkanten wird als Menge der Atome $A = V \cup E$ des Hypergraphen bezeichnet. Über die Funktion $att : E \rightarrow V^*$ wird einer Hyperkante eine Knotensequenz V^* zugeordnet. Die Funktion $lab : A \rightarrow L_V \cup L_E$ weist den einzelnen Knoten und Hyperkanten einen Knoten- oder Hyperkantentyp zu.

Durch die Zuweisungsfunktion att ist es möglich, einer Hyperkante eine Sequenz von Knoten zuzuweisen. Dabei wird die Anzahl n als Arität der Hyperkante bezeichnet. Sie lässt sich für eine Hyperkante $e \in E$ aus der Länge der zugeordneten Knotensequenz $|att(e)|$ bestimmen.
Hypergraphen, bei denen alle Hyperkanten identische Aritäten besitzen, werden als uniform bezeichnet. Ein Beispiel für einen uniformen Hypergraphen ist ein herkömmlicher Graph. Dieser besitzt die Hyperkantenarität zwei.
Aufbauend auf der Definition eines allgemeinen Hypergraphen wird ein Hypergraph mit gerichteten Kanten durch eine Separierung der Funktion $att : E \rightarrow V^*$ in zwei Funktionen $source: E \rightarrow V^*$ und $target: E \rightarrow V^*$ definiert. Dabei werden durch die Funktion *source* die Quellknoten der Hyperkante und durch die Funktion *target* die Zielknoten der Hyperkante bestimmt.
Für die graphische Darstellung von Hypergraphen werden im Folgenden Vierecke als Hyperkanten und Kreise als Knoten verwendet. Die Typen werden bei den Hyperkanten in die Vierecke und bei Knoten an die Kreise geschrieben. Die Reihenfolge in der Sequenz der verknüpften Knoten einer Hyperkante wird durch eine gegen den Uhrzeigersinn erfolgende Nummerierung beschrieben, welche um zwölf Uhr beginnt. Bei einem gerichteten Hypergraphen werden die Quellknoten oberhalb der Hyperkante und die Zielknoten unterhalb der Hyperkante dargestellt.

Beispiel 5-1

Zur Veranschaulichung wird im Folgenden ein Hypergraph mit drei Hyperkanten dargestellt. Die Hyperkanten *A* und *B* sind mit drei Knoten assoziiert und haben daher die Arität drei. Die Hyperkante *C* ist eine normale binäre Hyperkante zwischen den Knoten 2 und 3.



5.1.2 Modulare Hypergraphen

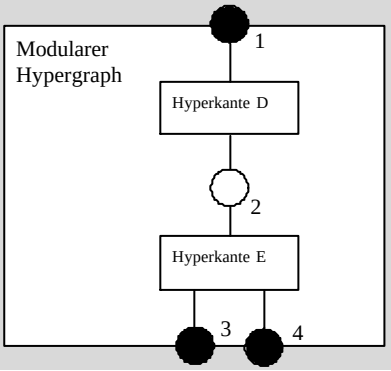
Für die Definition von hierarchischen Hypergraphen bilden modulare Hypergraphen (*multi-pointed hypergraph*s) die Grundlage. Ein solcher modularer Hypergraph besitzt externe Knoten, welche zur Verbindung mit anderen Hypergraphen genutzt werden. Die externen Knoten stellen somit Klebepunkte für den modularen Hypergraphen dar. Formal werden modulare Hypergraphen wie folgt definiert.

Definition 5.2 Modulare Hypergraphen
Ein modularer Hypergraph wird durch das Tupel $\langle V, E, att, lab, ext \rangle$ charakterisiert. Dabei entspricht $\langle V, E, att, lab \rangle$ einem Hypergraphen und ext beschreibt eine Sequenz von externen Knoten $ext \in V^*$.

Die Arität n eines modularen Hypergraphen wird durch die Länge der Sequenz der externen Knoten $|ext|$ bestimmt. In Abhängigkeit der Arität n werden modulare Hypergraphen auch als (n)-Hypergraphen bezeichnet. Nach /Drewes et al. 02/ sind Hypergraphen eine Untermenge der modularen Hypergraphen. Sie entsprechen modularen (0)-Hypergraphen.
Alle Knoten eines Hypergraphen, welche nicht in ext enthalten sind, werden als interne Knoten int bezeichnet.
Modulare Hypergraphen eignen sich ebenfalls zur Beschreibung von gerichteten Graphen. Dazu wird die Sequenz der externen Knoten in die Sequenzen von eingehenden und ausgehenden externen Knoten aufgegliedert. Ist bei einem solchen modularen Hypergraphen die Arität der eingehenden Knoten n und die Arität der ausgehenden externen Knoten m, so wird dieser als (n, m)-Hypergraph bezeichnet.
Bei der graphischen Darstellung werden ausgefüllte Kreise als Repräsentation von externen Knoten verwendet.

Beispiel 5-2

Ein modularer Hypergraph kann wie folgt dargestellt werden:



5.1.3 Hierarchische Hypergraphen

Die bereits dargestellten allgemeinen und modularen Hypergraphen besitzen eine flache Struktur. Für die Abbildung von Software- und Systemarchitekturen ist es jedoch erforderlich, Hypergraphen hierarchisch zu strukturieren /Grunske 03c/. Dies führt zu einer Reduktion der Komplexität der einzelnen Graphen /Tapken 99/. Des Weiteren wird die Anwendung von Abstraktionskonzepten ermöglicht, da kontextabhängig bestimmte Hierarchieebenen ausblendbar sind.
Ein Hierarchiekonzept in Hypergraphen kann nach /Hoffmann, Minas 01/ und /Drewes et al 02/ durch die Verfeinerung von Hyperkanten realisiert werden. Dabei wird in einem Hypergraph eine Menge von Hyperkanten definiert, welche zur Einbettung von modularen Hypergraphen genutzt werden. Für die Einbettung werden Hyperkantenrahmen (*frames*) erzeugt, welche wie folgt definiert werden.

Definition 5.3 Hyperkantenrahmen
Ein Hyperkantenrahmen (*frame*) beschreibt eine Sequenz von Knoten V^*. Für jede Hyperkante kann ein solcher Hyperkantenrahmen über die Zuweisungsfunktion $att : E \rightarrow V^*$ ermittelt werden.

Basierend auf der Definition von Hyperkantenrahmen und modularen Hypergraphen lassen sich hierarchische Graphen wie folgt definieren.

Definition 5.4 Hierarchische Hypergraphen
Ein hierarchischer Hypergraph G aus der Menge aller bildbaren hierarchischen Hypergraphen G über die Typen L_V und L_E wird durch das Tupel $\langle V, E, att, lab, ext, cts \rangle$ charakterisiert. Dabei ist $\langle V, E, att, lab, ext \rangle$ ein modularer Hypergraph und $cts : E \rightarrow G$ eine Zuweisungsfunktion, welche einer Hyperkante einen eingebetteten hierarchischen Hypergraphen zuordnet.

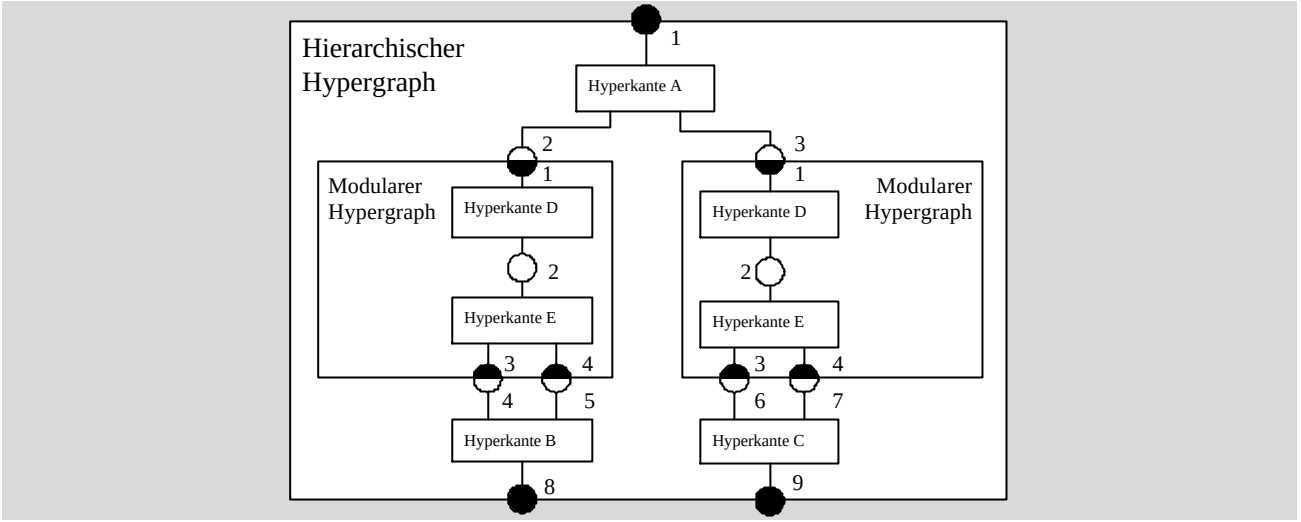
Für die Einbettung wird die Hyperkante aus dem Graphen entfernt und der Hyperkantenrahmen mit dem enthaltenen Hypergraphen verbunden. Die Knoten des Hyperkantenrahmens und die externen Knoten des eingebetteten Hypergraphen stellen somit die Schnittstelle zwischen den beiden Graphen dar und werden auch als Ports bezeichnet /Drewes et al. 02/, /Hoffmann, Minas 01/. Dadurch werden ausschließlich Hyperkanten innerhalb eines modularen Hypergraphen erforderlich und hierarchieüberschreitende Hyperkanten vermieden.
Nach der Definition wird die Struktur eines hierarchischen Hypergraphen induktiv über die Hierarchieebenen beschrieben. In der untersten Hierarchieebene G_0 sind ausschließlich nicht-hierarchische, modulare Hypergraphen enthalten. Bei diesen ist $cts(e) = \varnothing$ für alle $e \in E$. In den höheren Hierarchieebenen $G_{\geq 1}$ werden durch den Definitionsbereich der Funktion $cts : E \rightarrow G$ Hyperkanten bestimmt, in welche modulare hierarchische Hypergraphen eingebettet werden.

Beispiel 5-3

Zur Veranschaulichung wird im Folgenden ein hierarchischer Hypergraph dargestellt, welcher zwei modulare Hypergraphen einbettet. Diese eingebetteten Hypergraphen entsprechen den modularen Hypergraphen aus dem vorherigen Beispiel. Für die Integration enthält der hierarchische Hypergraph zwei Hyperkanten, welche die Knoten 2,4,5 sowie 3,6,7 verbinden.

Die Funktion $cts : E \rightarrow G$ weist diesen Hyperkanten jeweils einen eingebetteten modularen Hypergraphen zu. Die assoziierten Knoten 2-7 der verfeinerten Hyperkanten sind dabei sowohl im hierarchischen als auch im eingebetteten Graphen enthalten und besitzen daher zwei Bezeichner. Des Weiteren sind die Knoten für den eingebetteten Graphen externe Knoten und für den hierarchischen Graphen interne Knoten. Daher sind sie halb als interne und halb als externe Knoten markiert.

Es ist erkennbar, dass der hierarchische Hypergraph wiederum ein modularer Hypergraph ist. Somit kann dieser ebenfalls in komplexere hierarchische Hypergraphen eingebettet werden.



Neben der Einbettung von hierarchischen Hypergraphen in Hyperkantenrahmen wird in /Pratt 79/, /Busatto, Hoffmann 01/ und /Engels, Schürr 95/ die Knotenverfeinerung vorgeschlagen. Dabei enthalten Knoten hierarchische Hypergraphen. Für den Zusammenhalt des Graphen müssen bei der Knotenverfeinerung spezielle Hyperkanten mit den Knoten des eingebetteten Hypergraphen verbunden werden. Diese Hyperkanten werden auch als intermodulare Kanten bezeichnet /Engels, Schürr 95/. Diese intermodularen Kanten führen zwar zu einer flexiblen Graphstruktur, widersprechen jedoch dem Kapselungsprinzip /Drewes et al 02/. Daher sollte für die Architekturspezifikation die dargestellte Kantenverfeinerung verwendet werden.

5.1.4 Typisierte Hypergraphen und Vererbungshierarchien

Bei der Definition von Hypergraphen wurden bereits Knoten- und Hyperkantentypen eingeführt. In Anlehnung an die objektorientierte Modellierung lassen sich diese Typen durch Klassen beschreiben. Dadurch wird jeder Knoten- und jeder Hyperkantentyp durch eine Menge von Attributen und Operationen spezifiziert. Neben den Knoten und Hyperkanten sind auch Hypergraphen durch Klassen typisierbar. Die enthaltenen Knoten und Hyperkanten eines Hypergraphen werden durch Komposition der Hypergraphklasse zugeordnet. Durch Einführung von Hypergraphklassen sind auch komplexe Hyperkanten modellierbar, welche einen Hypergraphen enthalten. Dies ist für die Spezifikation der in Definition 5.4 beschriebenen hierarchischen Hypergraphen notwendig. Zusammenfassend wird ein typisierter hierarchischer Hypergraph durch das in Abbildung 5-1 dargestellte Meta-Modell beschrieben.

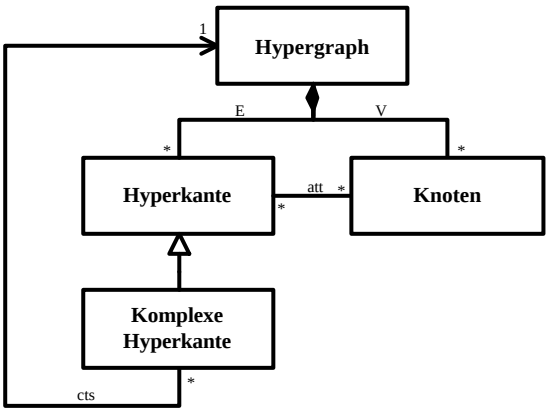


Abbildung 5-1 Meta-Modell eines hierarchischen typisierten Hypergraphen

Ausgehend von diesem Meta-Modell lassen sich anwendungsspezifische Klassen durch Vererbung erzeugen. Diese erweitern die Meta-Klassen um anwendungsspezifische Attribute und Operationen. Ein Beispiel für solche anwendungsspezifischen Attribute sind z. B. die Position oder die Farbe des instanziierten Knotens oder der instanziierten Hyperkanten. Ein Beispiel für eine anwendungsspezifische Operation ist das Verschieben eines Knotens oder einer Hyperkante.

Durch das Konzept der Vererbung entstehen zwischen den Klassen Vererbungshierarchien. Diese Vererbungshierarchien werden durch partielle Ordnungsrelationen beschrieben /Mens 99/ und wie folgt definiert.

Definition 5.5 Vererbungshierarchien
Die Vererbungshierarchien der Kontentypen L_V, Kantentypen L_E und Hypergraphtypen $G \in G$ werden durch partielle Ordnungsrelationen $|_V$, $|_E$ und $|_G$ beschrieben, wobei $y \mid x$ besagt, dass der Typ y ein Subtyp des Typen x ist und somit alle Eigenschaften des Typen x erbt.

Beispiel 5-4

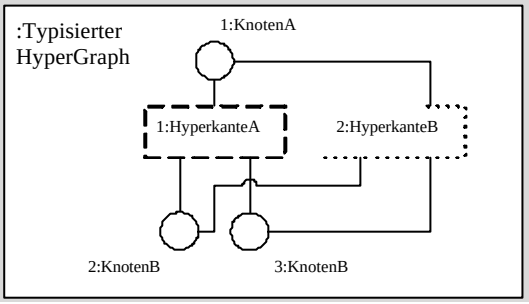
Zur Darstellung der Typisierung und der Vererbungshierarchien wird im Folgenden ein Hypergraph dargestellt, welcher aus einem Knoten des Typs KnotenA und zwei Knoten des Typs KnotenB besteht. Dabei sind die Knotentypen von dem Basisknotentypen abgeleitet:

KnotenA $|_V$ Knoten und KnotenB $|_V$ Knoten

Der Hypergraph enthält zudem jeweils eine Hyperkante des Typs HyperkanteA und HyperkanteB. Diese besitzen die folgenden Vererbungsbeziehungen:

HyperkanteA $|_E$ Hyperkante und HyperkanteB $|_E$ Hyperkante

Die Hyperkantentypen HyperkanteA und HyperkanteB sind jeweils durch ein anwendungsspezifisches Attribut erweitert. Dieses weist einer Hyperkante vom Typ HyperkanteA eine gestrichelte Linie und einer Hyperkante vom Typ HyperkannteB eine gepunktete Linie in der graphischen Darstellung zu.



5.2 Strukturspezifikation

5.2.1 Anwendung von Hypergraphen zur Strukturspezifikation

Für die Anwendung von Hypergraphtransformationen auf eine Softwarearchitekturspezifikation ist es erforderlich, die Strukturspezifikation auf das Konzept der Hypergraphen abzubilden. Dadurch wird eine formale Strukturspezifikation für Softwarearchitekturen möglich /Grunske 03c/.
Für die Architekturbeschreibungssprache COOL wurde dazu das in Abbildung 5-1 dargestellte Meta-Modell eines hierarchischen typisierten Hypergraphen um eine Meta-Ebene erweitert. Diese Meta-Ebene beschreibt die Bestandteile einer Softwarearchitektur /Hofmeister et al. 99/ und bildet die Klassen des hierarchischen typisierten Hypergraphen auf die Elemente der Softwarearchitektur ab. Dies wird durch Vererbungsbeziehung modelliert, da die Elemente der Softwarearchitektur alle Eigenschaften der Hypergraphelemente besitzen und diese gegebenenfalls erweitern sollen. Die zusätzliche Meta-Ebene wird als Softwarearchitektur-Meta-Ebene bezeichnet. Die Hypergraph- und die Softwarearchitektur-Meta-Ebenen der COOL werden in Abbildung 5-2 zusammen dargestellt.
In der Softwarearchitektur-Meta-Ebene werden mit den Basisklassen und den Architekturklassen zwei Klassentypen unterschieden. Die Basisklassen sind direkt von den Klassen der Hypergraph-Meta-Ebenen abgeleitet und werden zur Strukturierung der Vererbungshierarchie eingesetzt. Die Architekturklassen werden von diesen Basisklassen abgeleitet. Sie sind in Abbildung 5-2 grau hinterlegt und dienen bei der Erstellung einer konkreten Architektur zur Ableitung der anwendungsspezifischen Architekturelemente. Sie stellen somit das Vokabular der Architekturbeschreibungssprache COOL dar.

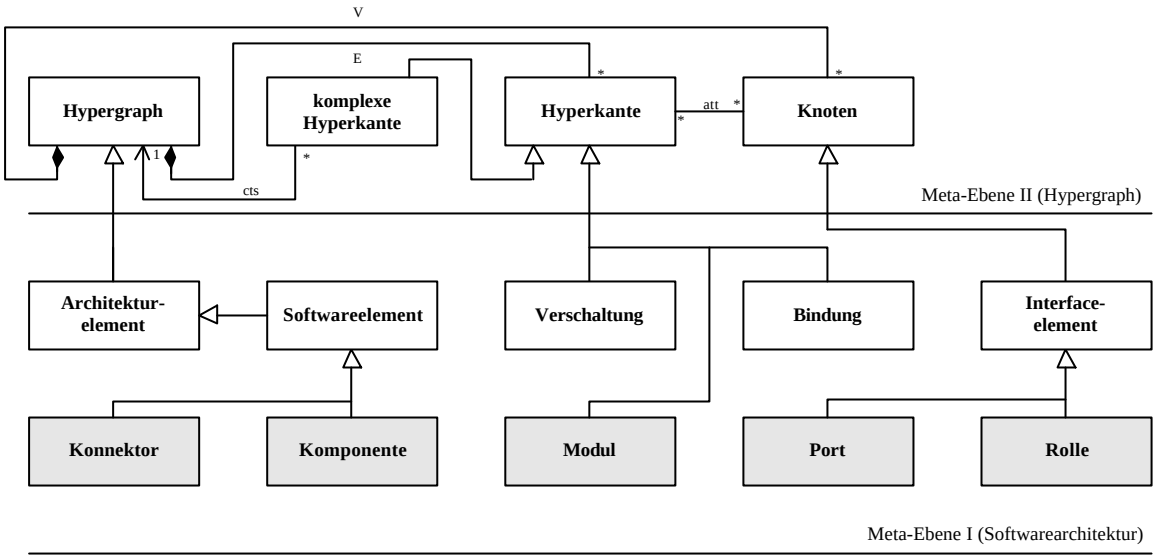


Abbildung 5-2 Softwarebestandteile des COOL-Meta-Modells

Im COOL-Meta-Modell ist erkennbar, dass die Meta-Klasse *Hypergraph* durch die Meta-Klasse *Architekturelement* und *Softwareelement* verfeinert wird. Die Metaklasse *Softwareelement* wird wiederum durch die Meta-Klassen *Komponente* und *Konnektor* verfeinert, wodurch die Strukturspezifikation grundlegend einem Komponent-Konnektor-Modell /Hofmeister et al. 99/ entspricht. Für die Interaktion mit der Umgebung wird in diesem Modell jeder Komponente eine Menge von Ports und jedem Konnektor eine Menge von Rollen zugeordnet. Im COOL-Meta-Modell werden die Meta-Typen *Port* und *Rolle* von der Meta-Klasse *Interfaceelement* abgeleitet, welche wiederum von der Meta-Klasse *Knoten* abgeleitet ist. Da jedes Softwareelement von der Meta-Klasse *Hypergraph* abgeleitet ist, enthält es durch die Komposition *V* eine endliche Menge von *Interfaceelementen*.

Zur Strukturspezifikation in der Architekturbeschreibungssprache COOL wird zudem zwischen flachen und hierarchischen Architekturelementen unterschieden. Dabei beschreibt ein flaches Architekturelement einen Hypergraphen, welcher nur eine Hyperkante enthält. Diese Hyperkante wird durch die Meta-Klasse *Modul* spezifiziert und verbindet alle Interfaceelemente des flachen Architekturelementes.

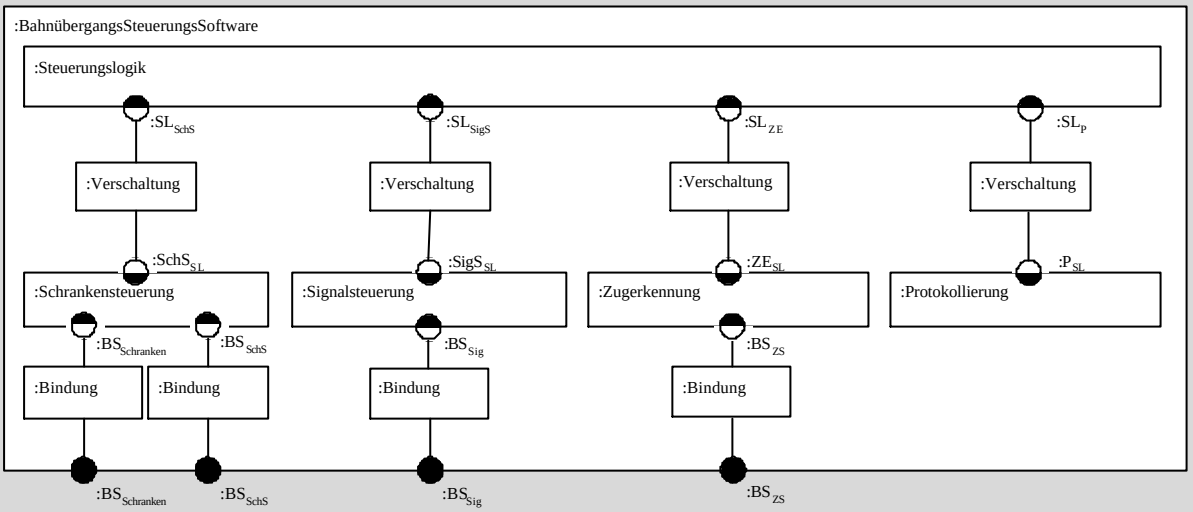
Hierarchische Architekturelemente enthalten für die Spezifikation von Kommunikations- und Kompositionsbeziehungen mehrere Hyperkanten. Für die Kompositionsbeziehungen wird mit der Meta-Klasse *komplexe Hyperkante* eine spezielle Hyperkante beschrieben, welche durch einen Hypergraphen ersetzbar ist. Dadurch lassen sich sowohl Komponenten als auch Konnektoren in hierarchische Architekturelemente einbetten. Die hierarchische Zerlegung eines Architekturelementes wird somit durch die Kompositionshierarchie des Graphen beschrieben. Die Blattknoten dieser Kompositionshierarchie sind flache Architekturelemente. Die Wurzel der Kompositionshierarchie ist eine spezielle Komponente, welche als Ursprungskomponente (*genesis component*) bezeichnet wird. Ausgehend von dieser Ursprungskomponente wird rekursiv das Softwaresystem instanziiert. Dies erfolgt durch die in der Hypergraph-Meta-Ebene enthaltenen Kompositionen *V* und *E*.

Für die Beschreibung der Kommunikationsbeziehungen zwischen den Komponenten und Konnektoren werden die Meta-Klassen *Verschaltung* und *Bindung* von der Meta-Klasse *Hyperkante* abgeleitet. Dabei beschreibt die Hyperkante *Verschaltung* eine Kommunikationsbeziehung zwischen den Interfaceelementen von Architekturelementen in einer Hierarchieebene. In /Hofmeister et al. 99/ wird dabei ausschließlich eine Verschaltung eines Ports mit einer Rolle erlaubt. Dadurch wird eine strikte Strukturspezifikation erreicht, bei der Komponenten nur über Konnektoren miteinander kommunizieren. In der Architekturbeschreibungssprache COOL wird auf diese Einschränkung zur besseren Modellierbarkeit der Softwarearchitektur verzichtet. Eine Verschaltung kann auch Ports mit Ports und Rollen mit Rollen verbinden. Dadurch sind auch Strukturspezifikationen ohne Konnektoren möglich.

Durch die Hyperkante *Bindung* werden Kommunikationsbeziehungen zwischen Softwareelementen auf unterschiedlichen Hierarchieebenen modelliert. Dabei wird ein interner Knoten eines eingebetteten Softwareelementes mit einem externen Knoten eines umschließenden Softwareelementes verbunden.

Beispiel 5-5

In diesem Beispiel wird eine Softwarearchitektur für eine einfache Bahnübergangssteuerung modelliert. Diese Softwarearchitektur wird durch die verfeinerte Komponente *BahnübergangsSteuerungsSoftware* veranschaulicht. Die Komponente enthält die Komponenten *Schrankensteuerung*, *Signalsteuerung*, *Zugerkennung*, *Protokollierung* und *Steuerungslogik*. Diese Komponenten sind als komplexe Hyperkanten mit den entsprechenden Verweisen auf die Hypergraphen dargestellt. Für die Kommunikation mit anderen Komponenten enthalten die Komponenten Ports, welche Knoten im Graphen sind. Verbunden sind diese Ports durch Hyperkanten vom Typ *Verschaltung*. Für die Kommunikation mit externen Komponenten enthält die Komponente *BahnübergangsSteuerungsSoftware* Ports, welche als externe Knoten modelliert werden. Verbunden sind diese Knoten mit den Knoten der restlichen Architektur über Hyperkanten vom Typ *Bindung*.



5.2.2 Erweiterung der Strukturspezifikation zur Beschreibung von Hardwarebestandteilen

Bei der Modellierung einer Architektur eines softwareintensiven technischen Systems sind neben den Softwareelementen auch Hardwareelemente zu spezifizieren /Douglass 99/, /Gomma 00/, /Liggesmeyer 99/, da diese einen Einfluss auf die Qualitätseigenschaften des Systems, wie z.B. die Zuverlässigkeit, Performanz und Sicherheit haben. In die Architekturbeschreibungssprache COOL werden daher die folgenden Hardwareelemente integriert:

- Sensor
- Aktor
- Hardwareplattform
- Übertragungskanal

Die Sensoren und Aktoren repräsentieren die Schnittstellen zur Systemumwelt. Sie werden als Nachrichtenquellen (Sensoren) und Nachrichtensenken (Aktoren) modelliert.

Die Hardwareplattform ist für die Ausführung der Softwaresysteme verantwortlich. Daher ist eine Hardwareplattform als Verbund von Prozessoren, Bussystemen, Speicherbausteinen, Stromversorgung und Peripheriegeräten sowie den notwendigen Softwarebasissystemen zu sehen. Ein Übertragungskanal verbindet die einzelnen im System befindlichen Hardwareplattformen. Er besteht aus dem Übertragungsmedium und den bereitgestellten Kommunikationsmechanismen.

Für die Integration der Hardwareelemente ist es erforderlich, das Meta-Modell zu erweitern. Diese Erweiterungen sind in Abbildung 5-3 in der Hardware-Meta-Ebene dargestellt. Es ist erkennbar, dass die Meta-Klasse *Hardwareelement* zusätzlich von der Meta-Klasse *Architekturelement* abgeleitet wird. Sie dient als Superklasse für die vier Meta-Klassen *Sensor*, *Aktor*, *Hardwareplattform* und *Übertragungskanal*. Durch die Ableitung von der Meta-Klasse *Architekturelement* sind die Hardwareelemente ebenso hierarchisch strukturierbar wie die Softwareelemente. Daher lassen sich Hardwareelemente und Softwareelemente ähnlich modellieren.

Darüber hinaus kann auf Basis des in Abbildung 5-3 dargestellten Meta-Modells spezifiziert werden, auf welchem Hardwareelement ein Softwareelement ausgeführt wird. Dafür werden mit der Hyperkante *Ausführungsverschaltung* und dem Knoten *Ausführungsknoten* zusätzliche Meta-Klassen eingeführt. Jedes Softwareelement muss einen Ausführungsknoten enthalten und einem Hardwareelement wird ein Ausführungsknoten zugewiesen, sobald auf ihm Softwareelemente ausgeführt werden. Die Ausführungsverschaltung verbindet die Ausführungsknoten eines Software- und eines Hardwareelementes miteinander. Dadurch ist eine eindeutige Zuordnung eines Softwareelementes zu einem Hardwareelement möglich.
Zur gemeinsamen Modellierung der Hardware- und Softwarebestandteile einer Architektur wird im Meta-Modell mit der Meta-Klasse *System* eine weitere Meta-Klasse von *Architekturelement* abgeleitet. Diese Meta-Klasse kann sowohl Hardware- als auch Softwareelemente enthalten und ist daher geeignet, softwareintensive technische Systeme zu beschreiben.

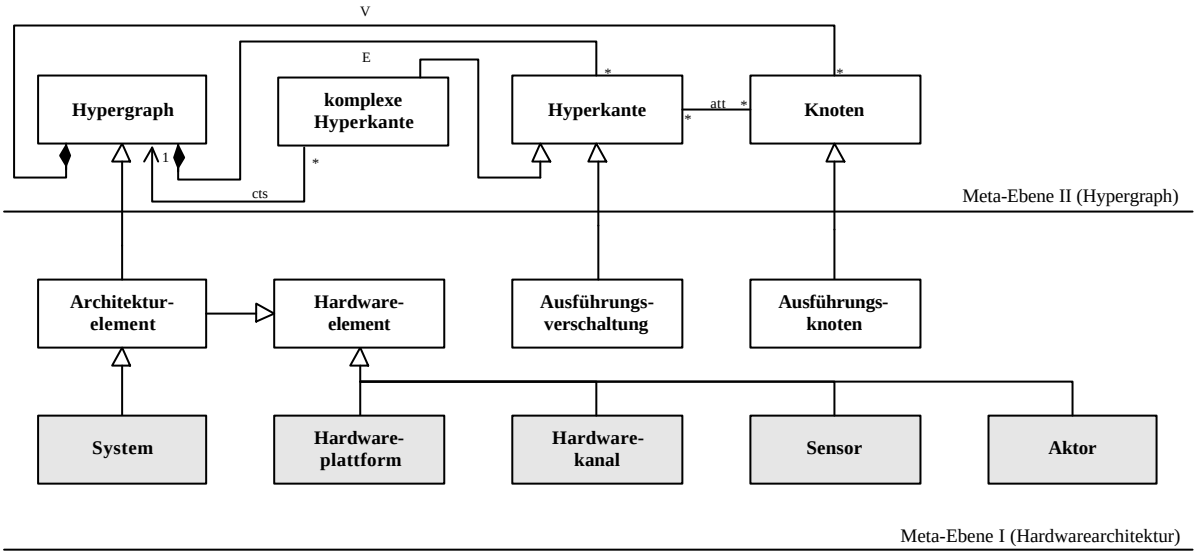
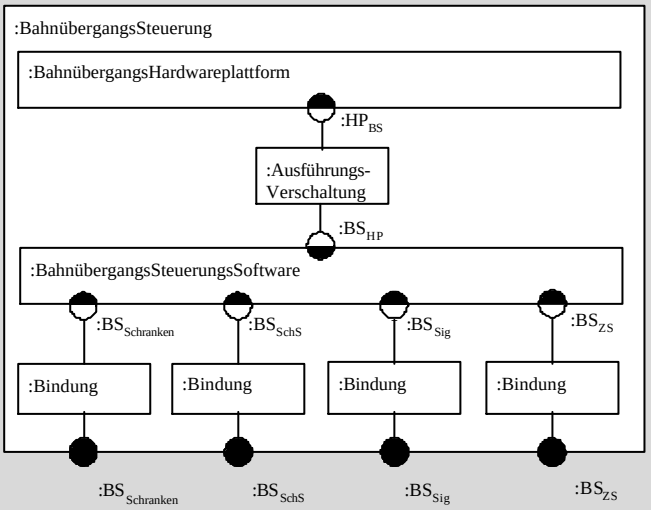
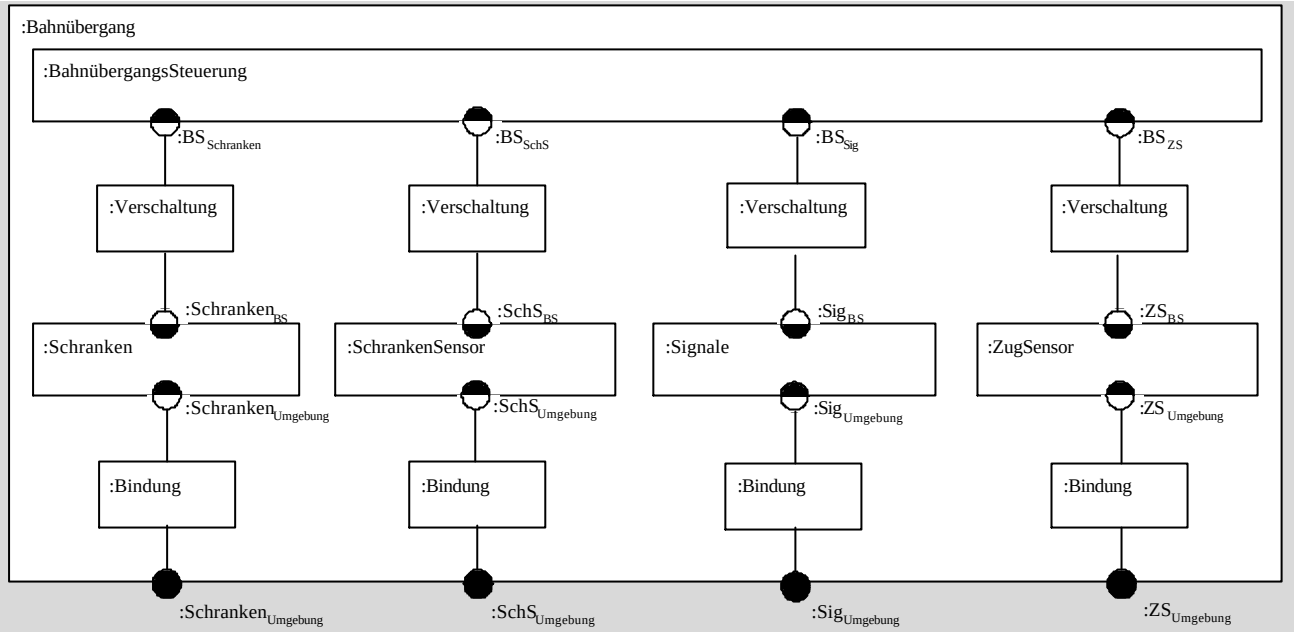


Abbildung 5-3 Hardwarebestandteile des COOL-Meta-Modells

Beispiel 5-6

Zur Veranschaulichung der Spezifikation einer Architektur mit Hardware und Softwareelementen wird das vorherige Beispiel und die Softwarekomponente *BahnübergangsSteuerungsSoftware* erweitert. Dabei wird ein System *BahnübergangsSteuerung* erstellt, welches die *BahnübergangsSteuerungsSoftware* und eine *BahnübergangsHardwareplattform* enthält, auf welcher die Softwarekomponente ausgeführt wird. Das Architekturelement *BahnübergangsSteuerung* wird im zweiten Hypergraphen mit den notwendigen Aktoren und Sensoren verbunden. Diese Sensoren und Aktoren entsprechen nicht weiter verfeinerten Hypergraphen.

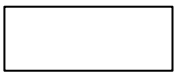
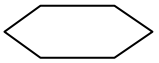
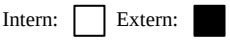




5.2.3 Graphische Notation

Die Spezifikation von Softwarearchitekturen mit der bisher verwendeten Notation für Hypergraphen ist für Softwarearchitekten gewöhnungsbedürftig. Daher wird für die Strukturspezifikation mit der Architekturbeschreibungssprache COOL eine neue graphische Notation vorgeschlagen. Die Notation ordnet jedem COOL-Element, wie in Tabelle 5-1 dargestellt, ein architekturspezifisches grafisches Symbol zu. Die verwendeten Symbole wurden dabei in Anlehnung an /Hoffmeiser et al. 99/ und /Selic 94/ gewählt.

Tabelle 5-1 Elemente der graphischen Notation in der ADL COOL

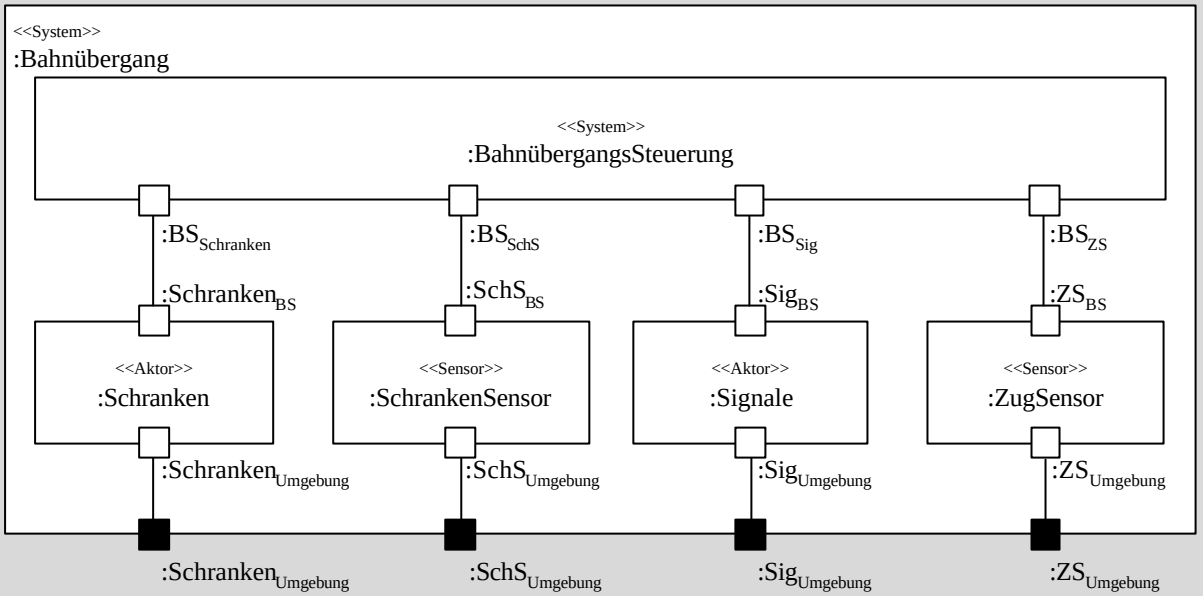
Hypergraph Element	COOL Element	Graphisches Symbol
Hypergraph	Komponente, Sensor, Aktor, Hardwareplattform, Architektur, Hardware und Softwareelement	
Hypergraph	Konnektor, Hardwarekanal	
Knoten	Port, Abwicklerknoten und Ausführungsknoten	
Knoten	Role	
Hyperkante	Verschaltung, Bindung, Ausführungsverschaltung	

Da mehrere COOL Metaklassen identische graphische Symbole besitzen, ist es erforderlich, die graphische Notation für die COOL-Elemente weiter zu differenzieren. In Anlehnung an die UML werden dazu Stereotypen verwendet. Die Stereotypnamen sind dabei identisch mit den Namen der Metaklasse, von welchem das Architekturelement erbt. Somit werden in der Architekturbeschreibungssprache COOL die folgenden Stereotypen verwendet: <<Architekturelement>>, <<Softwareelement>>, <<Hardwareelement>>, <<System>>, <<Komponente>>, <<Konnektor>>, <<Hardwareplattform>>, << Hardwarekanal >>, <<Sensor>> und <<Aktor>>. Stereotypen für abgeleitete Knoten- und Kantentypen werden nicht benötigt, da diese eindeutig aus der Strukturspezifikation bestimmbar sind. Als Beispiel dafür sind Kanten zwischen internen und ex-

ternen Knoten stets Bindungen, Kanten zwischen Hardwareelementen oder Kanten zwischen Softwareelementen sind Verschaltungen und Kanten zwischen einem Softwareelement und einem Hardwareelement sind Ausführungsverschaltungen.

Beispiel 5-7

Auf Basis der graphischen Notation für die Architekturbeschreibungssprache COOL kann die Architektur aus dem vorherigen Beispiel wie folgt spezifiziert werden.



5.3 Verhaltensspezifikation

Wie bereits in Kapitel 3 dargestellt, wird das Verhalten eines Architekturelementes in der Architekturentwurfsphase durch eine Interfacespezifikation beschrieben. Diese Interfacespezifikationen beschreiben das extern beobachtbare Verhalten eines Architekturelementes. Als Spezifikationsnotation werden in dieser Arbeit Interfaceautomaten verwendet. Die Auswahl wird durch die folgenden Eigenschaften der Interfaceautomaten begründet:

1. Für Interfaceautomaten existieren geeignete Algorithmen zur Überprüfung der Kompatibilität von Interfacespezifikationen und der korrekten Verfeinerung /de Alfaro et al. 01/.
2. Interfaceautomaten besitzen eine verständliche graphische Repräsentation /de Alfaro et al. 01/.
3. Interfaceautomaten eignen sich für die Modellierung von ereignisgesteuerten Systemen /Lee 02/
4. Interfaceautomaten eignen sich zur Spezifikation von Verhaltenstypen /Lee, Xiong 02/. Damit ist es auf Basis der Interfacespezifikation wie in /Meyer 99/ und /Liskov, Wing 94/ möglich, Verhaltenssubtypen zu definieren, was der strikten Vererbungsstruktur der Architekturbeschreibungssprache COOL entgegen kommt.
5. Durch die vorgeschlagenen Erweiterungen in /de Alfaro et al. 02/ sind Interfaceautomaten auch für die Beschreibung des temporalen und stochastischen Verhaltens geeignet. Dies ist für die Analyse der Qualitätseigenschaften notwendig.
6. Für Interfaceautomaten existiert ein Konstruktionsalgorithmus, welcher eine Interfacespezifikation für ein hierarchisches Architekturelement auf Basis der Interfacespezifikationen der enthaltenen Architekturelemente erzeugt /Grunske 03d/.

Für die Integration der Verhaltensspezifikation mit Interfaceautomaten wird im Folgenden dargestellt, wie sich die Interfaceautomaten auf gerichtete typisierte Hypergraphen abbilden lassen und wie die Integration in die Architekturbeschreibungssprache COOL erfolgt. Anschließend wird gezeigt, wie ein Interfaceautomat eines hierarchischen Softwareelementes konstruiert wird.

5.3.1 Abbildung von Interfaceautomaten auf Hypergraphen

Zur Vereinheitlichung des Spezifikationsformalismus werden die Interfaceautomaten auf einen gerichteten Hypergraphen abgebildet. Auf Grund der Struktur eines Interfaceautomaten werden dazu gerichtete typisierte Hypergraphen verwendet. Eine Hierarchisierung ist nicht erforderlich, da bei Interfaceautomaten die Zustände nicht hierarchisch verfeinert werden.
Die konkrete Zuordnung der spezifischen Klassen eines Interfaceautomaten zu den Klassen eines gerichteten typisierten Hypergraphen wird in Abbildung 5-4 dargestellt. Dabei werden die Zustände des Interfaceautomaten als Knoten und die Ereignisse als Hyperkanten dargestellt. Da ein Ereignis in einem Interfaceautomaten nur einen Quell- und einen Zielknoten haben kann, enthält der Hypergraph ausschließlich binäre Hyperkanten.

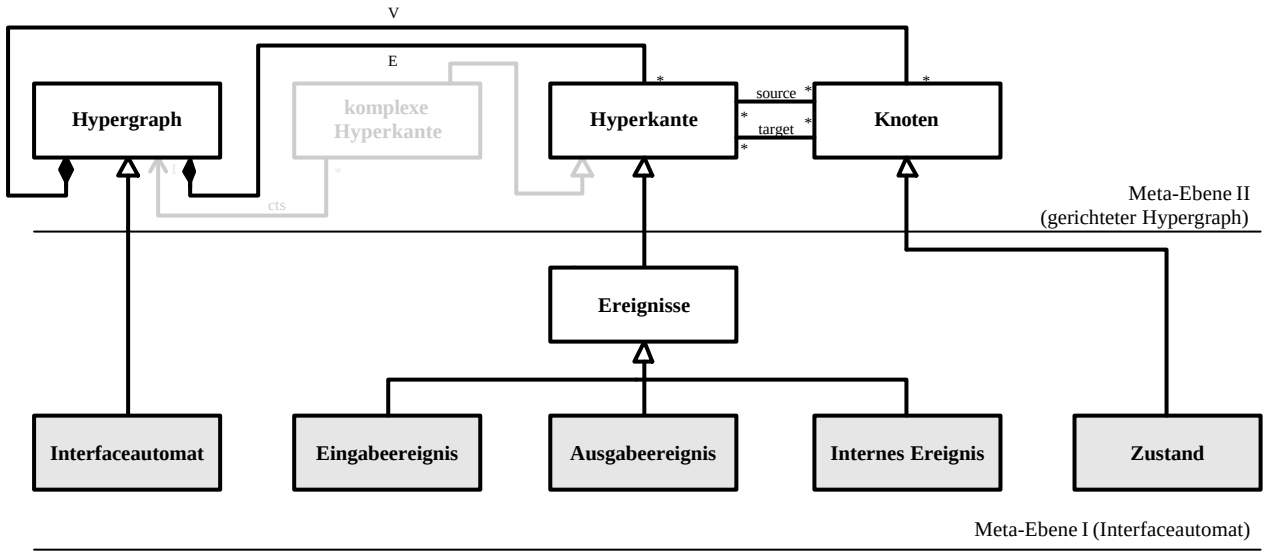


Abbildung 5-4 Abbildung von Interfaceautomaten auf gerichtete typisierte Hypergraphen

5.3.2 Integration von Interfaceautomaten in die Architekturbeschreibungssprache

Für die Integration der Interfaceautomaten in die Architekturbeschreibungssprache COOL wird die Meta-Klasse *Architekturelement* um das Attribut *Interfacespezifikation* erweitert. Dieses Attribut charakterisiert eine Referenz auf einen Interfaceautomaten. Als Erweiterung ist aber auch denkbar, andere Interfacespezifikationen in anderen Notationen zu referenzieren. Dadurch ist die Architekturbeschreibungsprache COOL nicht an die Interfaceautomaten gebunden.
Zusätzlich zu den Interfacespezifikationen wird in der Architekturbeschreibungssprache COOL auch spezifiziert, welche Ereignisse über welche Interfaceelemente mit der Umgebung ausgetauscht werden. Dies erfolgt durch die Erweiterung der Meta-Klasse *Interfaceelement* um zwei Attribute, welche die Menge der eingehenden und ausgehenden Ereignistypen beschreiben.

5.3.3 Spezifikation von Interfaceautomaten für hierarchische Architekturelemente

Durch die im vorherigen Abschnitt beschriebenen Erweiterungen der Architekturbeschreibungssprache COOL kann das Verhalten jedes Architekturelementes mit einem Interfaceautomaten spezifiziert werden. Aus ökonomischen Gründen ist die Spezifikation jedoch nur für flache Architekturelemente sinnvoll, da sich das Verhalten eines hierarchischen Elementes aus dem Verhalten der enthaltenen Elemente ergibt. Daher sind auch die Interfaceautomaten für hierarchische Architekturelemente aus den Interfaceautomaten der enthaltenen Architekturelemente konstruierbar. Verwendet wird ein Algorithmus, welcher sukzessive die Interfaceautomaten der enthaltenen Architekturelemente komponiert. Für die Komposition wird das in 3.3.2 dargestellten Verfahren zur Komposition von Interfaceautomaten verwendet. Der komplette Algorithmus für die Erstellung von Interfaceautomaten für Architekturelemente sieht wie folgt aus:

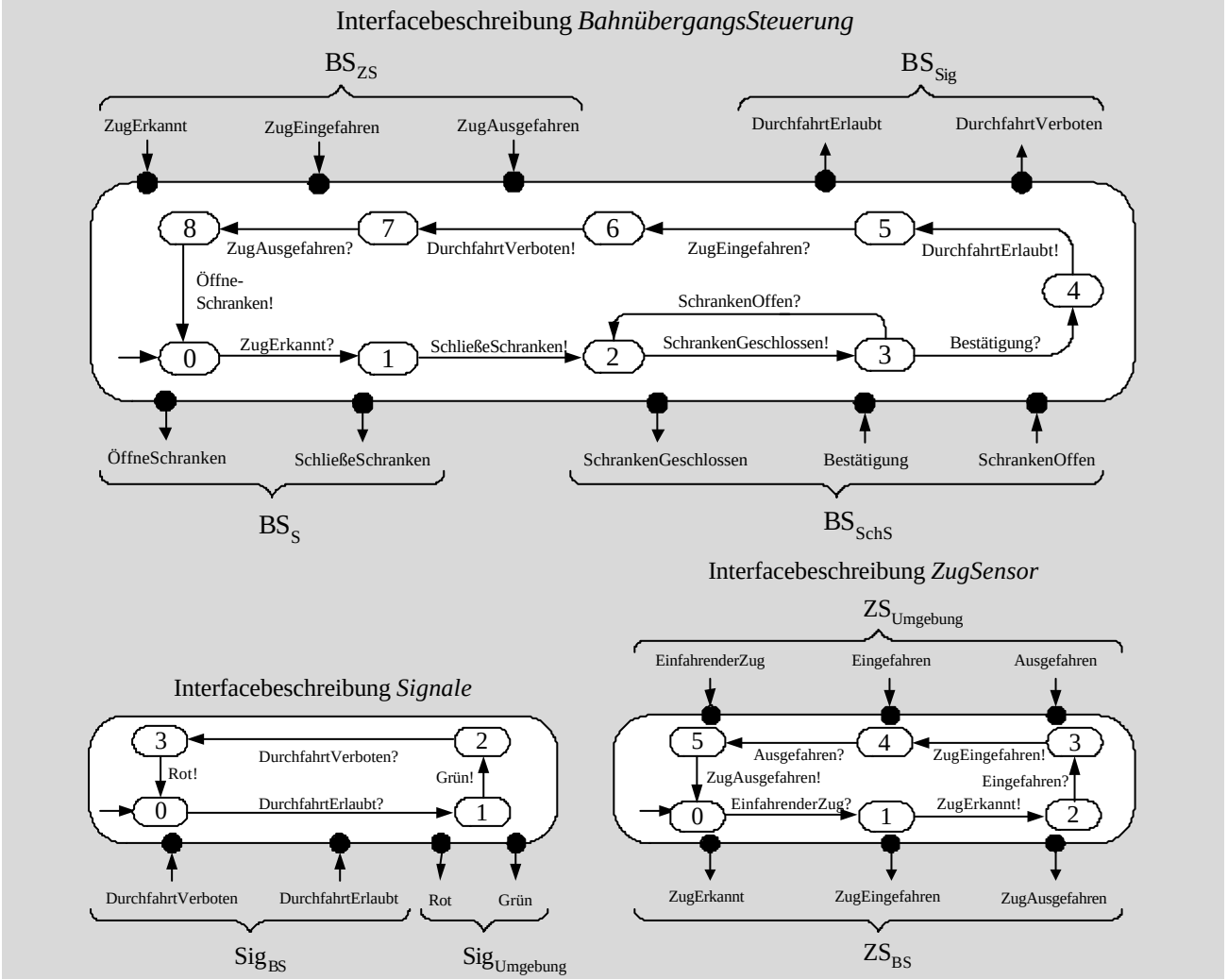
Algorithmus erstelleInterfaceAutomat(k)

Input: Architekturelement k, für das die Verhaltensspezifikation erzeugt werden soll
Output: Verhaltensspezifikation des Architekturelementes k

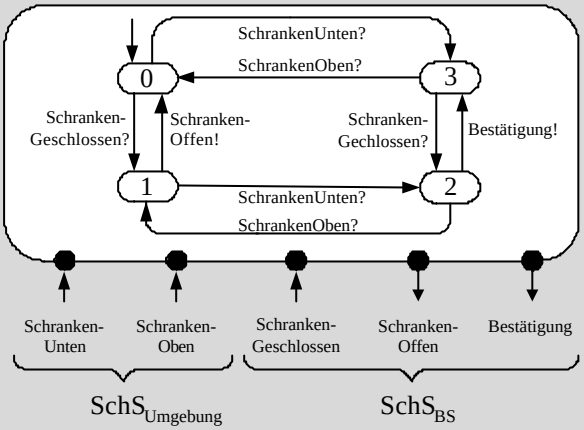
1. Wenn k keine Architekturelemente enthält, dann gebe den Interfaceautomaten des Architekturelementes zurück.
2. Sonst, bestimme alle enthaltenen Architekturelemente ks_n und erzeuge einen neuen Interfaceautomaten *result* für das betrachtete Architekturelement *k* und weise diesem den Interfaceautomaten des ersten enthaltenen Architekturelementes ks_1 zu.
3. Iteriere über alle verbleibenden Architekturelemente ks_n
   a. Bestimme den Interfaceautomaten ks_n.*Interfaceautomat* für das Architekturelement ks_n rekursiv mit der Funktion *erstelleInterfaceAutomat*(ks_n)
   b. Bestimme alle gemeinsamen Ereignisse (*sharedAction*) von ks_n und *result* durch die Iteration über alle Verschaltungen.
   c. *result* := KompositionInterfaceautomat(ks_n.*Interfaceautomat, result, sharedAction*)
4. Gebe *result* zurück

Enthält ein Architekturelement wiederum hierarchische Architekturelemente, so werden deren Interfaceautomaten im Schritt 3a nach demselben Algorithmus konstruiert. Somit lässt sich rekursiv für jedes hierarchische Architekturelement der dazugehörige Interfaceautomat generieren.
Als Einschränkung für die Anwendung des Konstruktionsalgorithmus ist es erforderlich, dass ein Ereignistyp nur in einem assoziierten Interfaceelement verwendet wird.

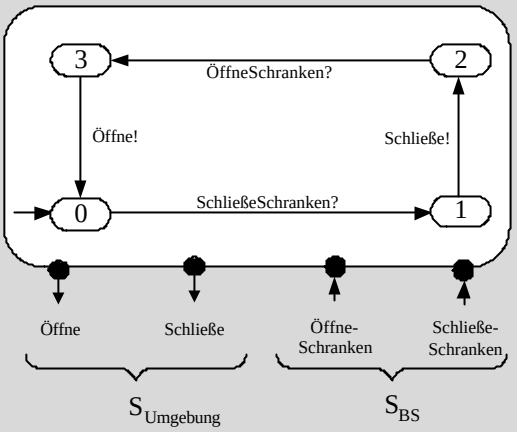
Beispiel 5-8 Das Beispiel zeigt die Verhaltensspezifikation für die Komponenten des bereits illustrierten Bahnübergangsbeispiels.



Interfacebeschreibung *SchrankenSensor*



Interfacebeschreibung *Schranken*

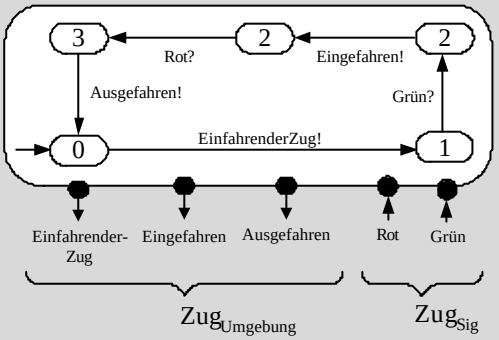


Ausgehend von diesen Interfaceautomaten ergibt sich für das Architekturelement Bahnübergang ein Interfaceautomat mit 3456 Zuständen. Diese Anzahl ergibt sich aus dem Produkt der Zustandsanzahlen der einzelnen Interfaceautomaten. Auf Grund dieser Komplexität ist der Interfaceautomat der Bahnübergangssteuerung für einen Menschen schwer konstruier- und interpretierbar. Die Erstellung und Interpretation des Interfaceautomaten lässt sich demnach nur mit Werkzeugunterstützung beherrschen.

Zur Vereinfachung des Interfaceautomaten für die Bahnübergangssteuerung werden in diesem Beispiel die folgenden Annahmen über die Umgebung getroffen:

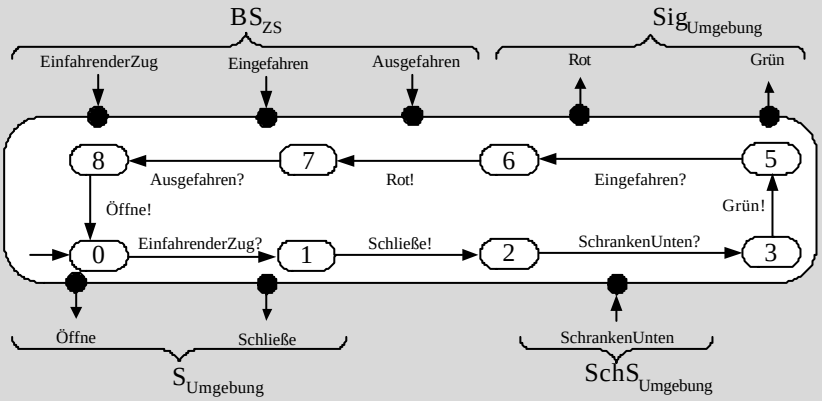
- Der Bahnübergang wird in einem Zyklus nur von einem Zug benutzt.
- Das Verhalten des Zuges entspricht dem folgenden Interfaceautomaten:

Interfacebeschreibung *Zug*



Auf Basis dieser Annahmen entspricht das Verhalten des Architekturelementes Bahnübergangssteuerung dem folgenden Interfaceautomaten:

Interfacebeschreibung
BahnübergangsSteuerung X Signale X ZugSensor X SchrankenSensor X Schranken



5.3.4 Modulspezifikationen

Für die Verhaltensspezifikation in der Modulimplementierungsphase sind die Interfacespezifikationen zu erweitern und zu verfeinern. In der Architekturbeschreibungssprache COOL erfolgt dies durch die Erweiterung der Meta-Klasse *Modul* um das anwendungsspezifische Attribut *Modulspezifikation*. Diese Modulspezifikation verfeinert die Interfacespezifikation und beschreibt das vollständige Verhalten eines flachen Architekturelementes. Dies lässt sich z. B. durch den Quellcode einer Klassenspezifikation in einer objektorientierten Programmiersprache realisieren.

5.4 Architekturevaluation

Eine Anforderung an die Architekturbeschreibungssprache COOL für den SOQA-Prozess ist die automatische Ermittlung der Qualitätseigenschaften auf Basis der Architekturspezifikation, sowie die Überprüfung der Konformität mit den Qualitätsanforderungen. Dazu wird im Folgenden beschrieben, wie die Architekturbeschreibungssprache um Evaluationsmodelle und Anforderungsreferenzen zu erweitern ist. Ausgehend von diesen Erweiterungen lassen sich kompositionsbasierte Evaluationsverfahren anwenden, welche die Qualitätseigenschaften der Architekturspezifikation ermitteln. Die Vorgehensweise dieser kompositionsbasierten Evaluationsverfahren wird zunächst allgemein beschrieben. Anschließend wird am Beispiel von Fehlerbaummodellen eine konkrete Realisierung gezeigt.

5.4.1 Evaluationsmodelle und Anforderungsreferenzen

Für die Integration der geforderten Anforderungsreferenzen und Evaluationsmodelle werden in der Architekturbeschreibungssprache die Meta-Klassen *Softwareelement* und *Hardwareelement* um die anwendungsspezifischen Attribute *Anforderungen* und *Evaluationsmodelle* erweitert. Das Attribut *Anforderungen* referenziert dabei mit einer Liste die eindeutigen Bezeichner der relevanten Anforderungen. Durch das Attribut *Evaluationsmodelle* wird eine Liste von Referenzen auf alle verfügbaren oder notwendigen Evaluationsmodelle spezifiziert. Ein Evaluationsmodell beschreibt dabei Annahmen über die Ausprägungen der Qualitätseigenschaften eines Architekturelementes. Spezifiziert werden diese Annahmen für die verschiedenen Qualitätseigenschaften mit geeigneten Evaluationsmodelltypen. Eine Auswahl der verwendeten Evaluationsmodelltypen sowie deren Zuordnung zu den Qualitätseigenschaften ist in Tabelle 5-2 dargestellt.

Tabelle 5-2 Qualitätseigenschaften und geeignete Evaluationsmodelltypen

Qualitätseigenschaft	Evaluationsmodelltyp
Sicherheit	Fehlerbaummodelle
Zuverlässigkeit, Wartbarkeit, Verfügbarkeit	Fehlerbaum- und Markov-Modelle, Zuverlässigkeitsblockdiagramme
Echtzeitfähigkeit	Scheduling- und Performanzmodelle
Ökonomische Eigenschaften	Allgemeine Ökonomische- und Lebenszykluskostenmodelle

5.4.2 Allgemeine Vorgehensweise bei der kompositionsbasierten Architekturevaluation

Für die Architekturevaluation werden in der Architekturbeschreibungssprache COOL kompositionsbasierte Evaluationstechniken verwendet. Diese ermitteln auf Basis der Strukturspezifikation und den annotierten Evaluationsmodellen die Qualitätseigenschaften einer Softwarearchitektur.
Eine solche kompositionsbasierte Architekturevaluation erfolgt in zwei Schritten. Im ersten Schritt werden für alle flachen Architekturelemente die Evaluationsmodelle erstellt. Dazu werden Annahmen über die Qualitätseigenschaften des Architekturelementes getroffen. Als Beispiele für diese Annahmen werden bei Fehlerbaummodellen die Auftrittswahrscheinlichkeiten von elementaren Fehlverhalten, bei Markov-Modellen die Übergangswahrscheinlichkeiten zwischen den einzelnen Zuständen, bei Scheduling-Modellen die WCET (*worst case execution times*) und bei ökonomischen Modellen die voraussichtlichen Entwicklungskosten bestimmt.

Im zweiten Schritt wird ausgehend von den Evaluationsmodellen der flachen Architekturelemente ein Evaluationsmodell für die gesamte Architektur erstellt. Dies erfolgt rekursiv absteigend in der Kompositionshierarchie. Dafür wird für jeden Evaluationsmodelltyp ein Konstruktionsverfahren benötigt. Dieses erzeugt für ein Evaluationsmodell ein hierarchisches Architekturelement ausgehend von den Evaluationsmodellen aller eingebetteten Architekturelemente.

5.4.3 Fehlerbaummodelle

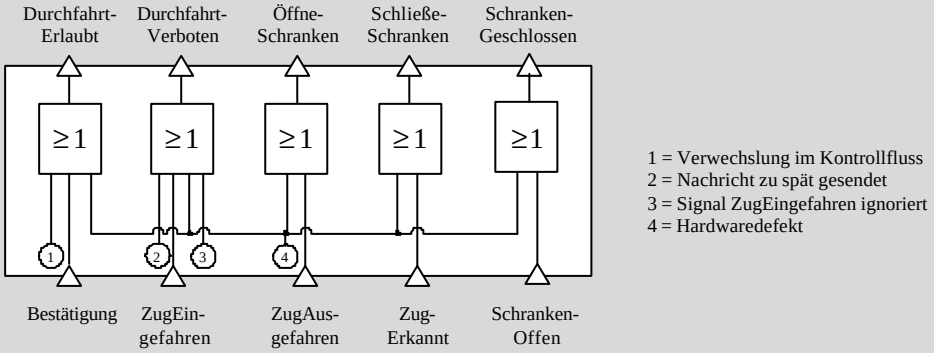
Zur Verdeutlichung der Konzepte von modularen Evaluationsmodellen und kompositionsbasierten Evaluationstechniken werden im Folgenden modulare Fehlerbaummodelle als Beispiel verwendet. Diese werden zunächst eingeführt. Anschließend wird gezeigt, wie die Fehlerbäume für flache Architekturelemente spezifiziert und für hierarchische Architekturelemente konstruiert werden.

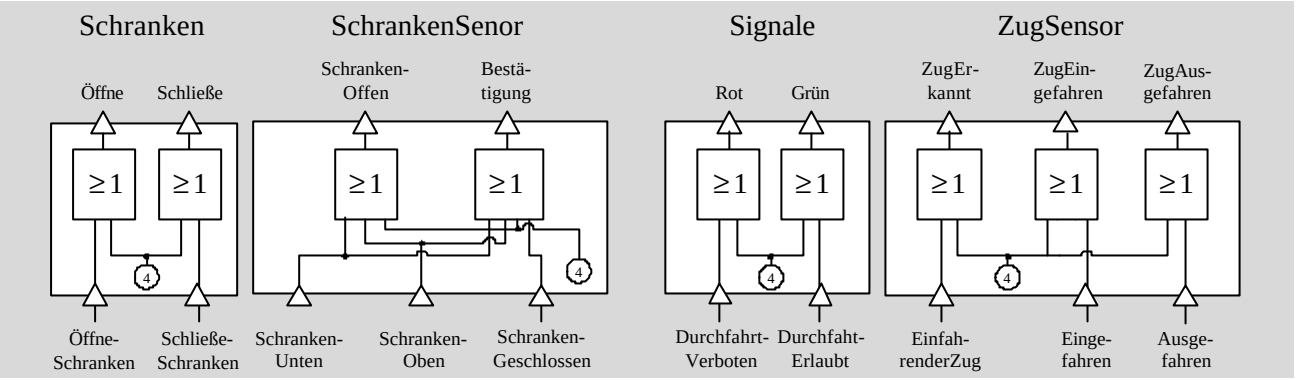
5.4.3.1 Einführung

Allgemeine Fehlerbäume /IEC 61025/ wurden bereits in Abschnitt 3.4.2 vorgestellt. Für die Integration in die Architekturbeschreibungssprache COOL und für die Verwendung in kompositionsbasierten Evaluationstechniken müssen diese jedoch erweitert werden.
Die erweiterten Fehlerbäume werden als modulare Fehlerbäume bezeichnet /Kaiser et al. 03/, /Grunske 03b/. Ihnen liegt ein Komponentenkonzept zugrunde, das es erlaubt, Fehlerbäume zu modularisieren und hierarchisch zu strukturieren. Die Grundlage des Komponentenkonzeptes sind Fehlerbaumkomponenten, welche die verursachten Fehlverhalten einer Komponente und deren Auftrittswahrscheinlichkeiten spezifizieren. Zur Spezifikation der Fehlverhalten werden interne Fehlerereignisse verwendet. Da die Fehlverhalten der Komponenten in einem System jedoch nicht unabhängig voneinander sind, ist es erforderlich, mit Fehlerbaumkomponenten zu beschreiben, wie sich die Fehlverhalten von Komponenten untereinander auswirken. Dazu besitzen Fehlerbaumkomponenten Schnittstellenelemente, welche wie bei einer Architekturspezifikation als Ports bezeichnet werden. Bei den Ports wird zwischen Eingabeports und Ausgabeports unterschieden. Ein Eingabeport spezifiziert dabei ein Fehlverhalten in der Umgebung einer Komponente, welches eine Auswirkung auf das korrekte Verhalten der Komponente hat. Ein Ausgabeport beschreibt ein Fehlverhalten der Komponente, welches Auswirkung auf das korrekte Verhalten der Komponenten in der Umgebung hat.
Die interne Beschreibung einer Fehlerbaumkomponente spezifiziert mit logischen Verknüpfungen, wie sich die internen Fehler und die Fehlverhalten der Eingabeports auf die Fehlverhalten der Ausgabeports auswirken. Die logischen Verknüpfungen werden dabei als Fehlerbaumgatter bezeichnet.

Beispiel 5-9

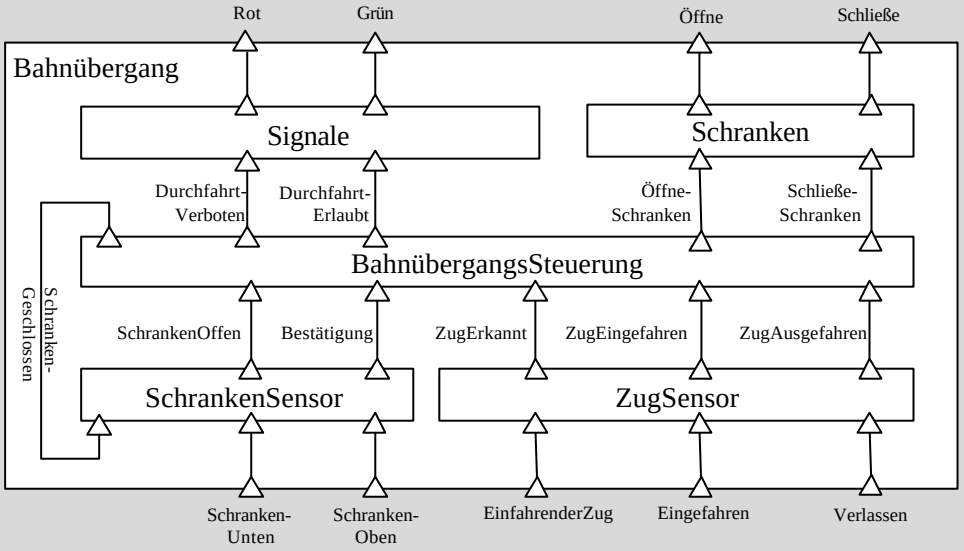
Als Beispiel werden die Fehlerbäume des bereits modellierten Bahnübergangs modelliert. Jeder Komponente ist ein Fehlerbaum zugeordnet, der die fehlerhaften Ausgabeereignisse auf fehlerhafte Eingabeereignisse und interne Fehler zurückführt. Als Beispiel dafür ist das Ausgabesignal *DurchfahrtErlaubt* der Komponente *BahnübergangsSteuerung* fehlerhaft, wenn ein interner Fehler im Kontrollfluss auftritt oder das Eingabesignal *Bestätigung*, das angibt, dass die Schranken geschlossen sind, falsch ist, oder ein Hardwarefehler auftritt.





Da Softwarearchitekturen hierarchisch verfeinert werden, kann eine Fehlerbaumkomponente auch feingranulare Fehlerbaumkomponenten enthalten. Von den enthaltenen Fehlerbaumkomponenten kann jedoch nur auf die Ports zugegriffen werden. Die Interna der Fehlerbaumkomponente werden somit gekapselt und bleiben verborgen.

Beispiel 5-10 Die Fehlerbäume der einzelnen Komponenten lassen sich analog zur Kompositionshierarchie des Bahnübergangs miteinander verbinden. Das Ergebnis ist der im Folgenden dargestellte Fehlerbaum des Bahnübergangs.



5.4.3.2 Definition

Zur Vereinheitlichung der Spezifikation werden die modularen Fehlerbäume als hierarchische typisierte Hypergraphen dargestellt. Im Gegensatz zur Strukturspezifikation werden jedoch gerichtete zyklenfreie Hypergraphen verwendet, um der Struktur und der Evaluierbarkeit eines Fehlerbaumes gerecht zu werden.

Definition 5.6 modularer Fehlerbaum

Ein modularer Fehlerbaum (Fehlerbaumkomponente) mFB ist ein gerichteter, azyklischer, hierarchischer und typisierter Hypergraph. Dieser Hypergraph wird aus der Menge aller bildbaren Hypergraphen G_{mFB} über die Typen $L_V = \langle P_{Eingabe}, P_{Ausgabe}, P_{Intern} \rangle$ und $L_E = \langle G_{Gatter}, G_{Vebindung}, G_{Proxy} \rangle$ durch das Tupel $\langle V, E, att, lab, ext, cts \rangle$ charakterisiert, wobei Folgendes gilt:

1 Ein Knoten $n \in V$ ist entweder ein Ausgabeport $lab(n) = P_{Ausgabe}$, ein Eingabeport $lab(n) = P_{Eingabe}$, oder ein internes Fehlerereignis $lab(n) = P_{Intern}$.
2 Eine Hyperkante $e \in E$ ist entweder ein logisches Gatter $lab(e) = G_{Gatter}$, eine Verbindung $lab(e) = G_{Verbindung}$ oder ein Proxy für einen enthaltenen modularen Fehlerbaum $lab(e) = G_{Proxy}$
3 Die Funktion att wird durch das Tupel $\langle source, target \rangle$ spezifiziert, wobei durch $source : E \rightarrow V^*$ die Sequenz der Quellknoten und durch $target : E \rightarrow V^*$ eine Sequenz der Zielknoten der Hyperkante spezifiziert wird.

4 Die Sequenz ext beschreibt den externen Knoten $ext \in V^*$ des modularen Fehlerbaumes. Als externe Knoten werden jedoch nur Knoten des Typs $P_{Eingabe}$ und $P_{Ausgabe}$ akzeptiert.

5 Die Funktion $cts : E \rightarrow G_{mFB}$ ist eine Zuweisungsfunktion, welche allen Hyperkanten des Typs G_{Proxy} einen eingebetteten modularen Fehlerbaum zuordnet.

6 Ein modularer Fehlerbaum ist syntaktisch korrekt, wenn die folgenden Verbindungsregeln gelten:

(i) Für alle Hyperkanten $e \in E$ des Typs logisches Gatter $lab(e) = G_{Gatter}$ oder Proxy $lab(e) = G_{Proxy}$ gilt: $\forall n \in attsource(e) : lab(n) = P_{Eingabe} \wedge n \notin ext$ und $\forall n \in atttarget(e) : lab(n) = P_{Ausgabe} \wedge n \notin ext$

(ii) Für alle Hyperkanten $e \in E$ des Typs Verbindung $lab(e) = G_{Verbindung}$ gilt: $\forall n \in att.source(e) : lab(n) = P_{Eingabe} \vee lab(n) = P_{Intern} \vee \left(lab(n) = P_{Ausgabe} \wedge n \notin ext \right)$ und $\forall n \in atttarget(e) : lab(n) = P_{Ausgabe} \vee \left(lab(n) = P_{Eingabe} \wedge n \notin ext \right)$

(iii) Alle Knoten $n \in V$ des Typs Ausgabeport des modularen Fehlerbaumes $lab(n) = P_{Ausgabe} \wedge n \in ext$ oder internen Eingabe ports $lab(n) = P_{Eingabe} \wedge n \notin ext$ sind ausschließlich mit einer Hyperkante verbunden.

Die Verbindungsregeln (i) und (ii) sichern die Struktur des Fehlerbaumes als gerichteten Graphen. Die Regel (iii) ist für die Analysierbarkeit des Fehlerbaumes notwendig. Ist ein Ausgabeport des modularen Fehlerbaumes oder ein Eingabe port eines eingebetteten modularen Fehlerbaumes mit mehreren Hyperkanten verbunden, so kann nicht mehr entschieden werden, welche Ausfallwahrscheinlichkeit für den Knoten gilt.

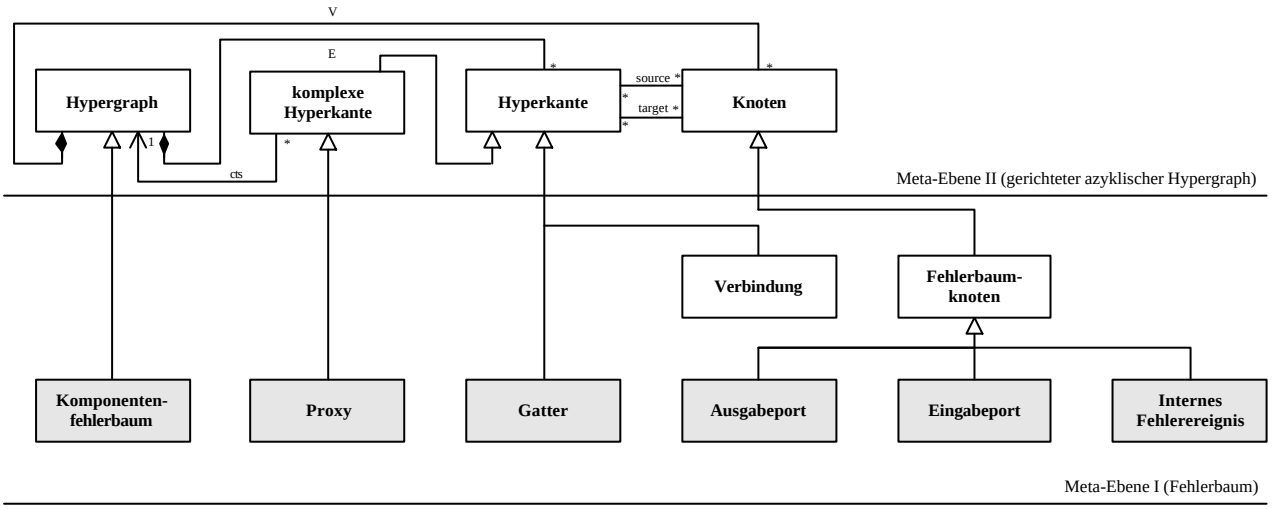


Abbildung 5-5 Meta-Modell eines modularen Fehlerbaumes

Für die vollständige Spezifikation eines modularen Fehlerbaumes besitzen die *Fehlerbaumknoten* eine anwendungsspezifisches Attribut, welches eine Ausfallwahrscheinlichkeit oder die Wahrscheinlichkeit eines Fehlverhaltens beschreibt. Die *Gatter* besitzen zusätzlich eine logische Funktion, welche zur Berechnung der Ausgabewahrscheinlichkeit bestimmt wird. Dabei werden die in der /DIN 25424/ beschriebenen logischen Verknüpfungen und Berechnungsvorschriften verwendet.

5.4.3.3 Erstellung von Fehlerbaumkomponenten für flache Architekturelemente

Bei der Erstellung eines Komponentenfehlerbaumes für ein flaches Architekturelement kann die IF-FMEA (Interfacefokussierte FMEA) /Papadopoulos et al. 01/ eingesetzt werden. Diese sucht in einem strukturierten Prozess nach möglichen Abweichungen vom korrekten Verhalten. Im ersten Schritt wird dazu aus der Interfacespezifikation die Menge aller ausgehenden Nachrichten ermittelt. Jede dieser ausgehenden Nachrichten kann die folgenden Typen von Fehlverhalten aufweisen:

- zs Eine Nachricht erfolgt außerhalb des spezifizierten Zeitfensters und ist zu spät
- zf Eine Nachricht erfolgt außerhalb des spezifizierten Zeitfensters und ist zu früh
- w Eine Nachricht ist außerhalb des spezifizierten Wertebereiches
- u Eine Nachricht wurde unterlassen
- ng Eine Nachricht wurde erzeugt, aber nicht gefordert

In den Komponentenfehlerbaum werden daher für jede ausgehende Nachricht fünf Ausgabeports eingefügt, welche jeweils als Nachrichtenname.zs, Nachrichtenname.zf, Nachrichtenname.w, Nachrichtenname.u und Nachrichtenname.ng bezeichnet werden.
Im zweiten Schritt werden aus der Interfacespezifikation die eingehenden Nachrichten bestimmt und entsprechend jeweils fünf Eingabeports pro Nachricht zur Fehlerbaumkomponente hinzugefügt. Die Eingabeports werden dabei nach dem gleichen Schema benannt wie die Ausgabeports. Im dritten Schritt werden mit Brainstormingtechniken oder auf Basis von Erfahrungswissen die internen Fehlverhalten der Komponente ermittelt. Diese werden entsprechend als interne Fehlerereignisse spezifiziert. Im vierten Schritt wird für jeden Ausgabeport bestimmt, von welchen Eingabeports und welchen internen Fehlerereignissen das Fehlverhalten abhängt. Die ermittelten Abhängigkeitsbeziehungen werden als Fehlerbaum notiert. Dabei wird der Ausgabeport als Wurzel und die verursachenden internen Fehlerereignisse und die Eingabeports als Blätter des Fehlerbaumes dargestellt. Bei der Identifizierung der Abhängigkeitsbeziehungen werden oft zusätzliche interne Fehlerereignisse erkannt. Diese müssen ebenfalls in die Fehlerbaumkomponente integriert werden. Im abschließenden Schritt wird die Fehlerbaumkomponente vereinfacht. Dazu werden alle nicht benötigten internen Fehlerereignisse und Eingabeports gelöscht. Des Weiteren werden alle Ausgabeports, welche nicht Wurzel eines Fehlerbaumes sind, gelöscht.

5.4.3.4 Konstruktion von Fehlerbaumkomponenten für hierarchische Architekturelemente

Da hierarchische Softwarekomponenten zumeist komplexe Interfacespezifikationen besitzen, ist eine automatische Erstellung der Fehlerbäume für die Evaluation vorteilhaft. Daher wird im Folgenden ein Konstruktionsformalismus vorgestellt, welcher einen Komponentenfehlerbaum für hierarchische Architekturelemente basierend auf der Kompositionshierarchie erstellt. Dabei werden von allen enthaltenen Architekturelementen die Komponentenfehlerbäume benötigt. Diese werden in den zu konstruierenden Fehlerbaum integriert. Für die Verschaltung der integrierten Fehlerbäume sind anschließend zwei Schritte erforderlich. Im ersten Schritt werden Eingabe und Ausgabeports zweier Komponentenfehlerbäume verbunden, wenn sich Fehlverhalten zwischen den Komponenten fortpflanzen. Eine solche Fehlerfortpflanzung erfolgt über die Kommunikationsbeziehung Verschaltung in der ADL COOL. Ausgehend von den Kommunikationsbeziehungen werden die ausgetauschten Ereignisse und die dazugehörigen Ausgabe- und Eingabeports in den Fehlerbäumen identifiziert. Besitzt dabei das sendende Architekturelement im Fehlerbaum einen Ausgabeport, welcher einen identischen Bezeichner besitzt wie ein Eingabeport im Fehlerbaum des empfangenden Architekturelementes, so werden diese beiden Ports verschaltet.
Im zweiten Schritt wird die Fehlerfortplanzung zwischen den eingebetteten Komponenten und der Komponentenumgebung betrachtet. Eine solche Fehlerfortpflanzung erfolgt über die Kommunikationsbeziehung Bindung in der ADL COOL. Daher werden für alle Bindungen die übertragenen Ereignisse ermittelt. Für jedes Fehlverhalten, welches durch einen Ausgabeport mit dem Namen des Ereignisses bei der eingebetteten Komponente charakterisiert ist, wird ein Ausgabeport bei dem konstruierten Fehlerbaum erzeugt. Beide Ausgabeports werden anschließend verbunden. Analog erfolgt dies für alle über Bindungen ausgetauschten fehlerhaften Eingabeereignisse.
Zusammenfassend kann die Konstruktion von Fehlerbaumkomponenten für hierarchische Architekturelemente mit dem folgenden Algorithmus erfolgen /Grunske 03b/:

Algorithmus erzeugeFehlerbaum(k)

Input: k Komponente, für die ein Fehlerbaum erzeugt werden soll
Output: Fehlerbaum der Komponente k

1. Wenn *k* keine Subkomponenten hat, gib den Fehlerbaum von *k* zurück.
2. Erzeuge einen leeren Fehlerbaum *f* für *k*.
3. Iteriere über Subkomponenten *sk* von *k*
    a. erzeuge rekursiv den Fehlerbaum von *sk*
    b. füge den erzeugten Fehlerbaum zu *f* hinzu
4. Iteriere über alle Verschaltungen *v* in *k*
    a. Ermittle die Fehlerbäume *fsk1* und *fsk2* der durch die Verschaltungen *v* verbundenen Subkomponenten im Fehlerbaum *f*
    b. Iteriere über alle Aktionen *a* der Verschaltung
        i. Wenn *fsk1* einen Ausgabeport und *fsk2* einen Eingabeport für ein Fehlverhalten von *a* aufweist, verbinde die beiden Ports entsprechend

ii. Wenn *fsk2* einen Ausgabeport und *fsk1* einen Eingabeport für ein Fehlverhalten von *a* aufweist, verbinde die beiden Ports entsprechend

5. Iteriere über alle Bindungen *b* in *k*
   a. Ermittle den Fehlerbaum *fsk* der zur Bindung *b* gehörenden Subkomponente im Fehlerbaum *f*
   b. Iteriere über alle Aktionen *a* die zu *b* gehören
      i. Wenn *fsk* für *a* einen Fehler aufweist, erzeuge einen Ein- bzw. Ausgang für den Fehler und verbinde diesen mit *fsk*
6. gebe *f* zurück

Eine ausführliche Spezifikation dieses Algorithmus ist im Anhang B enthalten.

5.5 Wohlgeformtheitsregeln

Ausgehend von den Metamodellen für die Strukturspezifikation, für die Verhaltensspezifikation und für die Evaluationsmodelle lassen sich mit der Architekturbeschreibungssprache COOL gültige und syntaktisch korrekte Architekturen spezifizieren, welche ein softwareintensives technisches System unzureichend oder gar falsch beschreiben. Daher werden im Folgenden Wohlgeformtheitsregeln für die Architekturbeschreibungssprache COOL vorgestellt, die die Wohlgeformtheit einer vom Anwender erstellten Architektur sichern.

Die notwendigen Wohlgeformtheitsregeln werden zur besseren Verständlichkeit in Regeln für die strukturelle Wohlgeformtheit, in Regeln für die Evaluierbarkeit und in Regeln für eine wohlgeformte Verhaltensbeschreibung unterteilt. Für die strukturelle Wohlgeformtheit wurden die folgenden Regeln identifiziert:

- Jede Architektur und jeder Architekturelementtyp wird durch einen zusammenhängenden Graphen beschrieben.
- Jede Architektur enthält mindestens einen Sensor, einen Aktor und ein Architekturelement.
- Jedes Softwareelement besitzt genau einen Ausführungsknoten und dieser ist über eine Ausführungsverschaltung mit dem Ausführungsknoten einer Hardwareplattform oder eines Hardwarekanals verbunden.
- Jedes interne Interfaceelement gehört zu einem Architekturelement und ist mit mindestens einer Verschaltung oder Bindung verbunden.
- Jede Verschaltung, Bindung, Ausführungsverschaltung in einer Architektur ist mit mindestens zwei Interfaceelementen verbunden.
  - Durch eine Bindung wird dabei jeweils ein internes und ein externes Interfaceelement verbunden.
  - Durch eine Verschaltung werden dabei zwei interne Interfaceelemente verbunden.
  - Durch eine Ausführungsverschaltung werden zwei Ausführungsknoten miteinander verbunden.
- Jeder Architekturelementtyp besitzt mindestens ein externes Interfaceelement.
  - Für die Spezifikation der externen Interfaceelemente von Komponententypen, Aktortypen, Sensortypen und Hardwareplattformtypen werden Typen verwendet, welche von der Metaklasse Port abgeleitet wurden.
  - Für die Spezifikation der externen Interfaceelemente von Konnektortypen und Hardwarekanaltypen werden ausschließlich Typen verwendet, welche von der Metaklasse Rolle abgeleitet wurden.

Die Wohlgeformtheitsregeln für die Evaluierbarkeit und für eine wohlgeformte Verhaltensbeschreibung lassen sich wie folgt spezifizieren:

- Jedes flache Architekturelement enthält genau eine wohlgeformte Interfacespezifikation
- Jedes flache Architekturelement enthält für jedes zu evaluierende Qualitätsmerkmal ein entsprechendes wohlgeformtes Evaluationsmodell.

Diese Wohlgeformtheitsregeln setzten sowohl die Wohlgeformtheit der Interfacespezifikation als auch die Wohlgeformtheit der Evaluationsmodelle voraus. Daher müssen für jede Interfacespezifikationsnotation und für jedes Evaluationsmodell separat die Wohlgeformtheitsregeln spezifiziert werden. Für die in diesem Kapitel beschriebenen Interfaceautomaten und Komponentenfehlerbäume werden diese Wohlgeformtheitsregeln in /de Alfaro et al. 01/ sowie in /DIN 25424/, /Grunske 03b/ und /Kaiser et al. 03/ spezifiziert.

5.6 Zusammenfassung

In diesem Kapitel wurde mit der Architekturbeschreibungssprache COOL eine neuartige Architekturbeschreibungssprache für softwareintensive technische Systeme vorgeschlagen. Diese Architekturbeschreibungssprache nutzt für die Strukturspezifikation hierarchische typisierte Hypergraphen. Dieser Spezifikationsformalismus ermöglicht es, Architekturtransformationsoperatoren als Hypergraphtransformationsregeln, wie im folgenden Kapitel beschrieben, zu spezifizieren und automatisch auf eine Architekturspezifikation anzuwenden. Dadurch erfüllt die Architekturbeschreibungssprache COOL die im Abschnitt 4.2.2 aufgestellte Anforderung A1 für die Realisierung des automatisierten SOQA-Prozesses.
Für die Erfüllung der Anforderung A2 des automatisierten SOQA-Prozesses wurde in der Architekturbeschreibungssprache COOL zudem ein Konzept zur Annotierung der Architekturelemente mit Interfacespezifikationen und modularen Evaluationsmodellen eingeführt. Als Interfacespezifikationen wurden dabei Interfaceautomaten verwendet. Diese Interfaceautomaten sollten das extern beobachtbare Verhalten jedes flachen Architekturelementes beschreiben. Ausgehend davon ist die Konstruktion der Interfacespezifikation für hierarchische Architekturelemente und somit auch für die gesamte Architekturspezifikation mit einem Kompositionsalgorithmus möglich. Der Kompositionsalgorithmus dazu wurde in diesem Kapitel vorgestellt.
Durch die Annotierung mit modularen Evaluationsmodellen ist die Bestimmung der Qualitätseigenschaften einer Architektur möglich. Dabei wurde ein Ansatz ähnlich den Interfacespezifikationen gewählt. Dieser Ansatz ermittelt zunächst die Evaluationsmodelle für alle flachen Architekturelemente und konstruiert anschließend die Evaluationsmodelle für die hierarchischen Architekturelemente mit kompositionsbasierten Techniken. Die technische Realisierbarkeit von modularen Evaluationsmodellen sowie die Konstruktion von Evaluationsmodellen für hierarchische Architekturelemente wurden am Beispiel von Fehlerbäumen gezeigt.
Sowohl für die Interfaceautomaten als auch für die Fehlerbäume wurde in diesem Kapitel eine Abbildung auf hierarchische typisierte Hypergraphen vorgeschlagen. Somit werden Hypergraphen als einheitlicher Spezifikationsformalismus für die gesamte Spezifikation der Architekturbeschreibungssprache COOL verwendet. Diese erleichtert vor allem die Erstellung von Werkzeugen für die Modellierung, Transformierung und Evaluierung von Architekturen /Grunske 04/.

6 Architekturtransformationsoperatoren

Die in diesem Kapitel vorgestellten Architekturtransformationsoperatoren bilden die Grundlage für die Restrukturierung einer Architektur im SOQA-Prozess. Diese Restrukturierung soll die Qualitätseigenschaften der Softwarearchitektur verbessern, ohne dabei die funktionalen Eigenschaften zu verändern. Des Weiteren sollen die Architekturtransformationen automatisch auswählbar und anwendbar sein.
Im vorherigen Kapitel wurde gezeigt, wie sich Architekturspezifikationen für softwareintensive technische Systeme mit hierarchischen typisierten Hypergraphen spezifizieren lassen. Ausgehend davon lassen sich Graphtransformationsregeln zur Spezifikation der Architekturtransformationsoperatoren verwenden. In diesem Kapitel werden zunächst die theoretischen Grundlagen der Graphtransformationsregeln vorgestellt. Anschließend wird beschrieben, wie diese Graphtransformationsregeln zur Spezifikation von Architekturtransformationsoperatoren genutzt werden und wie sich diese Architekturtransformationsoperatoren automatisch auf eine Softwarearchitekturspezifikation in der Architekturbeschreibungssprache COOL anwenden lassen.

6.1 Einführung in die Graphtransformation

Eine Graphtransformation ersetzt einen Teilgraphen in einem Anwendungsgraphen durch einen neuen Graphen. Der ersetzte Teilgraph wird dabei im Folgenden mit L und der einzufügende Teilgraph mit R bezeichnet. Die Spezifikation der Ersetzung erfolgt durch eine Graphtransformationsregel, welche einer Produktionsregel in einer String-Grammatik entspricht /Andries et al. 99/. Allgemein werden kontextsensitive und kontextfreie Graphtransformationsregeln unterschieden. Dabei besteht der ersetzte Teilgraph bei einer kontextfreien Regel ausschließlich aus einem nicht-terminalen Symbol, also einem Knoten oder einer Hyperkante mit einem entsprechenden Typ. Bei kontextsensitiven Regeln kann der ersetzte Teilgraph ein Hypergraph sein.
Neben der Art des zu ersetzenden Graphen werden Graphtransformationsregeln auch durch den Einbettungsmechanismus unterschieden. Dieser legt fest, wie der einzufügende Teilgraph mit dem Rest des Anwendungsgraphen verbunden wird, d.h. welche Kanten neu zu bilden sind. Dabei werden zwei Verfahren unterschieden, welche als *connecting*- und *gluing*-Verfahren bezeichnet werden. Beim *connecting*-Verfahren werden explizit Regeln angegeben, welche Knoten des einzufügenden Teilgraphen mit welchen Knoten des Anwendungsgraphen verbunden werden. Beim *gluing*-Verfahren hingegen werden Knoten im ersetzten und einzufügenden Teilgraphen identifiziert, welche bei der Ersetzung erhalten bleiben. Diese Knoten stellen die Klebestellen dar, über die der einzufügende Graph mit dem Anwendungsgraphen verbunden wird.

6.1.1 Graphtransformation in allgemeinen Hypergraphen

Im Folgenden werden zunächst die notwendigen Grundlagen für die Spezifikation von Graphtransformationen spezifiziert. Anschließend werden darauf aufbauend mit der Knoten-, Hyperkanten- und Hypergraphersetzung die grundlegenden Graphtransformationsregeln für Hypergraphen vorgestellt und eine Einführung in Graphgrammatiken und Graphtransformationssysteme gegeben.

6.1.1.1 Grundlagen

Grundlage für die Spezifikation und Anwendung von Hypergraphtransformationen ist die Identifizierung eines Teilgraphen in einem Anwendungsgraphen. Ein solcher Teilgraph wird wie folgt definiert.

Definition 6.1 Teilgraphen
Seien $G = \langle V, E, att, lab \rangle$ und $G' = \langle V', E', att', lab' \rangle$ zwei Hypergraphen, dann ist G' der Teilgraph von G, geschrieben $G' \subseteq G$, wenn $V' \subseteq V$, $E' \subseteq E$ und $att(e) = att'(e)$, $lab(e) = lab'(e)$ $lab(v) = lab'(v)$ für alle $e \in E'$ und $v \in V'$.

Nach dieser Definition muss ein Teilgraph exakt im Anwendungsgraphen identifiziert werden. Diese exakte Identifizierung würde die Anwendung einer Graphtransformationsregel jedoch zu stark einschränken. Daher erfolgt die Identifikation über struktur- und typerhaltende Abbildungen /Habel 92/. Eine solche struktur- und typerhaltende Abbildung ist ein Hypergraphmorphismus, welcher wie folgt definiert wird.

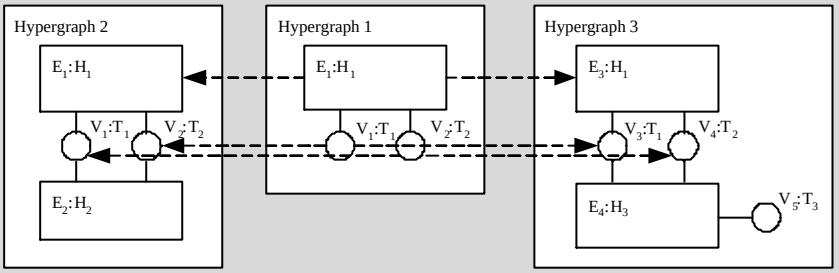
Definition 6.2 Hypergraphmorphismen
Seien $G = \langle V, E, att, lab \rangle$ und $G' = \langle V', E', att', lab' \rangle$ zwei Hypergraphen, dann wird ein Morphismus $m: G \to G'$ durch ein Tupel $\langle m_V, m_E \rangle$ charakterisiert. Dabei sind $m_V : V \to V'$ und $m_E : E \to E'$ zwei struktur- und typerhaltende Abbildungen, für die gilt:

$$\forall e \in E : lab'(m_E(e)) = lab(e)$$
$$\forall e \in E : att'(m_E(e)) = m_V^*(att(e))$$
$$\forall v \in V : lab'(m_V(v)) = lab(v)$$

Sind beide Abbildungen $m_V : V \to V'$ und $m_E : E \to E'$ injektiv (surjektiv, bijektiv), so ist auch die Abbildung $m: G \to G'$ injektiv (surjektiv, bijektiv). Ist $m: G \to G'$ bijektiv, so wird der Morphismus als Isomorphismus bezeichnet und die beiden Graphen G und G' sind isomorph. $G \cong G'$.

Beispiel 6-1

Zur Veranschaulichung der Unterschiede zwischen Teilgraphen und Hypergraphmorphismen wird das folgende Beispiel verwendet:



In diesem Beispiel ist der Hypergraph 1 ein Teilhypergraph des Hypergraphen 2. Es existiert ebenfalls ein HypergraphMorphismus von Hypergraph 1 auf Hypergraph 2, welcher durch die Abbildungspfeile dargestellt wird.

Im Gegensatz dazu ist Hypergraph 1 kein Teilhypergraph des Hypergraphen 3, da die Knoten- und Hyperkantenbezeichnungen unterschiedlich sind. Ein Hypergraphmorphismus von Hypergraph 1 auf Hypergraph 3 ist jedoch beschreibar. Dieser erhält die Typen und die Struktur und stellt keinerlei Bedingungen an die Bezeichnungen von Knoten und Hyperkanten.

Neben der Identifikation von Teilgraphen in einem Anwendungsgraphen muss für die Hypergraphtransformation definiert werden, wie ein Teilgraph aus einem Anwendungsgraphen entfernt wird und wie er zu einem Anwendungsgraphen hinzugefügt wird. Daher wird im Folgenden mit der disjunkten Vereinigung die Hypergraphaddition und mit dem Entfernen von Teilhypergraphen die Hypergraphsubtraktion definiert.

Definition 6.3 Disjunkte Vereinigung von Hypergraphen, Hypergraphaddition
Seien $G = \langle V, E, att, lab \rangle$ und $G' = \langle V', E', att', lab' \rangle$ zwei Hypergraphen mit $A \cup A' = \varnothing$, dann wird die disjunkte Vereinigung $G + G'$ durch den Hypergraphen $\langle V \cup V', E \cup E', att_{G+G'}, lab_{G+G'} \rangle$ beschrieben, wobei $att_{G+G'}$ und $lab_{G+G'}$ wie folgt gebildet werden:

$$att_{G+G'}(e) = \begin{cases} att_{G+G'}(e) = att(e) & für e \in E \\ att_{G+G'}(e) = att'(e) & sonst \end{cases}$$

$$lab_{G+G'}(a) = \begin{cases} lab_{G+G'}(a) = lab(a) & für a \in E \cup V \\ lab_{G+G'}(a) = lab'(a) & sonst \end{cases}$$

Definition 6.4 Entfernen von Teilhypergraphen, Hypergraphsubtraktion
Seien $G = \langle V, E, att, lab \rangle$ und $G' = \langle V', E', att', lab' \rangle$ zwei Hypergraphen, wobei $G' \subseteq G$, dann wird der Hypergraph $G - G'$ durch das Tupel $\langle V - V', E - E', att_{G-G'}, lab_{G-G'} \rangle$ konstruiert, wobei:
$att_{G-G'}(e) = att(e) - (V'), \forall e \in (E - E')$
$lab_{G-G'}(a) = lab(a), \forall a \in (V - V') \cup (E - E')$

6.1.1.2 Knotenersetzung

Bei der Knotenersetzung werden Nichtterminal-Knoten eines Anwendungsgraphen durch einen neuen Graphen ersetzt, welcher wiederum Terminal- und/oder Nichtterminal-Knoten enthält. Zur Verbindung des einzufügenden Graphen mit dem Anwendungsgraphen werden verschiedene *connecting*-Mechanismen verwendet. Im Folgenden wird mit dem NLC-Verfahren (*node label controlled*) exemplarisch ein einfaches Knotenersetzungs-Verfahren beschrieben. Dieses bildet die Grundlage für komplexere Knotenersetzungs-Verfahren, welche unter anderem in /Engelfriet, Rozenberg 97/ dargestellt werden.
Eine Graphtransformationsregel des NLC-Verfahrens wird wie folgt definiert:

Definition 6.5 Graphtransformationsregel (Knotenersetzung)
Bei der Knotenersetzung (NLC) wird eine Graphtransformationsregel durch das Tupel $\langle K, G_R, VR \rangle$ beschrieben. Dabei ist $K \in L_V$ ein Knotentyp, $G_R \in G$ der einzufügende Graph und $VR \in L_V \times L_V$ ist eine Verbindungsrelation.

Durch ein Element der Verbindungsrelation $(m, d) \in VR$ werden zwei Knotentypen spezifiziert. Dabei müssen nach der Anwendung der Graphtransformationsregel alle Knoten des einzufügenden Graphen mit dem Label d mit einem Knoten des Anwendungsgraphen mit dem Label m durch eine neue Kante verbunden werden. Diese Verbindung erfolgt jedoch nur, sobald der Knoten mit dem Label m vor der Anwendung der Graphtransformationsregel mit dem ersetzten Knoten verbunden war.
Für die Anwendung von Graphtransformationsregeln beim NLC-Verfahren ergibt sich ein allgemeiner Algorithmus. Dieser besitzt die folgenden Schritte:

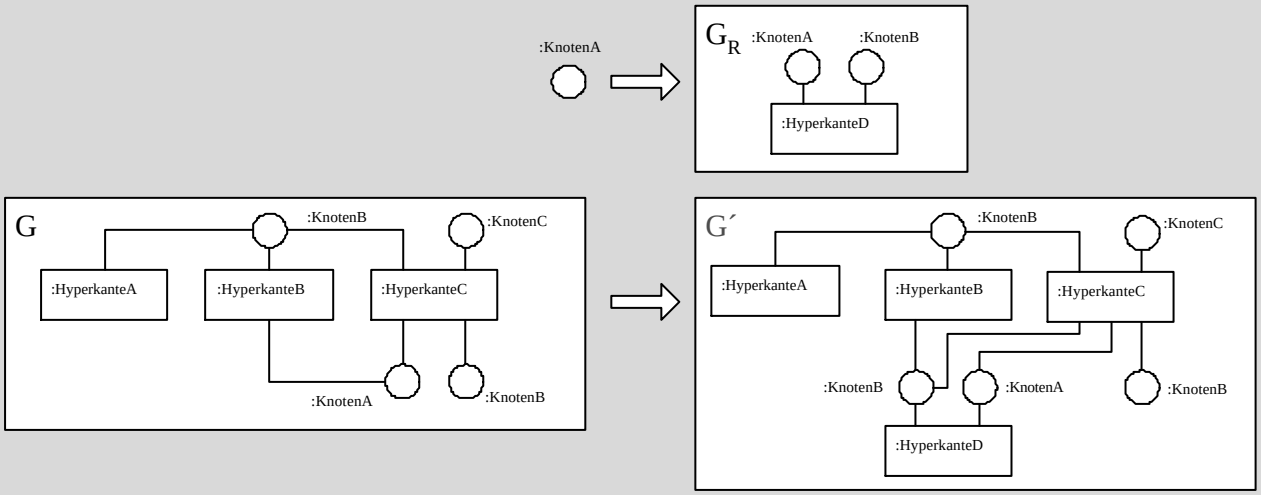
1. Finde einen Knoten mit dem Knotentyp K im Anwendungsgraphen.
2. Entferne den Knoten aus dem Anwendungsgraphen und alle Kanten, welche mit einem Knoten verbunden sind.
3. Füge den Graphen G_R in den Anwendungsgraphen ein und verbinde die Knoten des Graphen G_R mit den Knoten des Anwendungsgraphen gemäß der Verbindungsrelation VR.

Beispiel 6-2

Gegeben sei die folgende Knotenersetzungsregel r_{KE}:

$\langle KnotenA, G_r, \{(KnotenB, KnotenB), (KnotenC, KnotenA)\} \rangle$

Der einzufügende Graph G_R ist in der unteren Abbildung spezifiziert. Eine Anwendung der Graphtransformationsregel auf den dargestellten Anwendungsgraph G würde wie folgt aussehen:



6.1.1.3 Hyperkantenersetzung

Bei der Hyperkantenersetzung werden Nichtterminal-Hyperkanten durch einen neuen Graphen ersetzt. Diese Ersetzung ist kontextfrei und die Verbindung des einzufügenden Graphen mit dem Anwendungsgraphen erfolgt über das *gluing*-Verfahren. Definiert wird eine Graphtransformationsregel bei der Hyperkantenersetzung nach /Habel 92/ wie folgt:

Definition 6.6 Graphtransformationsregel (Hyperkantenersetzung)
Bei der Hyperkantenersetzung wird eine Graphtransformationsregel durch das Tupel $\langle HK, G_R \rangle$ beschrieben. Dabei ist $HK \in E_G$ ein Hyperkantentyp des Anwendungsgraphen und $G_R \in G$ ein modularer einzufügender Hypergraph.

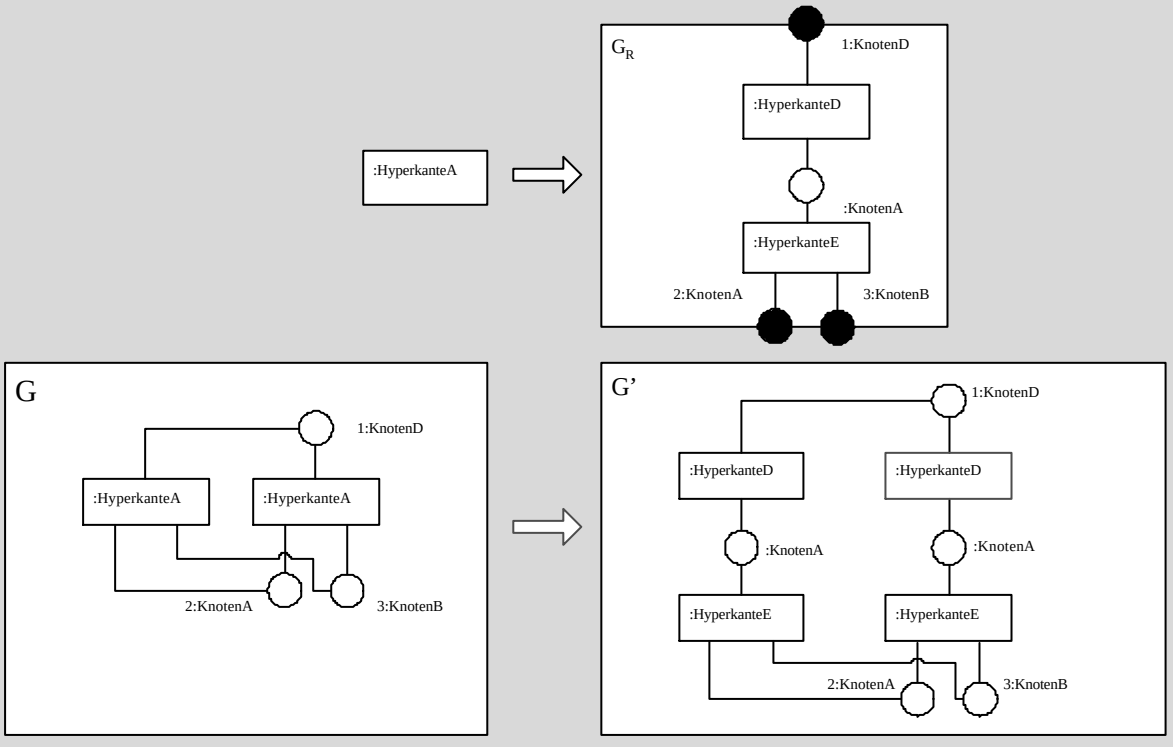
Für die Anwendung von Graphtransformationsregeln bei der Hyperkantenersetzung ergibt sich ein ähnlicher Algorithmus wie bei der Knotenersetzung. Dieser beinhaltet die folgenden Schritte:

1. Finde eine Hyperkante mit dem Hyperkantentyp *HK* im Anwendungsgraphen und prüfe, ob die Hyperkante und der modulare einzufügende Hypergraph gleicher Arität sind.
2. Entferne die Hyperkanten aus dem Anwendungsgraphen.
3. Füge den modularen einzufügenden Hypergraphen G_R in den Anwendungsgraphen ein. Entferne dazu die externen Knoten des modularen einzufügenden Hypergraphen G_R und verbinde alle mit diesen externen Knoten verbundenen Hyperkanten mit den entsprechenden Knoten der entfernten Hyperkante.

Im Schritt 3 wird der einzufügende modulare Hypergraph mit dem Anwendungsgraphen über das *gluing*-Verfahren verbunden. Als Klebepunkte dienen dabei die externen Knoten des einzufügenden modularen Hypergraphen und die assoziierten Knoten der entfernten Hyperkante. Zur Bestimmung, welche Knoten einen Klebepunkt bilden, wird eine Abbildungsfunktion verwendet. In /Habel 92/ werden dazu die assoziierten Knoten der Hyperkante und die externen Knoten des einzufügenden modularen Hypergraphen durchnummeriert. Auf Basis dieser Nummerierung lassen sich die Klebepunkte eindeutig identifizieren, indem beispielsweise Knoten mit identischen Nummer einen Klebepunkt bilden.

Beispiel 6-3

Zur Veranschaulichung der Hyperkantenersetzung wird im Folgenden die Regel $\langle HyperkanteA, G_R \rangle$ und deren Anwendung auf einen Anwendungsgraphen dargestellt.



6.1.1.4 Hypergraphersetzung

Bei der Hypergraphersetzung wird ein Teilgraph in einem Anwendungsgraphen identifiziert und durch einen neuen Graphen ersetzt. Die Spezifikation der Hypergraphersetzung erfolgt durch kontextsensitive Graphtransformationsregeln, welche wie folgt definiert werden:

Definition 6.7 Graphtransformationsregel (Hypergraphersetzung)
Eine Graphtransformationsregel wird durch das Tupel $\langle G_L, G_I, G_R, l, r \rangle$ charakterisiert. Dabei ist $G_L \in G$ der ersetzte Graph, $G_R \in G$ der einzufügende Graph und $G_I \in G$ ein Interfacegraph. Durch $l: G_I \rightarrow G_L$ und $r: G_I \rightarrow G_R$ werden zwei Hypergraphmorphismen beschrieben, welche die Beziehungen zwischen den drei Graphen beschreiben. Verkürzt wird eine Graphtransformationsregel zur Hypergraphersetzung wie folgt spezifiziert: $G_L \xleftarrow{l} G_I \xrightarrow{r} G_R$.

Für die Anwendung von Graphtransformationsregeln bei der Hypergraphersetzung ergibt sich ein Algorithmus, welcher die folgenden Schritte beinhaltet:

1. Finde mit einem Hypergraphmorphismus $o: G_L \rightarrow G$ ein struktur- und typähnliches Auftreten des ersetzten Graphen im Anwendungsgraphen.

1.1 Prüfe, dass keine Hyperkante $e \in E_G - E_{o(G_L)}$ mit einem Knoten $k \in V_{o(G_L)} - V_{o(G_I)}$ verbunden ist.

1.2 Prüfe, dass zwei verschiedene Hyperkanten $x, y \in E_{G_L}$ oder Knoten $x, y \in V_{G_L}$, für die gilt $o(x) = o(y)$, Hyperkanten oder Knoten des Interfacegraphen $x, y \in E_{G_I} \cup V_{G_I}$ sind.

2. Entferne den Hypergraphen $o(G_L - l(G_I))$ und alle Hyperkanten, welche mit einem Knoten $k \in V_{o(G_L)} - l(V_{o(G_I)})$ assoziiert sind, aus dem Anwendungsgraphen. Der resultierende Graph wird als Kontextgraph D und der Morphismus zwischen dem Interfacegraphen und diesem Kontextgraphen wird als $o': G_I \rightarrow D$ bezeichnet.

3. Füge den Hypergraphen $G_R - G_I$ in den Anwendungsgraphen ein.

3.1 $G' = D + (G_R - G_I)$

3.2 Verbinde alle Hyperkanten $e \in E_{G_R} - E_{G_I}$, welche mit einem Knoten $k \in V_{G_I}$ assoziiert sind, mit dem entsprechenden Knoten des Kontextgraphen $o'(k)$.

In den beiden Schritten 1.1 und 1.2 werden mit der *dangling condition* und der *identification condition* die notwendigen Bedingungen für die Graphersetzung geprüft /Habel 92/, /Ehrig 79/. Ein Verletzen der *dangling condition* würde zu einem Kontextgraphen führen, welcher Hyperkanten enthält, die mit einem gelöschten Knoten assoziiert sind. Veranschaulicht wird dies in Abbildung 6-1 (a). Wird die *identification condition* verletzt, kann es, wie in Abbildung 6-1 (b) dargestellt, passieren, dass der Knoten G1 aus dem Anwendungsgraphen entfernt werden muss, da er Teil des Graphen $o(G_L - l(G_I))$ ist und gleichzeitig erhalten bleiben muss, da er ein Klebepunkt ist. In diesem Fall kann der Kontextgraph nicht eindeutig konstruiert werden.

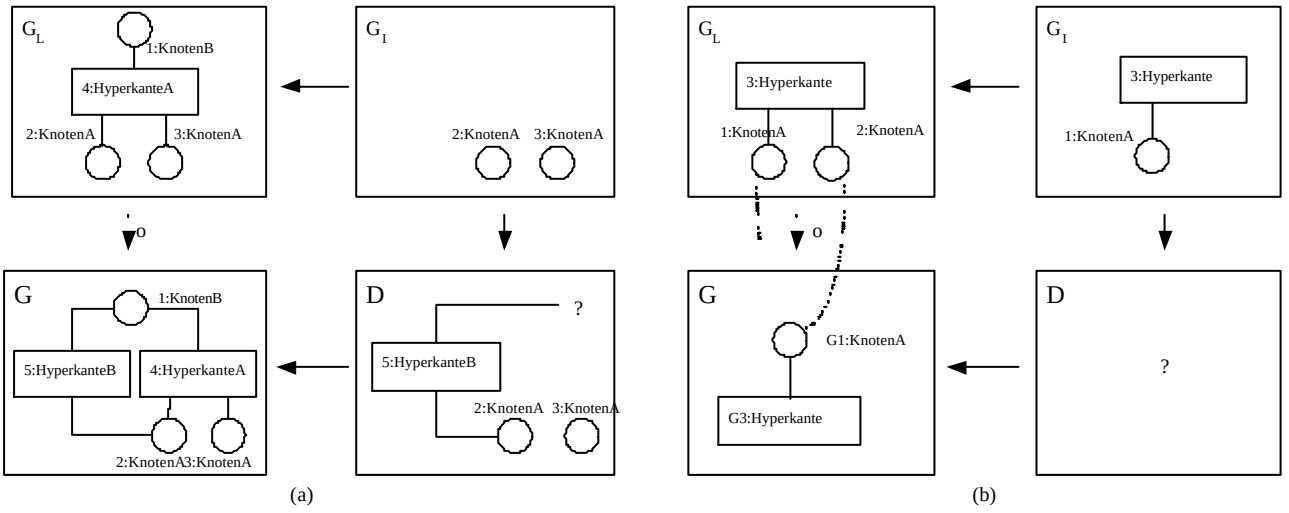
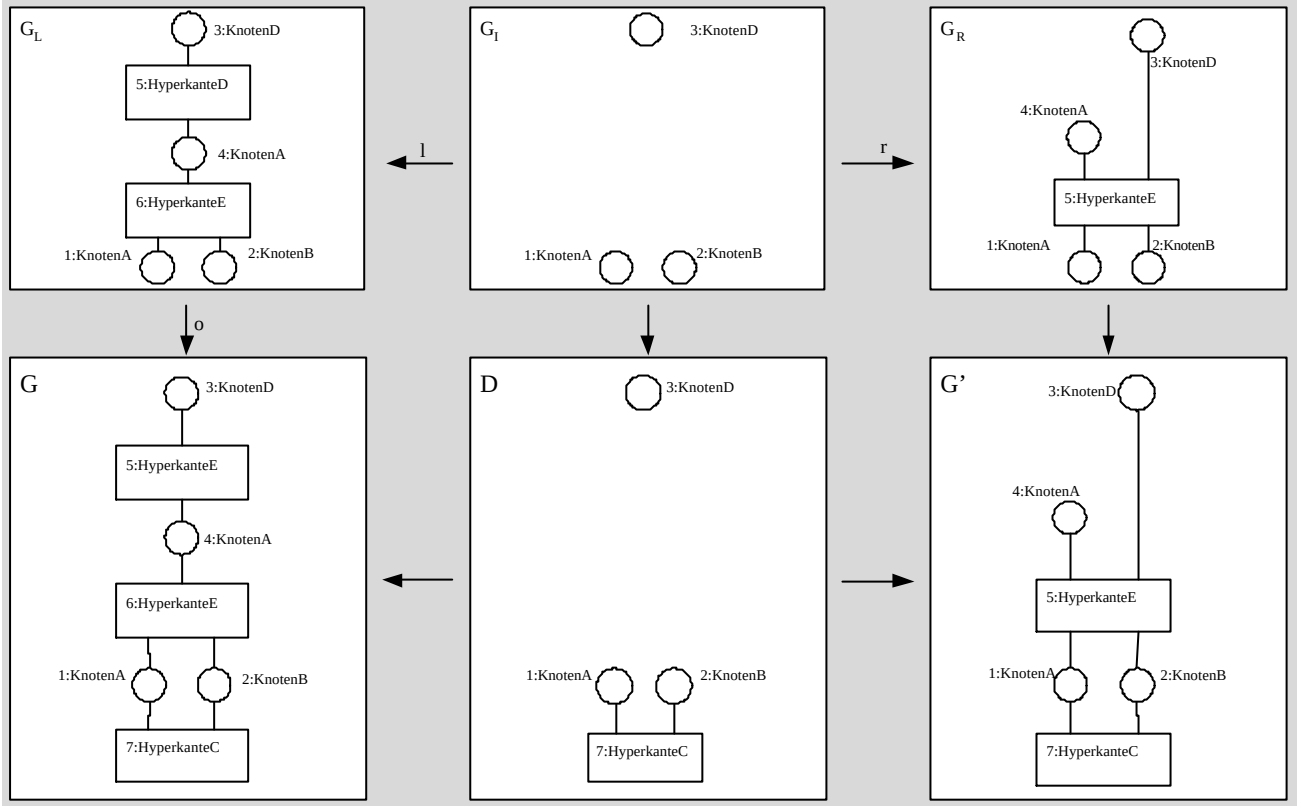


Abbildung 6-1 Beschreibung der *dangling condition* (a) und der *identification condition* (b)

Beispiel 6-4

Zur Veranschaulichung wird in diesem Beispiel eine Graphtransformationsregel $G_L \xleftarrow{l} G_I \xrightarrow{r} G_R$ auf den Anwendungsgraphen G angewendet. Die Morphismen wurden dabei entsprechend der Knoten- und Kantenbezeichner gewählt.



Für die Implementierung von Hypergraphersetzungsregeln wird der algebraische Ansatz von /Ehrig 79/ bevorzugt, welcher auch als *double pushout* Verfahren bezeichnet wird. Dieser Ansatz basiert auf der *pushout*- und *pushout complement*- Konstruktion in der Kategorie der typisierten Hypergraphen und den in Abbildung 6-2 dargestellten *double pushout* Diagramm. Zur Bestimmung des Kontextgraphen D ist das *pushout complement* von $\langle o,l \rangle$ zu bilden. Dabei ergibt sich auch der Morphismus $o' : G_I \rightarrow D$, welcher anschließend zur Konstruktion des resultierenden Graphen G' mit dem *pushout* $\langle o',r \rangle$ verwendet wird. Zusammenfassend lässt sich der dargestellte Algorithmus für die Anwendung einer Hypergraphersetzungsregel vereinfachen, indem die Schritte 2 und 3 inklusive der Unterschritte durch die folgenden beiden Schritte ausgetauscht werden:

2. Konstruiere das *pushout complement* von $\langle o,l \rangle$.
3. Konstruiere den *pushout* von $\langle o',l \rangle$.

Für einen vertiefenden Einblick in die Graphttransformation mit dem *double pushout* Verfahren wird /Corradini et al. 97/ empfohlen.

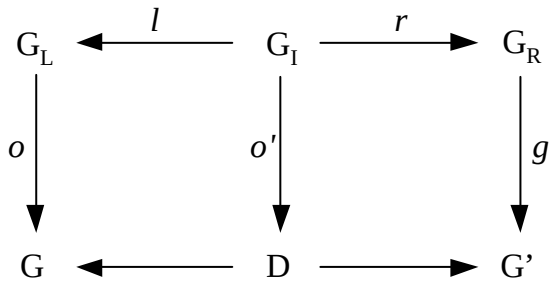


Abbildung 6-2 Darstellung einer Hypergraphersetzungsregel mit einem *double pushout* Diagramm

Neben dem *double pushout* Verfahren existiert auch das *single pushout* Verfahren /Löwe 93/, welches die Anwendung einer Graphtransformation auf die Konstruktion eines *pushouts* reduziert. Dieser *pushout* ist in Abbildung 6-3 dargestellt. Es ist erkennbar, dass beim *single pushout* Verfahren der Interfacegraph nicht mehr explizit dargestellt wird. Er ergibt sich implizit aus den beiden Graphen G_L, G_R und dem Morphismus lr

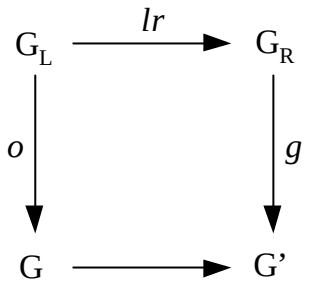


Abbildung 6-3 Darstellung einer Hypergraphersetzungsregel mit einem *single pushout* Diagramm

Nach /Löwe 93/ besitzen das *single pushout* und *double pushout* Verfahren eine ähnliche Mächtigkeit. Vorteile ergeben sich aber bei der Anwendung einer Regel im *single pushout* Verfahren, da Hyperkanten automatisch gelöscht werden, wenn sie mit einem gelöschten Knoten im Anwendungsgraphen assoziiert waren. Dadurch muss die *dangling condition* bei der Anwendung einer *single pushout* Regel nicht geprüft werden.
In dieser Arbeit werden jedoch *double pushout* Graphtransformationsregeln verwendet, da sie verständlicher sind. Diese Verständlichkeit folgt aus der einfacheren Erkennbarkeit der Klebepunkte durch die explizite Modellierung des Interfacegraphen.

6.1.1.5 Graphgrammatiken und Graphtransformationssysteme

Ausgehend von der Definition der verschiedenen Formen von Graphtransformationsregeln lassen sich die Graphgrammatiken und Graphtransformationssysteme beschreiben. Als Grundlage werden aber noch die Begriffe der direkten Ableitung und der Ableitungssequenz benötigt, welche wie folgt definiert werden:

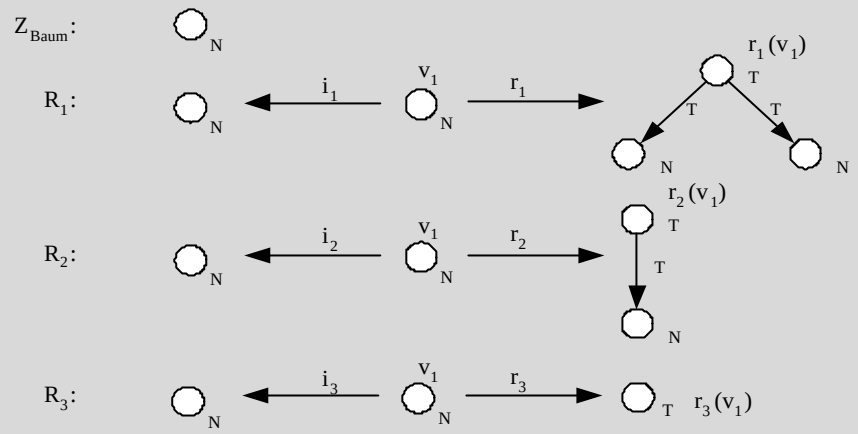
Definition 6.8 Direkte Ableitung und Ableitungssequenzen
Der resultierende Graph G' bei der Anwendung einer Graphtransformationsregel r auf einen Anwendungsgraphen G wird als direkte Ableitung bezeichnet. Notiert wird dies wie folgt: $G \Rightarrow^r G'$. Eine Sequenz von direkten Ableitungen $G \Rightarrow^r ... \Rightarrow^r G'$ wird als Ableitungssequenz der Länge k bezeichnet. Verkürzt kann diese als $G \Rightarrow_k^r G'$ bei einer Ableitungssequenz mit definierter Länge und $G \Rightarrow_*^r G'$ bei einer Ableitungssequenz mit undefinierter Länge notiert werden.

Ausgehend von dieser Definition wird eine Graphgrammatik nach /Habel 92/ wie folgt definiert:

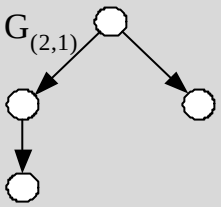
Definition 6.9 Graphgrammatik
Eine Graphgrammatik ist ein System $HG = \langle N, T, R, Z \rangle$, wobei $N \in L_E \cup L_V$ eine Menge von nichtterminalen Knoten- und Kantentypen, $T \in L_E \cup L_V$ eine Menge von terminalen Knoten- und Kantentypen, R eine Menge von Graphtransformationsregeln und Z ein initialer Graph ist. Die durch die Graphgrammatik erzeugbare Sprache $L(HG)$ wird durch eine Menge von Graphen beschrieben, welche ausschließlich terminale Knoten- und Kantentypen enthalten und durch eine Ableitungssequenz mit den Regeln der Graphgrammatik aus dem initialen Graph erzeugbar sind:
$L(HG) = \{ Z \Rightarrow_*^R H \}$

Beispiel 6-5

Als Beispiel wird eine Graphgrammatik $HG_{Baum} = \langle N_{Baum}, T_{Baum}, R_{Baum}, Z_{Baum} \rangle$ zur Konstruktion von binären Bäumen verwendet. Dabei enthält die Menge der nichtterminalen Knoten- und Kantentypen N_{Baum} ausschließlich den Knotentyp N und die Menge der terminalen Knoten- und Kantentypen T_{Baum} enthält den Kantentyp T und den Knotentyp T. Der Knotentyp T charakterisiert dabei einen Blattknoten im Baum. Der Startgraph Z_{Baum} und die Regeln $R_{Baum} = \{R_1, R_2, R_3\}$ sind wie folgt spezifiziert:



Mit dieser Graphgrammatik lassen sich alle binären Bäume erzeugen. Als Beispiel dafür wird durch die exemplarische Ableitungssequenz $Z \Rightarrow^{R_1} ... \Rightarrow^{R_2} ... \Rightarrow^{R_3} ... \Rightarrow^{R_3} G_{(2,1)}$ der folgende Baum erzeugt:



6.1.2 Graphtransformation mit Anwendungsbedingungen

Für die strukturierte Anwendung werden in Graphtransformationssystemen oft bedingte Graphtransformationsregeln verwendet /Habel et al. 96/, /Mens 99/. Diese erweitern die Graphtransformationsregeln durch Anwendungsbedingungen, deren Erfüllung vor der Anwendung der Graphtransformationsregel geprüft werden muss. Die vorgestellten Anwendungsalgorithmen für die Hyperkanten- und die Hypergraphersetzungen enthalten bereits Schritte zur Prüfung dieser Anwendungsbedingungen. Diese sind bei der Hyperkantenersetzung die Überprüfung der Aritätskonformität der zu ersetzenden Hyperkante und des einzufügenden Hypergraphen. Bei der Hypergraphersetzung sind die *dangling condition* und die *identification condition* Anwendungsbedingungen, welche für die fehlerfreie Anwendung der Graphtransformationsregel erforderlich sind /Habel 92/.
Allgemein schränken Anwendungsbedingungen den Kontext ein, in welchem eine Graphtransformationsregel anwendbar ist. Beispiele für solche Kontexteinschränkungen sind die Existenz oder Abstinenz bestimmter Knoten, Hyperkanten oder Teilgraphen im Anwendungsgraphen. Dabei wird ersteres (Existenz) als positive Kontextbedingung und zweiteres (Abstinenz) als negative Kontextbedingung bezeichnet. Definiert werden positive und negative Kontextbedingung sowie deren Erfüllbarkeit nach /Habel et al. 96/ wie folgt:

Definition 6.10 Anwendungsbedingungen (Vorbedingungen)
Sei $G_L \xleftarrow{l} G_I \xrightarrow{r} G_R$ eine Graphtransformationsregel und $o: G_L \to G$ ein identifizierender Morphismus, dann werden die Anwendungsbedingungen $AB(L)$ einer Graphtransformationsregel durch zwei Mengen $AB^+(L)$ und $AB^-(L)$ von totalen Graphmorphismen $c_n: G_L \to G_L'$ zwischen dem zu ersetzenden Graphen G_L und einem Graphen G_L' beschrieben. Die beiden Mengen werden als positive und negative Anwendungsbedingungen bezeichnet.
Eine positive Kontextbedingung $c_n \in AB^+(L)$ wird erfüllt, wenn es einen Morphismus $s: G_L' \to G$ gibt, für den $c_n \circ s = o$ gilt.
Eine negative Kontextbedingung $c_n \in AB^-(L)$ wird erfüllt, wenn es keinen Morphismus $s: G_L' \to G$ gibt, für den $c_n \circ s = o$ gilt.

Nach dieser Definition werden Anwendungsbedingungen durch einen Graphen G_L' und einen Morphismus $c: G_L \to G_L'$ spezifiziert. Die Erfüllbarkeit einer Anwendungsbedingung wird durch das Diagramm in Abbildung 6-4 veranschaulicht. Dabei ist es nach der Definition bei einer negativen Anwendungsbedingung

erforderlich, dass kein Morphismus $s : G_L' \rightarrow G$ existiert. Bei einer positiven Anwendungsbedingung muss mindestens ein solcher Morphismus existieren.

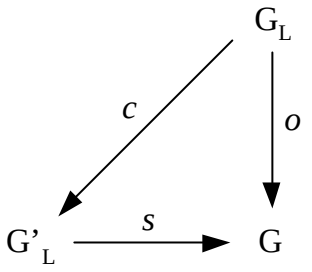
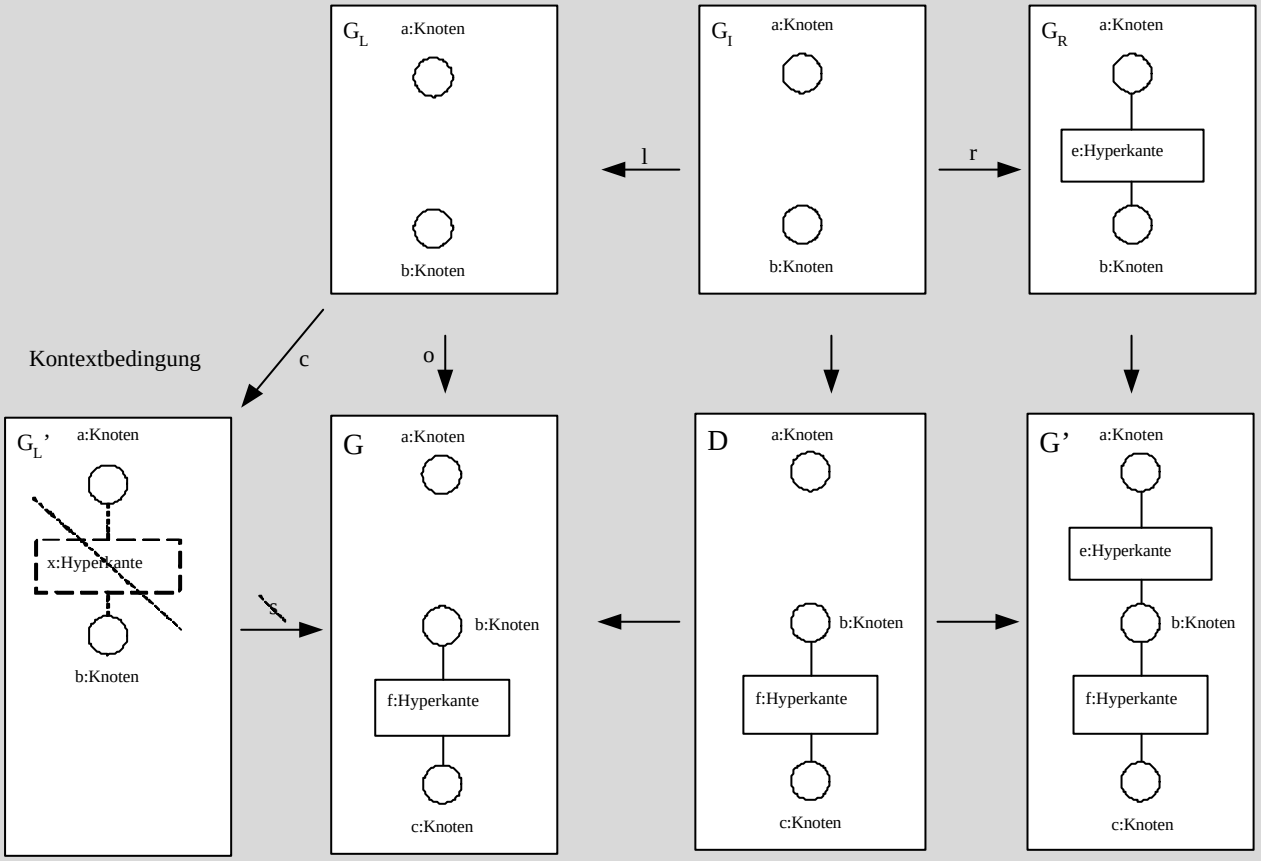


Abbildung 6-4 Erfüllbarkeit einer Kontextbedingung

Beispiel 6-6

Zur Veranschaulichung der Verwendung von Kontextbedingungen wird eine Graphtransformationsregel verwendet, welche zwei Knoten durch eine Hyperkante verbindet. Diese Graphtransformationsregel darf jedoch nur angewendet werden, wenn bisher keine Hyperkante zwischen den beiden Knoten existiert.



Bei der Anwendung der Graphtransformationsregel werden durch den identifizierenden Morphismus $o : G_I \rightarrow G$ die Knoten *a* und *b* im zu ersetzenden Graphen auf die Knoten *a* und *b* im Anwendungsgraphen abgebildet. Dadurch ist es nicht möglich, einen Morphismus $s : G_L' \rightarrow G$ zu finden.

Würden durch den identifizierenden Morphismus $o : G_I \rightarrow G$ die Knoten *a* und *b* im zu ersetzenden Graphen auf die Knoten *b* und *c* im Anwendungsgraphen abgebildet, dann würde ein Morphismus $s : G_L' \rightarrow G$ existieren. Dieser Morphismus bildet die Hyperkante *x* auf die Hyperkante *f* ab. Die Abbildung der Knoten erfolgt entsprechend dem identifizierenden Morphismus.

Bisher wurden nur Anwendungsbedingungen betrachtet, welche sich auf den ersetzten Graphen G_L beziehen. Diese werden auch als Vorbedingungen bezeichnet /Mens 99/. Neben den Vorbedingungen lassen sich für eine Graphtransformationsregel auch Nachbedingungen spezifizieren. Diese beziehen sich auf den einzufügenden Graphen und werden dementsprechend als $AB(R)$ bezeichnet.
Ausgehend von der Spezifikation der Vor- und Nachbedingungen wird eine konditionale Graphtransformationsregel wie folgt definiert.

Definition 6.11 Konditionale Graphtransformationsregeln
Eine konditionale Graphtransformationsregel wird spezifiziert durch ein Tupel $\langle r, AB \rangle$, dabei ist $r: G_L \xleftarrow{l} G_I \xrightarrow{r} G_R$ eine Graphtransformationsregel und $AB = AB(L) \cup AB(R)$ eine Menge von Vor- und Nachbedingungen.

Zur Verallgemeinerung der Anwendungsalgorithmen sollten alle Anwendungsbedingungen zusammengefasst und in einem separaten Schritt geprüft werden. Daher sollten alle Anwendungsalgorithmen um den folgenden Schritt erweitert werden:

1.1 Prüfung der Anwendungsbedingungen AB.

Dieser Schritt sollte die bisherige separate Prüfung der Anwendungsbedingungen ersetzen.

6.1.3 Graphtransformation mit typ-generischen Graphtransformationsregeln

Die Aussagekraft von Graphtransformationsregeln wird durch die strikte Typisierung eingeschränkt, da nur Knoten und Hyperkanten mit den in Graphtransformationsregeln spezifizierten Typen im Anwendungsgraphen identifiziert werden dürfen /Corradini et al. 97/. Als Erweiterung werden im Folgenden Graphtransformationsregeln vorgeschlagen, bei denen nicht nur die beschriebenen Typen identifizierbar sind, sondern auch alle Subtypen in der Vererbungshierarchie. Dazu sind zunächst die Hypergraphmorphismen um subtyperhaltene Abbildungen zu erweitern:

Definition 6.12 Subtypeerhaltende Hypergraphmorphismen
Seien $G = \langle V, E, att, lab \rangle$ und $G' = \langle V', E', att', lab' \rangle$ zwei Hypergraphen, dann wird ein subtyperhaltender Morphismus $m: G \to G'$ durch ein Tupel $\langle m_V, m_E \rangle$ charakterisiert. Dabei sind $m_V: V \to V'$ und $m_E: E \to E'$ zwei struktur- und subtyperhaltende Abbildungen für die gilt:

- $\forall e \in E: lab'(m_E(e)) \hat{\circ}_E lab(e)$
- $\forall v \in V: lab'(m_V(v)) \hat{\circ}_V lab(v)$
- $\forall e \in E: att'(m_E(e)) = m_V^*(att(e))$

Durch die Einführung der subtyperhaltenden Hypergraphmorphismen sind Hypergraphtransformationsregeln auch geeignet, Hypergraphen in einem Anwendungsgraphen zu identifizieren, welche Knoten und Hyperkanten eines Subtypen enthalten. Diese Identifizierung ist jedoch bei großen Hypergraphen und/oder großen Typhierarchien mit einem großen Aufwand verbunden. Daher wird im Folgenden mit der Einführung des Variablenkonzeptes ein leichtgewichtiger Formalismus als Alternative beschrieben. Im Gegensatz zu den subtyperhaltenden Hypergraphmorphismen werden nur ausgewählte Knoten und Hyperkanten in den Graphtransformationsregeln spezifiziert, welche durch Subtypen ersetzbar sind. Diese ausgewählten Knoten und Hyperkanten werden als Variablen bezeichnet. Die Ersetzung einer Variable durch einen Subtypen erfolgt durch eine Variablenumbenennung, welche wie folgt definiert wird.

Definition 6.13 Variablenumbenennung in Hypergraphen
Sei $G = \langle V, E, att, lab \rangle$ ein Hypergraph und $x \in \text{ran}(lab)$ ein in G verwendeter Variablentyp, dann kann dieser Variablentyp durch einen Subtypen y ($y \hat{\circ}_V x$ oder $y \hat{\circ}_E x$) ersetzt werden, indem für alle $a \in E \cup V$ mit $lab(a) = x$ der Variablentyp durch den Subtypen $lab(a) = y$ ersetzt wird. Geschrieben wird diese Umbenennung $G[x/y]$.

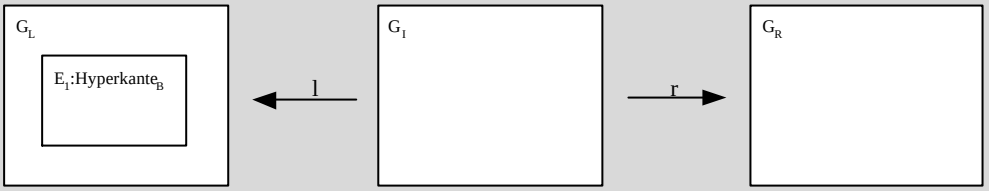
Bei der Identifizierung eines zu ersetzenden Graphen, welcher Variablen enthält, ist es erforderlich, alle Variablen in der Graphtransformationsregel so umzubenennen, dass diese Typen denen des Anwendungsgraphen entsprechen /Plum, Habel 96/. Dies wird in Anlehnung an logische Sprachen als Unifikation bezeichnet und wie folgt definiert:

Definition 6.14 Unifikation
Seien $G_L = \langle V_L, E_L, att_L, lab_L \rangle$ ein zu ersetzender Graph einer Graphtransformationsregel und $G = \langle V, E, att, lab \rangle$ ein Anwendungsgraph, dann ist eine Unifikation s eine Menge von Variablenumbenennungen, so dass gilt:
$s\, G_L \cong G$

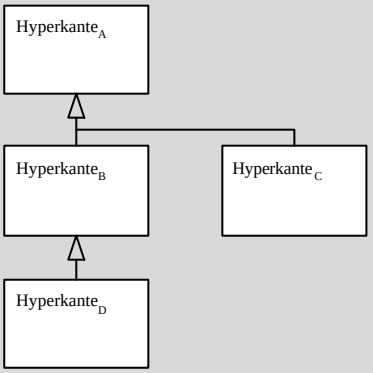
Für die Anwendung einer Graphtransformationsregel, die Variablen enthält, müssen die bei der Unifikation identifizierten Variablenumbenennungen s auf die drei Graphen G_L, G_I und G_R der Graphtransformationsregel angewendet werden. Diese Umbenennungen instanziieren eine konkrete Graphtransformationsregeln $s\, G_L \xleftarrow{l} s\, G_I \xrightarrow{r} s\, G_R$, welche nach dem herkömmlichen Verfahren auf die Anwendungsgraphen anwendbar ist.

Beispiel 6-7

Zur Veranschaulichung von typ-generischen Graphtransformationsregeln soll die folgende Regel dienen, welche eine variable Hyperkante des Typs Hyperkante_B löscht:



Geht man von der folgenden Typhierarchie aus, so kann bei der Anwendung der Regel entweder eine Hyperkante des Typs Hyperkante_B oder eine Hyperkante des Typs Hyperkante_D gelöscht werden.



6.1.4 Graphtransformation mit struktur-generischen Graphtransformationsregeln

Im vorhergehenden Abschnitt wurde die Mächtigkeit von Graphtransformationssystemen durch typ-generische Graphtransformationsregeln erweitert. Dabei konnte eine konkrete Graphtransformationsregel aus einer typ-generischen Graphtransformationsregel durch Umbenennung von Variablen mit Subtypen erzeugt werden. Eine solche typ-generische Graphtransformationsregel beschreibt also eine Menge von typähnlichen Graphtransformationsregeln. Für die Beschreibung einer Menge von strukturähnlichen Graphtransformationsregeln wird im Folgenden das Konzept der struktur-generischen Graphtransformationsregeln eingeführt. Diese struktur-generischen Graphtransformationsregeln sind besonders für die Erleichterung der Spezifikation der Architekturtransformationsoperatoren notwendig.
Bevor jedoch die struktur-generischen Graphtransformationsregeln definiert werden, wird zunächst bestimmt, unter welchen Bedingungen zwei Graphen struktur-ähnlich sind.

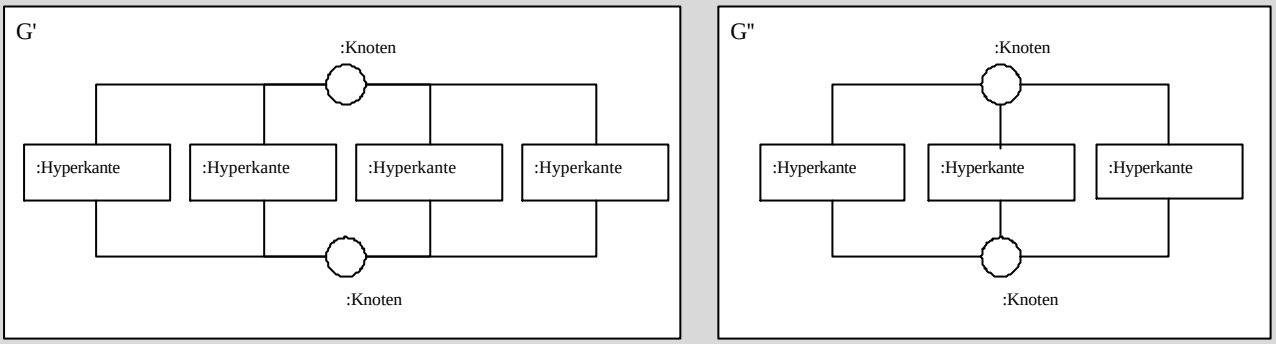
Definition 6.15 Strukturähnlichkeit von Hypergraphen
Zwei Hypergraphen $G' = \langle V', E', att', lab' \rangle$ und $G'' = \langle V'', E'', att'', lab'' \rangle$ sind strukturähnlich, wenn es einen Hypergraphen $G = \langle V, E, att, lab \rangle$ und eine Graphtransformationsregel $sr : G_L \xleftarrow{l} G_I \xrightarrow{r} G_R$ gibt, so dass beide Hypergraphen mit einer endlichen Ableitungssequenz aus dem Graphen G mit der Regel sr erzeugbar sind:

- $G \Rightarrow_*^{sr} G'$
- $G \Rightarrow_*^{sr} G''$

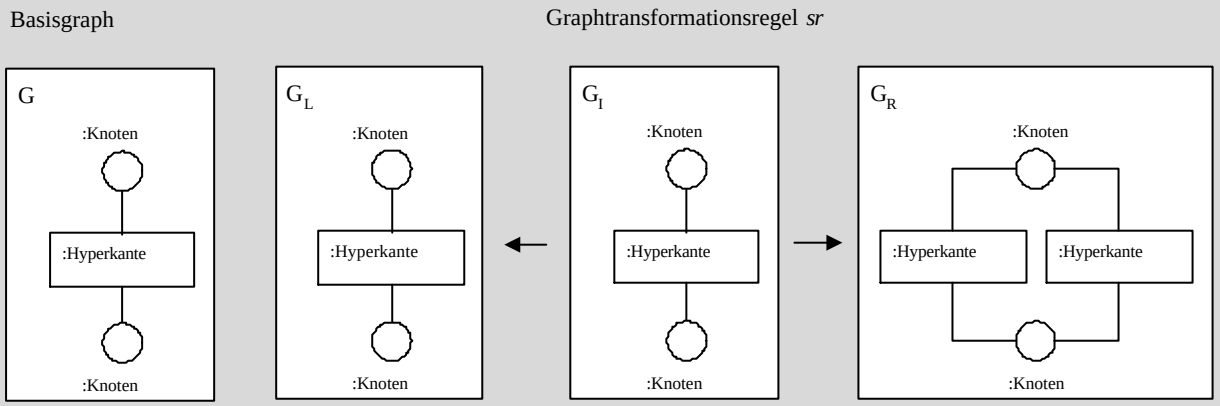
Der Hypergraph G wird dabei als Basisgraph und die Graphtransformationsregel sr als Strukturtransformationsregel bezeichnet.

Beispiel 6-8

Zur Verdeutlichung der Strukturähnlichkeit werden die folgenden beiden Graphen G' und G'' verwendet; beide besitzen eine ähnliche Struktur. Der Unterschied zwischen beiden Graphen besteht ausschließlich in der Anzahl der Hyperkanten, welche die beiden Knoten verbinden.



Die beiden Graphen lassen sich aus dem Basisgraphen G und der Regel sr mit einer endlichen Ableitungssequenz erzeugen.



Aufbauend auf der Definition der Strukturähnlichkeit lassen sich struktur-generische Graphtransformationsregeln wie folgt definieren:

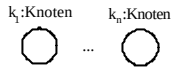
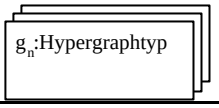
Definition 6.16 Struktur-generische Graphtransformationsregeln
Eine struktur-generische Graphtransformationsregel wird durch ein Tupel $\langle gtr, sr \rangle$ spezifiziert, wobei:

- $gtr : G_L^{gtr} \xleftarrow{l} G_I^{gtr} \xrightarrow{r} G_R^{gtr}$ eine Graphtransformationsregel ist, welche als Basistransformationsregel bezeichnet wird
- $sr : G_L^{sr} \xleftarrow{l} G_I^{sr} \xrightarrow{r} G_R^{sr}$ eine Graphtransformationsregel ist, welche als Strukturtransformationsregel bezeichnet wird

Eine konkrete Regel $kr : G_L^{kr} \xleftarrow{l} G_I^{kr} \xrightarrow{r} G_R^{kr}$ kann durch k-fache Ableitung aller Graphen der Basistransformationsregel mit der Strukturtransformationsregel sr erzeugt werden ($G_L^{gtr} \Rightarrow_k^{sr} G_L^{kr}$, $G_I^{gtr} \Rightarrow_k^{sr} G_I^{kr}$ und $G_R^{gtr} \Rightarrow_k^{sr} G_R^{kr}$).

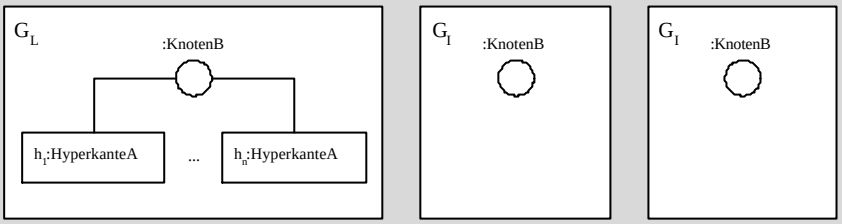
Zur Vereinfachung der Anwendung der struktur-generischen Graphtransformationsregeln wird in dieser Arbeit die Menge der Strukturtransformationsregeln eingeschränkt. Durch eine Einschränkung werden nur Strukturtransformationsregeln verwendet, welche entweder einen Knoten, eine Hyperkante oder eine hierarchische Hyperkante vervielfachen. Dadurch lassen sich die struktur-generischen Graphtransformationsregeln durch eine einfache Erweiterung der Notation graphisch darstellen. Diese Notationserweiterung wird in Tabelle 6-1 präsentiert.

Tabelle 6-1 Notation für strukturgenerische Graphtransformationen

Struktur transformations regel	Erweiterung der Notation
Knotenvervielfachung	
Hyperkantenvervielfachung	
Hypergraphvervielfachung	

Beispiel 6-9

Ein Beispiel für eine nutzbringende Anwendung von struktur-generischen Graphtransformationen und der Notationserweiterungen ist die im Folgenden dargestellte Regel, welche alle Hyperkanten des Typen HyperkanteA löscht, welche mit einem Knoten des Typen KnotenB verbunden sind:



6.1.5 Graphtransformation in hierarchischen typisierten Hypergraphen

In Kapitel 5 wurde die Architekturbeschreibungssprache COOL auf einen hierarchischen typisierten Hypergraphen abgebildet. Daher ist es ebenfalls sinnvoll, Graphtransformationsregeln für hierarchische Hypergraphen zu spezifizieren. Dazu müssen zunächst Morphismen in dieser Graphkategorie definiert werden. Dies erfolgt nach /Drewes et al. 02/ und /Hoffmann, Minas 01/ induktiv über Einbettungshierarchie.

Definition 6.17 Morphismen in hierarchischen Hypergraphen
Seien $G = \langle V, E, att, lab, cts \rangle$ und $G' = \langle V', E', att', lab', cts' \rangle$ zwei hierarchische typisierte Hypergraphen, dann wird ein Morphismus $m: G \to G'$ durch ein Tupel $\langle m_V, m_E, M \rangle$ charakterisiert. Dabei beschreibt $\langle m_V, m_E \rangle$ einen Morphismus eines flachen Graphen und M eine Familie von hierarchischen Morphismen M_e für alle in den komplexen Hyperkanten enthaltenen Hypergraphen. Dabei gilt für jeden Morphismus M_e mit $e \in \mathrm{dom}(cts)$:
$M_e : cts(e) \to cts'(m_E(e))$

Des Weiteren wird in /Drewes et al. 02/ nachgewiesen, dass in der Kategorie der hierarchischen Hypergraphen *pushout* und *pushout complement* existieren und gezeigt, wie sie sich konstruieren lassen. Somit lässt sich die Anwendung einer Hypergraphersetzungsregel für hierarchische Hypergraphen auf den algebraischen Ansatz für allgemeine Hypergraphen zurückführen. Ausgehend davon lässt sich die Hypergraphtransformation nach dem *double pushout* Verfahren auch auf hierarchische Hypergraphen abbilden /Drewes et al. 02/.

Beispiel 6-10

Ein Beispiel für die Anwendung von Graphtransformationen in hierarchischen typisierten Hypergraphen ist der in /Drewes et al. 02/ vorgestellte Flattening-Operator.

Dieser Transformationsoperator durchsucht einen Anwendungsgraphen nach komplexen Hyperkanten. Wird eine solche gefunden, werden im Anwendungsgraphen die komplexe Hyperkante und die externen Knoten gelöscht. Stattdessen wird der Inhalt der komplexen Hyperkante in den Anwendungsgraphen eingefügt. Die Knoten des eingefügten Graphen, die vorher mit den externen Knoten verbunden waren, werden dann mit den Knoten verbunden, mit denen die externen Knoten vorher verbunden waren.

Eine mögliche Anwendung dieses Flattening-Operators ist die Übersetzung der bereits dargestellten Komponentenfehlerbäume (hierarchische typisierte Hypergraphen) in nicht hierarchische Fehlerbäume. Diese ist beispielsweise bei der Berechnung erforderlich.

6.2 Spezifikation von Architekturtransformationsoperatoren

Durch die in Kapitel 5 beschriebene Abbildung von Strukturspezifikationen auf hierarchische typisierte Hypergraphen lassen sich die Konzepte der Graphtransformation auch zur Architekturtransformation anwenden. Dabei werden Teilgraphen in der Architektur identifiziert und durch neue Teilgraphen ersetzt. Dies wird im Folgenden als graphbasierte Architekturtransformation bezeichnet. Für die Spezifikation der Architekturtransformationen werden in diesem Abschnitt Architekturtransformationsoperatoren eingeführt und gezeigt, wie diese spezifiziert werden.
Dazu wird zunächst die allgemeine Vorgehensweise bei der Spezifikation von Architekturtransformationsoperatoren vorgestellt. Zur Verdeutlichung der konkreten Vorgehensweise wird anschließend die Spezifikation eines Architekturtransformationsoperators am Beispiel des Architekturmusters „Homogene Redundanz mit Mehrheitsentscheid“ gezeigt.

6.2.1 Allgemeine Vorgehensweise

Zur automatischen Anwendung des SOQA-Prozesses sind Transformationsoperatoren erforderlich, welche (teil)-automatisch und werkzeugunterstützt angewendet werden. Daher sind bei der Spezifikation eines Transformationsoperators alle Aspekte anzugeben, welche ein Werkzeug zu deren Anwendung benötigt. Diese Aspekte sind:

- die Transformationsregel,
- die Anwendungsbedingungen,
- die Verhaltensspezifikation aller hinzuzufügenden Architekturelementtypen und
- die Evaluationsmodelle aller hinzuzufügenden Architekturelementtypen.

Bei der Spezifikation eines Transformationsoperators sollte mit der Transformationsregel und den Anwendungsbedingungen begonnen werden. Ausgehend von der Transformationsregel ist bestimmbar, ob durch den Transformationsoperator neue Architekturelementtypen zur Architektur hinzugefügt werden. Ist dies der Fall, so werden für jeden hinzugefügten Architekturelementtyp eine Verhaltensspezifikation und die entsprechenden Evaluationsmodelle definiert.
Für die Spezifikation der Transformationsregel sind der zu entfernende Architekturgraph und der hinzuzufügende Architekturgraph zu identifizieren. Dabei ist darauf zu achten, dass der jeweilige Architekturgraph mit struktur- oder typgenerischen Konzepten flexibel gestaltet wird. Im zweiten Schritt werden die invarianten Elemente der Architekturtransformationsregel und somit der Interfacegraph ermittelt. Ausgehend davon lassen sich die Graphen G_L, G_I und G_R der Graphtransformationsregel bestimmen.
Die Anwendungsbedingungen ergeben sich aus den Zielen des Transformationsoperators, aus möglichen Einschränkungen bei der Operatoranwendung und aus den negativen Wechselwirkungen zu anderen Qualitätsmerkmalen. Ein Ziel eines Operators ist die Verbesserung einer speziellen Qualitätseigenschaft, beispielsweise die Verbesserung der Zuverlässigkeit. Spezifiziert werden diese Anwendungsbedingungen als negative Vorbedingungen, welche sich auf die Ausprägung von COOL-spezifischen Attributen beziehen. Auf Grund dieser negativen Vorbedingungen lassen sich die Transformationsoperatoren nur anwenden, wenn eine Verbesserung der Qualitätseigenschaften erforderlich ist. Für die Spezifikation von Einschränkungen bei der Operatoranwendung werden Anwendungsbedingungen in Form von positiven und negativen Kontextbedingungen verwendet. Durch eine negative Kontextbedingung sollte beispielsweise eine redundante Anwendung eines Transformationsoperators verhindert werden. Dazu muss die Kontextbedingung eine Operatoranwendung verhindern, bei der Architekturelemente oder Architekturstrukturen auf eine redundante Anwendung hindeuten. Positive Kontextbedingungen bei Transformationsoperatoren sind denkbar, wenn explizit bestimmte Architekturelemente oder Architekturstrukturen für die Anwendung benötigt werden. Die negativen Wechselwirkungen des Transformationsoperators müssen betrachtet werden, um überflüssige Anwendungen zu vermeiden. Dies ist der Fall, wenn ein Transformationsoperator eine Qualitätseigenschaft negativ beeinflusst. Wenn diese bereits schlecht ist, so sollte die Anwendung dieses Operators vermieden werden.

Bei der Definition der Verhaltensspezifikation und der Evaluationsmodelle für die hinzugefügten Elemente ist darauf zu achten, dass sich durch struktur- und typ-generische Transformationsregeln die Verhaltensspezifikation und die Evaluationsmodelle bei jeder Instanz der Transformationsregel unterscheiden. Es ist daher ein Konstruktionsalgorithmus zu spezifizieren, welcher in Abhängigkeit der Transformationsregelinstanz die Verhaltensspezifikation und die Evaluationsmodelle bestimmt.

6.2.2 Notation der Architekturtransformationsregel

Für eine bessere Verständlichkeit der Transformationsoperatoren ist es sinnvoll, eine graphische Notation zur Verfügung zu stellen /Grunske 03c/, welche grundsätzlich den ersetzten Graphen, den einzufügenden Graphen und den Interfacegraphen darstellt. Für allgemeine Graphtransformationsregeln werden in /Göttler 92/, /Kaplan et al. 91/ und /Loyall, Kaplan 92/ mit der X- und der Δ-Notation bereits zwei graphische Notationen vorgeschlagen.
Für die Darstellung und Verständlichkeit von Architekturtransformationsoperatoren sind die beiden Notationen jedoch zu kompakt /Grunske 03c/. Daher wurde bei der praktischen Anwendung der Architekturtransformationen die T-Notation entwickelt, welche grundsätzlich auf der X- und der Δ-Notation aufbaut. Bei dieser Notation wird auf Redundanzfreiheit verzichtet und der Operator wird in einer praxisverständlichen Form präsentiert. Des Weiteren werden keine graph-, sondern die architekturspezifischen Symbole verwendet, welche in Abschnitt 5.2.3 bereits eingeführt wurden.
Ihren Namen verdankt die T-Notation dem grundsätzlichen Aufbau, welcher schematisch in der Abbildung 6-5 dargestellt wird.

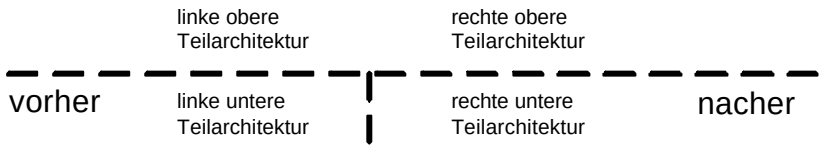


Abbildung 6-5 Graphische Darstellung der Graphtransformationsregeln in der T-Notation

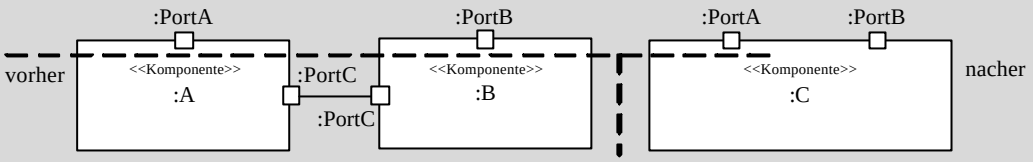
Durch die beiden gestrichelten Linien in Form eines T wird der Transformationsoperator in drei Teilgraphen unterteilt. Im linken, unteren Teilgraphen wird der Teil einer Architektur dargestellt, welcher bei der Anwendung des Operators entfernt wird. Im rechten unteren Teilgraphen wird der Teil der Architektur dargestellt, welcher bei der Anwendung des Transformationsoperators zur Architektur hinzugefügt wird. Die Architekturelemente oberhalb des T werden bei der Anwendung des Transformationsoperators nicht verändert. Sie sind invariant und repräsentieren somit die Klebepunkte zwischen den hinzugefügten oder den gelöschten Elementen der Architektur und der restlichen Architektur. Daraus folgt, dass sowohl der linke als auch der rechte untere Teilgraph mit den Elementen oberhalb des T verbunden ist. Zur besseren Verständlichkeit werden diese Elemente oberhalb des T deshalb redundant auf der linken und auf der rechten Seite dargestellt.
Aus einem Operator in der T-Notation ergibt sich der zu ersetzende Graph, der einzufügende Graph und der Interfacegraph wie folgt:

- Der zu ersetzende Graph entspricht dem linken unteren Teilgraphen zusammen mit dem linken Teilgraphen oberhalb des T.
- Der einzufügende Graph entspricht dem rechten unteren Teilgraphen zusammen mit dem rechten Teilgraphen oberhalb des T.
- Der Interfacegraph entspricht einem Teilgraphen oberhalb des T; da beide Teilgraphen identisch sind, ist es egal, welcher gewählt wird.

Durch die Abbildung der Teilgraphen auf den ersetzten Graphen, den einzufügenden Graphen und den Interfacegraphen ist jeder Operator in der T-Notation in eine Hypergraphtransformationsregel überführbar.

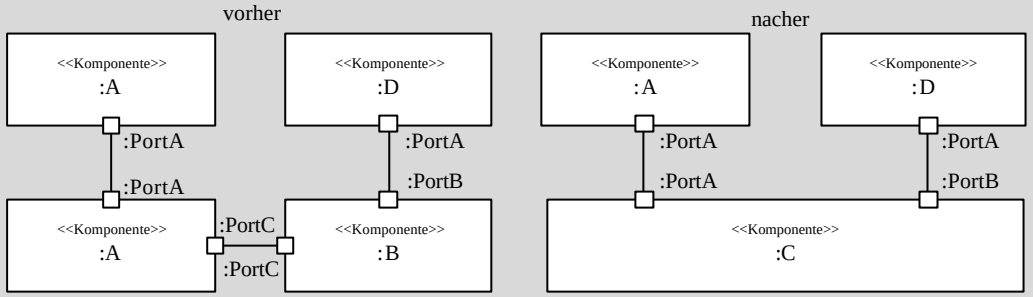
Beispiel 6-11

Als Beispiel soll der folgende Transformationsoperator dienen, welcher in der T-Notation spezifiziert ist:



Durch diesen Transformationsoperator werden die beiden Komponenten des Typs A und B, die zwei Ports des Typs C und die Verbindung zwischen den beiden Ports aus der Anwendungsarchitektur entfernt. Die Ports des Typs A und B oberhalb des T werden nicht verändert. Abschließend wird eine Komponente des Typs C zur Architektur hinzugefügt, welche mit den Ports oberhalb des T verbunden wird.

Eine Anwendung des Transformationsoperators auf eine konkrete Architektur wird im Folgenden dargestellt.



Struktur- und typ-generische Graphtransformationsregeln wurden eingeführt, um flexiblere Transformationsoperatoren zu ermöglichen. Zur Darstellung der Struktur- und Typgenerizität ist die graphische Notation eines Transformationsoperators noch zu erweitern.
Für struktur-generische Graphtransformationsregeln wurden drei Punkte als Symbol dafür verwendet, dass ein Knoten, eine Hyperkante oder ein Hypergraph mehrfach auftreten darf. Für die Notation eines Transformationsoperators kann dieses Symbol auch auf architekturspezifische Elemente übertragen werden. Die Strukturgenerizität ist somit für Architekturelemente, Interfaceelemente, Verbindungen, Verschaltungen und Ausführungsverschaltungen, wie in der folgenden Tabelle dargestellt, spezifizierbar.

Tabelle 6-2 Graphische Repräsentation von strukturgenerischen Elementen in der ADL COOL

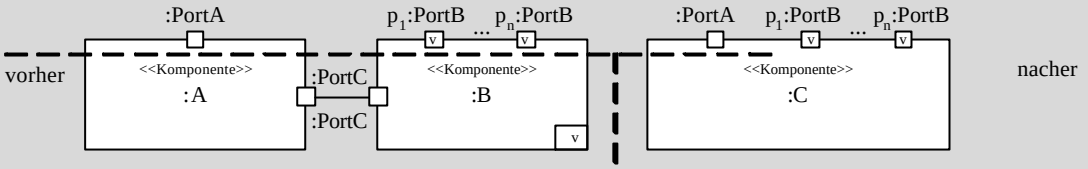
COOL Element	Graphische Repräsentation der Strukturgenerizität
Komponente, Sensor, Aktor, Hardwareplattform, Architektur, Hardware und Softwareelement	<<COOLStereotyp>> :A ... <<COOLStereotyp>> :A
Konnektor, Hardwarekanal	<<COOLSterotyp>> :A ... <<COOLSterotyp>> :A
Port, Abwicklerknoten und Ausführungsknoten	□ ··· □
Role	○ ··· ○
Verschaltung, Bindung und Ausführungsverschaltung	⋮

Für die Spezifikation von Variablen wird die Notation eines Transformationsoperators durch das Symbol *v* erweitert. Dieses Symbol ist mit einem Interfaceelement oder Architekturelement assoziierbar und markiert

das entsprechende Element als Variable. Als Konvention für die Annotation sollte das Symbol *v* für Interfaceelemente zentriert und für Architekturelemente links unten erfolgen.
Sollen bei der Anwendung eines Transformationsoperators zwei oder mehrere Variablen mit dem gleichen Typen instanziiert werden, so ist der Transformationsoperator mit Regeln zu erweitern, welche die Variableninstanziierung einschränken. Eine solche Regel spezifiziert daher eine Menge von Variableninstanzen, welche bei der Operatoranwendung identische Typen aufweisen müssen. Notiert wird diese Menge durch einen String, der zur Separierung der Variableninstanzen das Symbol „=“ verwendet.

Beispiel 6-12

Zur Verdeutlichung der Notationserweiterung wird der Architekturtransformationsoperator aus dem vorherigen Beispiel verwendet und wie folgt um struktur- und typgenerische Aspekte erweitert:



Durch die Erweiterung ist jeder Subtyp der Komponente *B* identifizierbar, welcher eine beliebige Anzahl von Ports enthält. Diese Ports sind ebenfalls Variablen und können von einem beliebigen vom PortB abgeleiteten Typen sein. Sollen dabei alle Variableninstanzen vom gleichen Typ sein, so wird dies durch die folgende Regel für die Variableninstanziierung spezifiziert: $p_1=,...,=p_n$

6.2.3 Notation für die Anwendungsbedingungen

Für die strukturierte Anwendung von Architekturtransformationsoperatoren werden grundsätzlich die folgenden Anwendungsbedingungen benötigt:
- Anwendungsbedingungen, welche den Kontext der Regelanwendung einschränken
- Anwendungsbedingungen, welche die Regelanwendung hinsichtlich der Ausprägung COOL-spezifischer Attribute einschränken

Die Anwendungsbedingungen hinsichtlich des Kontextes fordern die Existenz oder Abstinenz von COOL-spezifischen Elementen oder Strukturen in der Anwendungsarchitektur. Sie lassen sich als positive oder negative Kontextbedingungen, wie in Abschnitt 6.1.2 dargestellt, spezifizieren.
Durch die Einschränkungen hinsichtlich der Ausprägung COOL-spezifischer Attribute ist eine Anwendung eines Transformationsoperators verhinderbar, wenn ein Architekturelement bestimmte Eigenschaften nicht aufweist. Als Beispiel kann eine solches COOL-spezifisches Attribut die Erfüllung der Zuverlässigkeitseigenschaften eines Architekturelementes beschreiben. In diesem Fall ist mit einer Anwendungsbedingung ein Transformationsoperator spezifizierbar, welcher nur anwendbar ist, wenn die Zuverlässigkeitseigenschaften des Architekturelementes nicht erfüllt werden.
Zur einheitlichen Spezifikation beider Anwendungsbedingungstypen werden im Folgenden prädikatenlogische Formeln verwendet. Diese ermöglichen eine einfache Beschreibung der Anwendungsbedingungen. Des Weiteren lassen sie sich vor jeder Anwendung eines Transformationsoperators werkzeugunterstützt überprüfen.
Zur besseren Identifizierung werden die prädikatenlogischen Formeln durch ein Präfix erweitert. Dieses beginnt mit dem Buchstaben „A“ für Anwendungsbedingung gefolgt von einer Nummerierung. Demnach ist die erste Anwendungsbedingung eines Transformationsoperators „A1“. Für jede weitere Anwendungsbedingung wird die Zahl inkrementiert. Darüber hinaus wird durch das Präfix dargestellt, ob es sich um eine positive oder negative Anwendungsbedingung handelt. Dies wird durch ein hochgestelltes $^{+}$ oder $^{-}$ symbolisiert. Abschließend werden durch das Präfix optionale und essenzielle Anwendungsbedingungen unterschieden. Dabei müssen essenzielle Anwendungsbedingungen immer erfüllt werden. Bei einer optionalen Anwendungsbedingung kann der Transformationsoperator in Ausnahmefällen auch angewendet werden, wenn die Anwendungsbedingung nicht erfüllt ist. Essenzielle Anwendungsbedingungen werden dabei durch ein hochgestelltes ^{e} und optionale durch ein hochgestelltes ^{o} symbolisiert.
Zusammenfassend hat eine Anwendungsbedingung den folgenden Aufbau:
A[laufende Nummer]$^{[-/+][e/o]}$:prädikatenlogischer Ausdruck

Bei der Spezifikation der Anwendungsbedingung können die in Tabelle 6-3 dargestellten Basisprädikate genutzt werden. Diese erleichtern die Spezifikation von konkreten Anwendungsbedingungen. Im Besonderen lassen sich die Basisprädikate zur Spezifikation von Kontextbedingungen verwenden.

Tabelle 6-3 Auswahl von nutzbaren Basisprädikaten zur Spezifikation von Anwendungsbedingungen

Basisprädikat	Typ des Prädikates	Bedeutung
Verschaltet(A,B)	Strukturspezifisch	Zwischen den Architekturelementen A und B existiert eine Kommunikationsbeziehung
Typgleich(A,B)	Strukturspezifisch	Das Architekturelement A ist vom gleichen Typ wie Architekturelement B
Verhaltenskompatibel(A,B)	Strukturspezifisch	Die Verhaltensspezifikationen der Architekturelemente A und B sind kompatibel
IstSubType(S,P)	Strukturspezifisch	Das Architekturelement S ist Subtype des Architekturelementes P
Enthält(K,T)	Strukturspezifisch	Das Architekturelement K enthält ein Architekturelement vom Typ T
SelbeAusführungs-Plattform(K1,K2)	Strukturspezifisch	Die Softwareelemente K1 und K2 werden auf derselben Ausführungsplattform ausgeführt
ErfülltSicherheits-AnforderungenNicht (AE)	Attributspezifisch	Das Architekturelement AE erfüllt seine Sicherheitsanforderungen nicht. D.h. es existiert eine Gefährdung, deren Auftrittswahrscheinlichkeit größer ist als die tolerierbare Auftrittswahrscheinlichkeit. Formal wird dies, in der ADL COOL, wie folgt spezifiziert: $\exists$ g $\in$ AE.Gefährdungen: g.Evaluationsmodell.THP>g.Anforderungen.THP
ErfülltVerfügbarkeits-AnforderungenNicht (AE)	Attributspezifisch	Das Architekturelement AE erfüllt seine Verfügbarkeitsanforderungen nicht. D.h. die mit den Evaluationsmodellen ermittelte Verfügbarkeit ist geringer als die geforderte Verfügbarkeit des Architekturelementes. Formal wird dies, in der ADL COOL, wie folgt spezifiziert: AE.Evaluationsmodell.Verfügbarkeit<AE.Anforderungen.Verfügbarkeit
ErfülltZuverlässigkeits-AnforderungenNicht (AE)	Attributspezifisch	Das Architekturelement AE erfüllt seine Zuverlässigkeitsanforderungen nicht. D.h. die mit den Evaluationsmodellen ermittelte Zuverlässigkeit ist geringer als die geforderte Zuverlässigkeit des Architekturelementes. Formal wird dies, in der ADL COOL, wie folgt spezifiziert: AE.Evaluationsmodell.Zuverlässigkeit<AE.Anforderungen.Zuverlässigkeit
ErfülltWartbarkeits-AnforderungenNicht (AE)	Attributspezifisch	Das Architekturelement AE erfüllt seine Wartbarkeitsanforderungen nicht. D.h. die mit den Evaluationsmodellen ermittelte Wartbarkeit ist geringer als die geforderte Wartbarkeit des Architekturelementes. Formal wird dies, in der ADL COOL, wie folgt spezifiziert: AE.Evaluationsmodell.Wartbarkeit< AE.Anforderungen.Wartbarkeit
ErfülltScheduling-AnforderungenNicht(K)	Attributspezifisch	Die Komponente K erfüllt ihre Schedulinganforderungen nicht. D.h. bei der Evaluation des Schedulingmodells wurde erkannt, dass die Komponente nicht schedulbar sind. Formal wird dies, in der ADL COOL, wie folgt spezifiziert: AE.Evaluationsmodell.Scheduling.Auslastung >1
ErfülltÖkonomische-AnforderungenNicht (AE)	Attributspezifisch	Das Architekturelement AE erfüllt seine ökonomischen Anforderungen nicht. D.h. die evaluierten Kosten sind höher als das Budget des Architekturelementes. Formal wird dies, in der ADL COOL, wie folgt spezifiziert: AE.Evaluationsmodell.Kosten< AE.Anforderungen.Kosten

6.2.4 Beispielhafte Spezifikation eines Transformationsoperators

Zum besseren Verständnis wird im Folgenden die detaillierte Vorgehensweise bei der Spezifikation eines Transformationsoperators am Beispiel des Transformationsoperators für das Muster *homogene Redundanz mit Mehrheitsentscheid* beschrieben. Dieses Muster wird verwendet, um die Zuverlässigkeit, Sicherheit und Verfügbarkeit eines Systems zu erhöhen /Douglass 99/. Dazu werden die Softwarekomponenten vervielfacht und auf unabhängigen Hardwareplattformen realisiert. Zusätzlich wird ein Architekturelement des Typs Voter/Replikator verwendet. Dieser Voter/Replikator leitet alle eingehenden Nachrichten an die vervielfachten Softwarekomponenten weiter, welche diese Nachrichten entsprechend ihrer Verhaltensspezifikation verarbeiten. Die erzeugten ausgehenden Nachrichten werden aber nur dann an die Umgebung übermittelt, wenn eine vorher festgelegte Anzahl von Softwarekomponenten identische Nachrichten sendet. In der Praxis wird dabei oft ein 2-aus-3 Mehrheitsentscheid verwendet, bei dem zwei der drei Komponenten identische Nachrichten senden müssen. Durch die Struktur wird die Zuverlässigkeit des Systems verbessert, wenn die

Zuverlässigkeit der Hardwareplattform der Komponente Voter/Replikator größer ist als die der homogenen Komponenten. Die Zuverlässigkeit der Übertragungskanäle zwischen den verteilten Hardwareplattformen muss dabei zusätzlich in Betracht gezogen werden.

6.2.4.1 Transformationsregel

Aus der allgemeinen Beschreibung des homogenen Redundanzmusters ist erkennbar, dass eine Komponente eines beliebigen Typs entfernt und durch eine Teilarchitektur bestehend aus einem Architekturelement des Typs Voter/Replikator und mehreren homogenen Komponenten vom Typ der entfernten Komponente ersetzt werden muss.
Daher enthält der linke untere Teilgraph der Transformationsregel eine Komponentenvariable, welche bei der Anwendung des Operators aus der Architektur entfernt wird. Diese Komponentenvariable kann eine beliebige Anzahl von Ports haben, welche ebenfalls als Variablen spezifiziert werden. Jeder Port stellt einen Klebepunkt zu der restlichen Architektur dar und gehört somit zum linken und zum rechten oberen Teilgraphen. Zur Beschreibung der Hardwareaspekte ist es erforderlich, im linken unteren Teilgraphen ebenfalls die Hardwareplattform und die Ausführungsbeziehung zu spezifizieren. Da die Hardwareplattform von einem beliebigen Typ sein kann, wird diese ebenfalls als Variable dargestellt.
Der rechte untere Teilgraph enthält ein Architekturelement des Typs *Voter/Replikator*. Dieses Architekturelement assoziiert die Ports des rechten oberen Teilgraphen. Neben dem *Voter/Replikator* enthält der rechte untere Teilgraph eine Menge von homogenen Komponenten vom Typ der Komponentenvariable des linken unteren Teilgraphen und eine Menge von Hardwareplattformen. Die Mächtigkeit beider Mengen ist aus Flexibilitätsgründen generisch und wird erst bei der Instanziierung festgelegt. Für die Vollständigkeit des linken unteren Teilgraphen sind zusätzlich die Kommunikations- und Ausführungsbeziehungen zu spezifizieren. Die Kommunikationsbeziehungen werden durch Verschaltungen zwischen den einzelnen Ports der redundanten Komponenten und den replizierten Ports des Architekturelements Vorter/Replikator repräsentiert. Die Ausführungsbeziehungen werden durch jeweils eine Ausführungsverschaltung zwischen einer homogenen Komponente und einer Hardwareplattform bestimmt.
Zusammenfassend ergibt sich für den Transformationsoperator Homogene Redundanz mit Mehrheitsentscheid die in Abbildung 6-6 dargestellte Transformationsregel.

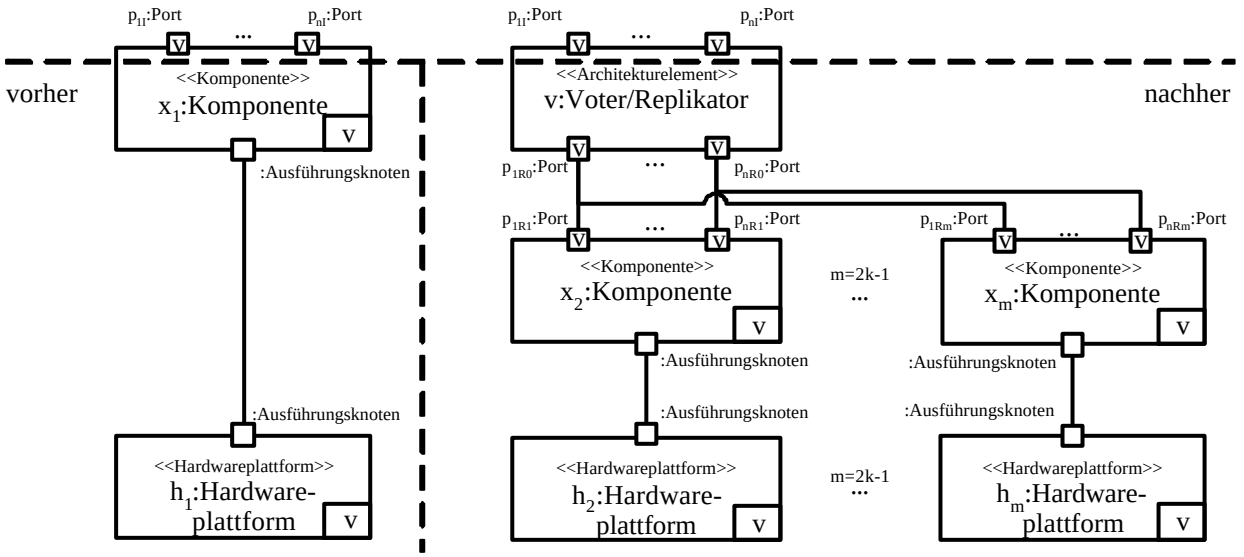


Abbildung 6-6 Transformationsregel: Homogene Redundanz mit Mehrheitsentscheid

Variableninstanziierung:
$h_1=h_2=...=h_m$
$x_1=x_2=...=x_m$
$p_{1I}=p_{1R0}=p_{1R1}=\ldots=p_{1Rm}$
...
$p_{nI}=p_{nR0}=p_{nR1}=\ldots=p_{nRm}$

6.2.4.2 Anwendungsbedingung

Für die Ermittlung der Anwendungsbedingungen müssen die Ziele des Transformationsoperators, die Kontexteinschränkungen und die negativen Wechselwirkungen betrachtet werden. Zu den Zielen der homogenen Redundanz mit Mehrheitsentscheid zählt nach der allgemeinen Beschreibung des Architekturmusters die Verbesserung der Zuverlässigkeit, Verfügbarkeit und Sicherheit einer Architektur. Vor der Anwendung des Transformationsoperators muss daher sichergestellt werden, dass durch die Komponente x_1 entweder die Zuverlässigkeitsanforderungen, Verfügbarkeitsanforderungen oder Sicherheitsanforderungen verletzt werden. Eine Verletzung der Zuverlässigkeitsanforderungen oder Verfügbarkeitsanforderungen kann durch einen Vergleich der jeweiligen Anforderungen der Komponenten und den Ergebnissen der Architekturevaluation ermittelt werden. Für die Verletzung der Sicherheitsanforderungen muss von den Komponenten eine Gefährdung verursacht werden, welche eine größere Auftrittswahrscheinlichkeit besitzt als sie durch die Anforderungen toleriert wird. Zusammenfassend lässt sich die zielgebundene Anwendungsbedingung durch Disjunktion der einzelnen Ziele beschreiben:

$A1^{+e}$: ErfülltVerfügbarkeitsAnforderungenNicht (x_1)
$\vee$ ErfülltZuverlässigkeitsAnforderungenNicht (x_1)
$\vee$ ErfülltSicherheitsAnforderungenNicht (x_1)

Durch kontexteinschränkende Anwendungsbedingungen wird ausgeschlossen, dass der Transformationsoperator mehrfach zur Lösung eines Problems angewendet wird. Eine solche mehrfache Anwendung kann durch die Existenz eines Architekturelementes *ae* vom Typ *Voter/Replikator* ermittelt werden, welche mit der Komponente x_1 kommuniziert. Für die Identifizierung der Kommunikationsbeziehung wird das Prädikat Verschaltet(x,y) verwendet. Die negative Kontextbedingung besitzt demnach den folgenden Aufbau.

$A2^{-e}$: $\exists$ ae$\in$ Anwendungsarchitektur:(ae.Typ=Voter/Replikator) $\wedge$Verschaltet(x_1,ae)

Durch die Anwendung des Transformationsoperators homogene Redundanz mit Mehrheitsentscheid werden zusätzlich Hardwareplattformen zur Architektur hinzugefügt. Dadurch erhöhen sich die Entwicklungskosten des zu entwickelnden Systems. Der Transformator sollte daher nur angewendet werden, wenn es das Entwicklungskostenbudget erlaubt. Dies kann durch eine optionale Anwendungsbedingung wie folgt ausgedrückt werden.

$A3^{-o}$: ErfülltÖkonomischeAnforderungenNicht(System)

6.2.4.3 Verhaltenspezifikation für die hinzugefügten Architekturelemente

Aus der Transformationsregel des Transformationsoperators homogene Redundanz mit Mehrheitsentscheid ist erkennbar, dass zur Architektur ein neues Architekturelement hinzugefügt wird. Dieses Architekturelement ist vom Typ *Voter/Replikator*, dessen Verhalten von den folgenden Aspekten abhängig ist:

- der Anzahl m der homogenen Komponenten
- dem Verhalten der homogenen Komponente (x_1)

Auf Grundlage dieser beiden Abhängigkeiten kann mit Hilfe des im Folgenden vorgestellten Algorithmus eine Interfacespezifikation automatisch generiert werden:

Schritt 1. Erzeuge einen temporären Interfaceautomaten für die ersetzte Komponente x_1

Zur Konstruktion der gewünschten Verhaltensspezifikation in Schritt 2 ist es erforderlich, dass der Interfaceautomat der zu ersetzenden Komponente aus aufeinanderfolgenden Sequenzen von Ein- und Ausgaben besteht /Grunske 03d/. Daher wird im ersten Schritt ein temporärer Interfaceautomat erzeugt, welcher dieser Normalform entspricht. Für die Konstruktion dieses temporären Interfaceautomaten werden die in Abbildung 6-7 spezifizierten Graphtransformationsregeln verwendet. In den ersten beiden Regeln wird dabei das neue Symbol „/“ eingeführt, welches in der Definition der Interfaceautomaten nicht verwendet wird. Dieses Symbol soll die Bezeichner zweier Aktionen voneinander trennen.

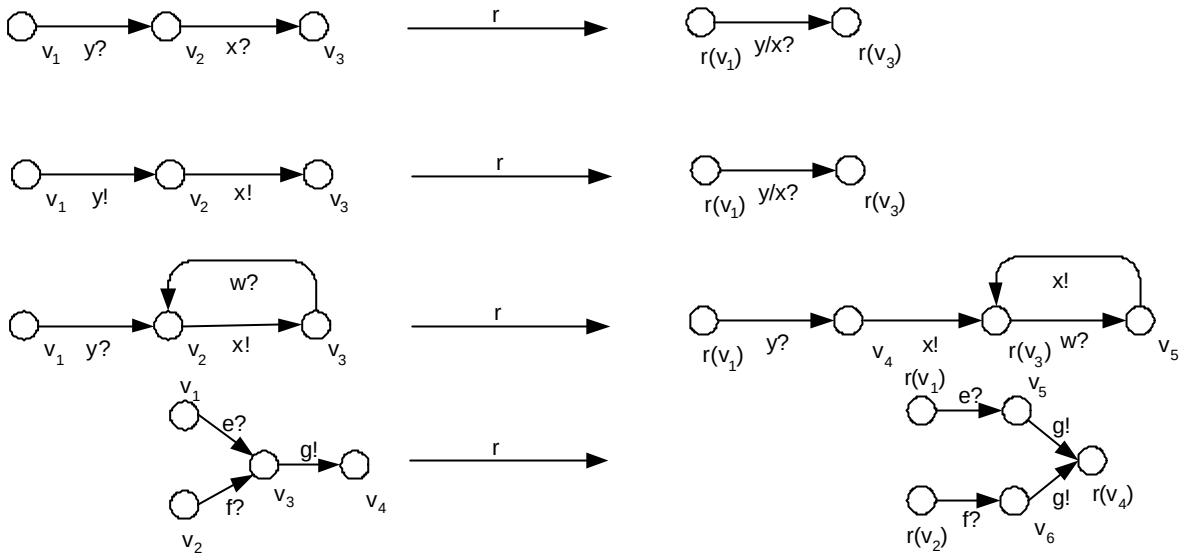
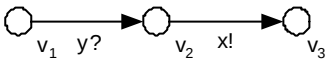


Abbildung 6-7 Graphtransformationsregeln zur Überführung eines Interfaceautomaten in eine Normalform

Schritt 2. Erzeuge das Mehrheitsentscheid-Verhalten für Anfragen an die ersetzte Komponente x_1

Ausgehend von der Normalenform muss jeder Teil des Interfaceautomaten der Form:



durch einen Interfaceautomaten ersetzt werden, welcher die Anfrage y? an die Komponenten weiterleitet und im Knoten v_3 vorfährt, sobald *n* der *m* Komponenten eine identische Nachricht gesendet haben.
Für die Konstruktion dieses Verhaltens müssen dazu jeweils die Schritte 2.1-2.3 wiederholt werden.

Schritt 2.1 Interaktionsbaum aufstellen

Im Schritt 2.1 wird ein so genannter Interaktionsbaum erstellt, dessen Knoten Zustände und dessen Kanten Zustandsübergänge im Interfacegraphen repräsentieren. In diesem Baum lassen sich durch die Pfade von der Wurzel zu den Blattknoten alle möglichen und zulässigen Ereignissequenzen des zu erstellenden n/m-Voters bestimmen. Jedem Knoten wird des Weiteren eine Ereignismenge zugeordnet, welche alle Ereignisse enthält, die im jeweiligen Pfad von der Wurzel zum Knoten bereits aufgetreten sind. Ausgehend von dieser Ereignismenge kann für jeden Knoten die Menge der ausgehenden Kanten und somit deren Unterknoten mit den folgenden Regeln bestimmt werden:

- Jeder Knoten enthält als ausgehende Kante eine Anfrage y_i! an eine der homogenen Komponenten. Der Index *i* ist dabei um eins größer als der größte Index der bisher in der Ereignismenge enthaltenen Anfrage. Sobald *i* größer ist als die Menge der homogenen Komponenten *m*, wird keine weitere Anfrage gesendet.
- Jeder Knoten enthält als ausgehende Kante eine Menge von Eingaben $\{x_j?\}$ der homogenen Komponenten. Dabei kann eine homogene Komponente *j* nur reagieren, wenn vorher eine Anfrage y_j! erzeugt wurde und vorher keine identische Eingabe x_j? erfolgt ist.
- Sind in einem Knoten *n* gleiche Antworten eingetroffen, so werden keine weiteren Antworten angenommen.

Ausgehend von diesen Regeln ergibt sich der folgende rekursive Algorithmus zur Konstruktion des Baumes:

Algorithmus ErzeugeKnoten(*n, m, s*)

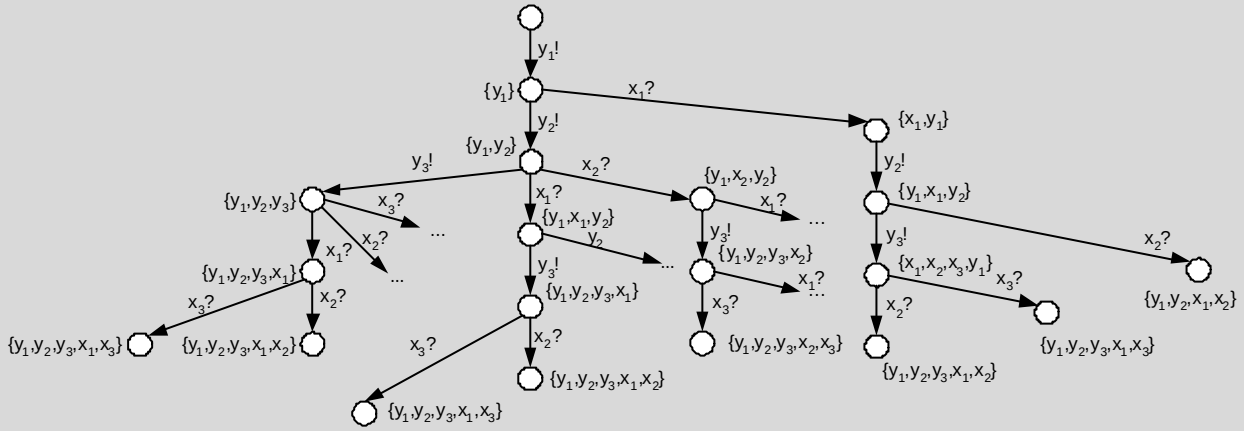
Input: Anzahl n identischer Antworten homogener Komponenten, Gesamtanzahl m der homogenen Komponenten, Ereignismenge s des zu erzeugenden Knoten,
Output: Knoten, der für jeden alternativen Zustandsübergang Kindknoten aufweist, denen die aus dem jeweiligen Zustandsübergang resultierende Ereignismenge zugeordnet ist.

1. Erstelle neuen leeren Knoten *k*
2. Ergänze Ereignismenge *s* zum Knoten *k*

3. Wenn die Anzahl der vorliegenden Ausgaben gleich n ist, dann
   gebe k zurück; *// Rekursionsende*
4. Wenn (e = Anzahl($a \in s \mid a$ ist Eingabe)) < m , dann *// Erzeuge ggf. eine weitere Eingabe*
   k_1 = ErzeugeKnoten($n, m, s \cup \{ y_{e+1}? \}$);
   Verbinde k_1 mit k und beschrifte die Kante mit y_{e+1}?
5. Für alle $y_i? \in \{ a_i \in s \mid a_i$ ist Eingabe und $x_i! \notin s \}$ *// Erzeuge Nachfolgeknoten für Ausgaben*
   k_i = ErzeugeKnoten($n, m, s \cup \{ x_i! \}$);
   Verbinde k_i mit k und beschrifte die Kante mit x_i!
6. Gebe k zurück;

Beispiel 6-13

Das folgende Beispiel demonstriert den erzeugten Baum für einen 2/3-Voter. Um die Übersichtlichkeit zu wahren, wurde der erstellte Baum an einigen Stellen durch Punkte („…“) abgekürzt.

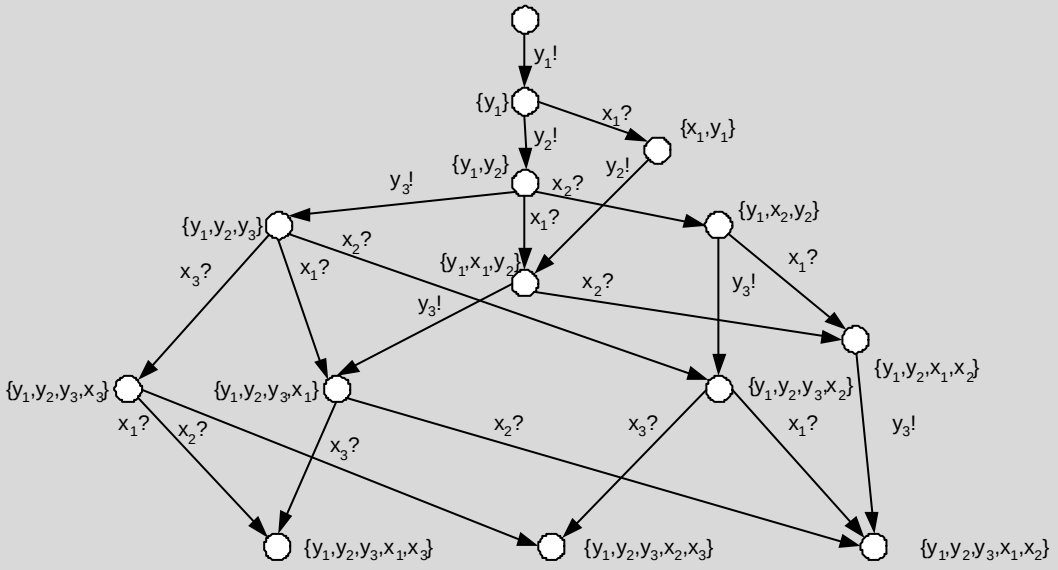


Schritt 2.2 Redundante Knoten zusammenfassen

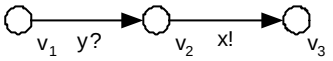
Im Schritt 2.2 werden alle Knoten mit gleichen Ereignismengen zusammengefasst. Dadurch werden die redundanten Knoten gelöscht und ein Graph erzeugt, dessen Knoten eindeutig durch ihre Ereignismenge identifizierbar sind.

Beispiel 6-14

Für den bisher konstruierten Baum ergibt sich der folgende Graph



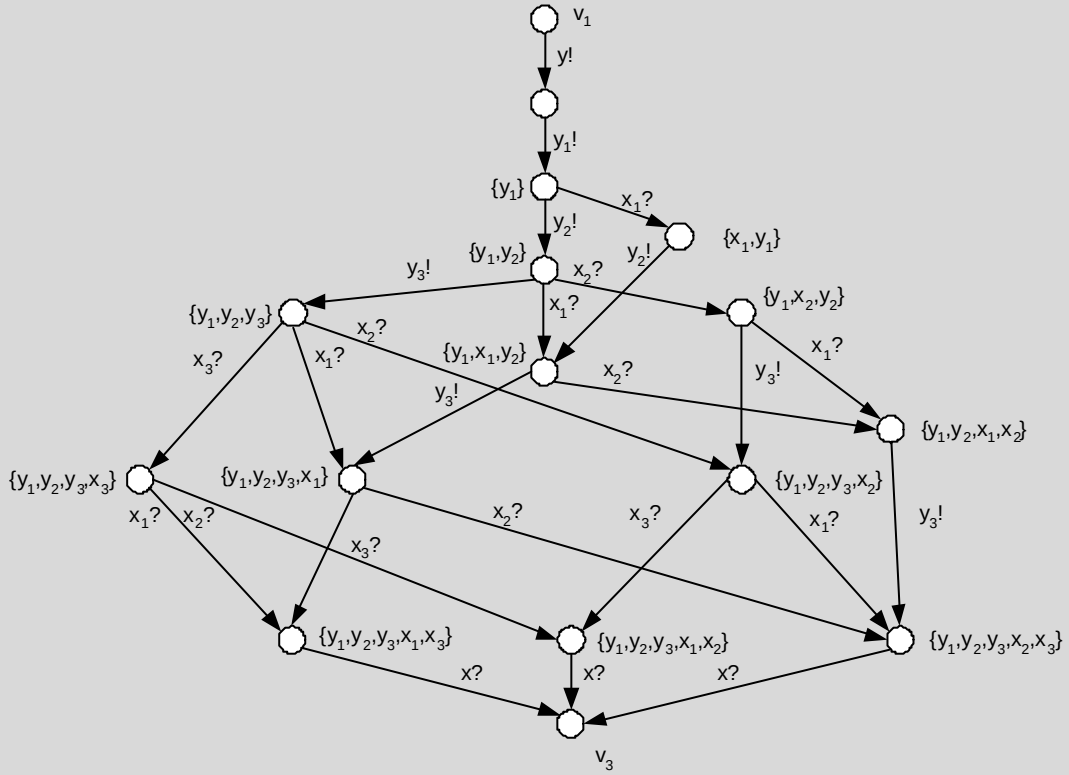
Schritt 2.3 Umwandlung in einen Interfaceautomaten
Im Schritt 2.3 wird der ursprüngliche Teil des Interfaceautomaten:



durch den in Schritt 2.2 erzeugten Graphen ersetzt. Für die Spezifikation der Ersetzungsregel muss dieser Graph jedoch noch erweitert werden, um die Klebepunkte zwischen den ersetzenden und den einzufügenden Graphen spezifizieren zu können. Bei dieser Erweiterung wird zum einen der Knoten v_1 eingefügt und über die Anfrage y? mit dem Wurzelknoten des Baumes verbunden. Zum anderen wird der Knoten v_3 eingefügt und eine Verbindung x! zwischen den ursprünglichen Blattkoten des Baumes und diesem Knoten erzeugt.

Beispiel 6-15

Für das bisher verwendete Beispiel ergibt sich der einzufügende Graph wie folgt:



Schritt 3 Umwandlung in einen regulären Interfaceautomaten
Im letzen Schritt wird aus der Normalform wieder ein regulärer Interfaceautomat erstellt. Dazu werden die beiden in Abbildung 6-8 enthalten Graphtransformationsregeln verwendet. Durch diese Regeln wird jedes Auftreten des Symbols „/“ wieder aus dem Interfaceautomaten entfernt.

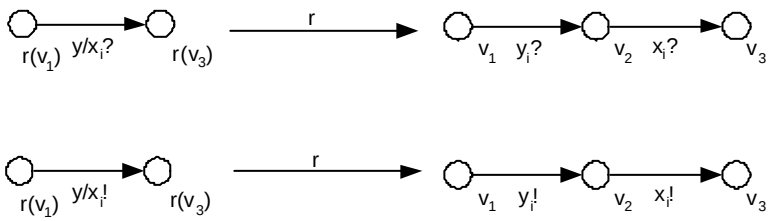


Abbildung 6-8 Graphtransformationsregeln zur Erstellung eines regulären Interfaceautomaten

6.2.4.4 Evaluationsmodelle für die hinzugefügten Architekturelemente

Ähnlich der Verhaltensspezifikation müssen für den Architekturelementtyp Voter/Replikator entsprechende Evaluationsmodelle erzeugt werden. Die konkrete Spezifikation dieses Modells hängt von den Evaluationsmodellen der homogenen Komponenten und deren Anzahl ab. Daher müssen diese ähnlich der Verhaltesspezifikation nach der Instanziierung des Transformationsoperators generiert werden.
Der Generierungsalgorithmus für Fehlerbäume ist trivial und basiert auf den beiden in Abbildung 6-9 dargestellten Fehlerbaummodulen. Diese Fehlerbaummodule müssen für jeden Ein- und Ausgabeport der homogenen Komponente zum Fehlerbaum des Architekturelementtypen Voter/Replikator hinzugefügt werden. Dabei wird für jeden Eingabeport das Fehlerbaummodul (a) und für jeden Ausgabeport das Fehlerbaummodul (b) verwendet. Die Anzahl *m* der erzeugten Ein- und Ausgabeports sowie der Parameter *n* des Votings hängen dabei von den Parametern des Transformationsoperators ab.

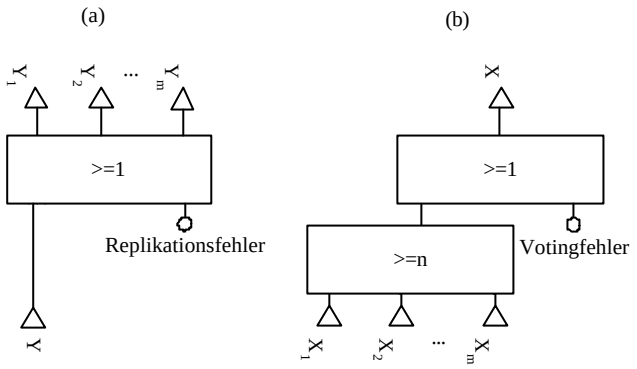


Abbildung 6-9 Konstruktionsmodule für den Fehlerbaum des Architekturelementtyp Voter/Replikator

6.3 Korrektheit von Architekturtransformationen

Für die automatische und werkzeugunterstützte Anwendung eines Architekturtransformationsoperators muss die Transformationsregel syntaktisch und semantisch korrekt sein. Die syntaktische Korrektheit einer Architekturtransformationsregel ist gegeben, wenn eine syntaktisch korrekte Softwarearchitektur nach der Anwendung der Regel immer noch syntaktisch korrekt und wohlgeformt ist. Eine Architekturtransformationsregel ist semantisch korrekt, wenn das Verhalten vor und nach der Regelanwendung identisch ist. Im Folgenden werden zunächst die notwendigen Einschränkungen der Transformationsregeln für die syntaktische Korrektheit dargestellt. Anschließend wird für die semantische Korrektheit ein Verfahren vorgestellt, welches die Verhaltensäquivalenz einer Softwarearchitektur vor und nach der Regelanwendung nachweist.

6.3.1 Syntaktische Korrektheit

Durch die Verwendung von algebraischen Hypergraphersetzungsregeln bestehen bereits einige Einschränkungen, welche die syntaktische Korrektheit des resultierenden Hypergraphen und somit der Softwarearchitektur sichern. Als Beispiel wird durch das Prüfen der *dangling condition* sichergestellt, dass nach der Regelanwendung keine nicht-assoziierten Hyperkantenenden existieren. Durch den *gluing*-Ansatz wird zudem sichergestellt, dass der resultierende Hypergraph syntaktisch korrekt ist, wenn der einzufügende Hypergraph syntaktisch korrekt ist /Habel 92/, /Drewes et al. 97/.
Für die syntaktische Korrektheit sind jedoch weitere Einschränkungen notwendig um die Wohlgeformtheit der resultierenden Architekturspezifikation sicherzustellen. Diese lassen sich durch zusätzliche Anwendungsbedingungen spezifizieren, welche vor jeder Anwendung einer Graphtransformationsregel überprüft werden. Für die Architekturbeschreibungssprache COOL wurden die folgenden zusätzlichen Anwendungsbedingungen identifiziert:

– Interfaceelemente des Interfacegraphen, welche mit einer Hyperkante des Typs *komplexe Hyperkante* oder *Modul* im ersetzenden Graphen verbunden sind, dürfen im Kontextgraph ausschließlich mit Hyperkanten des Typs *Verschaltung* und *Bindung* verbunden sein.

- Wird im ersetzten Graph eine Ausführungsverschaltung gelöscht, ohne dass das dazugehörige Softwareelement gelöscht wird, so muss bei der Anwendung der Regel eine neue Ausführungsverschaltung für das Softwareelement erzeugt werden.
- Wird im ersetzten Graph ein Softwareelement identifiziert, welches durch die Anwendung der Regel gelöscht wird, so muss auch die dazugehörige Ausführungsverschaltung gelöscht werden.

Durch die erste Anwendungsbedingung wird sichergestellt, dass in der resultierenden Softwarearchitektur keine zwei Hyperkanten des Typs *komplexe Hyperkante* oder *Modul* mit einem Knoten assoziiert sind. Die zweite Anwendungsbedingung gewährleistet, dass jedes Softwareelement eine Ausführungsverschaltung besitzt. Die dritte Anwendungsbedingung ist dazu notwendig, dass jede Ausführungsverschaltung einem Softwareelement zugeordnet ist.

6.3.2 Verhaltensäquivalenz

Für die semantische Korrektheit eines Architekturtransformationsoperators ist nachzuweisen, dass das Verhalten vor und nach der Regelanwendung identisch ist. Dies wird als Verhaltensäquivalenz bezeichnet.
Für die Bestimmung der Verhaltensäquivalenz gibt es grundsätzlich zwei Ansätze. Zum einen ist für jeden Transformationsoperator ein separater Nachweis zu erbringen, dass bei keiner Operatoranwendung das Verhalten verändert wird /Moriconi et al. 95/. Dieser Ansatz ist jedoch nur für einfache Transformationsoperatoren möglich. Ein genereller Nachweis der Verhaltensäquivalenz kann für struktur- und typgenerische Operatoren nicht erbracht werden, da erst bei der Regelanwendung die konkrete Ausprägung des Operators bekannt ist. Daher wird in dieser Arbeit der zweite Ansatz bevorzugt. Bei diesem wird ein allgemeiner Algorithmus angeben, welcher zur Laufzeit prüft, ob sich das Verhalten durch die Operatoranwendung verändert. Verwendet wird dazu ein Algorithmus, welcher die folgenden drei Grundschritte enthält:

1. Konstruiere den Interfaceautomaten I_L für den Teilgraphen der Architekturspezifikation, welcher durch den Architekturtransformationsoperator entfernt wird.
2. Konstruiere den Interfaceautomaten I_L für den Teilgraphen der Architekturspezifikation, welcher durch den Architekturtransformationsoperator hinzugefügt wird.
3. Prüfe die Äquivalenz beider Interfaceautomaten I_L und I_R.

Zur Konstruktion der Interfaceautomaten I_L und I_R wird der in Abschnitt 5.3.3 dargestellte Algorithmus verwendet. Dieser Algorithmus wird ursprünglich zur automatischen Bestimmung eines Interfaceautomaten für ein hierarchisches Architekturelement verwendet. Da ein Architekturelement in der COOL jedoch ein Hypergraph ist, lässt sich dieser Algorithmus auch für die Konstruktion eines Interfaceautomaten für einen Teilgraphen adaptieren, wenn der Teilgraph ein zusammenhängender Hypergraph ist.
Für die Äquivalenzprüfung im Schritt 3 wird ein Isomorphietest /Grunske 03d/ verwendet. Voraussetzung für die Anwendung des Isomorphietests ist jedoch, dass beide Interfaceautomaten in einer Normalform vorliegen /Grunske 03d/. In dieser Normalform wird durch den Interfaceautomaten ausschließlich das extern beobachtbare Verhalten beschrieben. Zur Konstruktion der Normalform werden daher alle Zustände fusioniert, welche ausschließlich durch interne Zustandsübergänge verbunden sind.

Definition 6.18 Reduktion von intern beobachtbaren Zuständen
Sei I ein Interfaceautomat $\langle S, S^{init}, E^I, E^O, E^H, T \rangle$ und s_a, s_b zwei Zustände, welche ausschließlich mit internen Ereignissen verbunden sind, dann kann der Interfaceautomat I zu I' reduziert werden. Wobei I' wie folgt bestimmt wird $\langle S' = S \backslash s_b, S'^{init} = S^{init} \backslash s_b, E'^I = E^I, E'^O = E^O, E'^H = E^H \backslash Action^H_{rem}, T' = s\,(T \backslash Step^H_{rem}) \rangle$, dabei ist:

- $Action^H_{rem}$ die Menge aller Ereignistypen, welche ausschließlich zwischen den Zuständen s_a and s_b genutzt werden
- $Step^H_{rem}$ ist die Menge der internen Zustandsübergänge zwischen den Zuständen s_a and s_b
- und s ist ein Umbenennungsoperator, welcher alle Zuständen s_b in s_a umbenennt.

Da in Abschnitt 5.3.1 Interfaceautomaten auf gerichtete, typisierte Hypergraphen abgebildet werden, lässt sich die Reduktion eines Interfaceautomaten auch durch Hypergraphtransformationsregeln beschreiben. Die Regel für das Entfernen von Zuständen, welche ausschließlich durch interne Ereignisse verbunden sind, wird durch die folgende Regel spezifiziert:

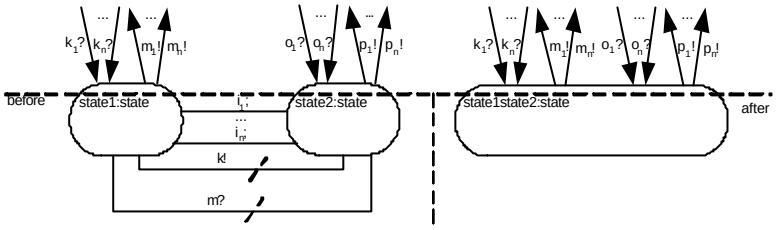


Abbildung 6-10 Graphtransformationsregel zur Reduktion eines Interfaceautomaten

Durch die Anwendung der Regel können zwischen zwei Zuständen Zustandsübergänge entstehen, welche identische Ereignisse beschreiben. Zur Entfernung dieser identischen Zustandsübergänge wird die folgende Regel verwendet:

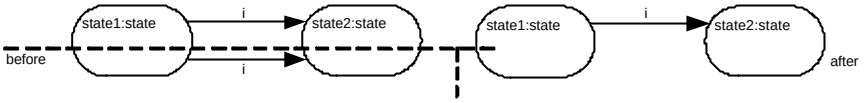
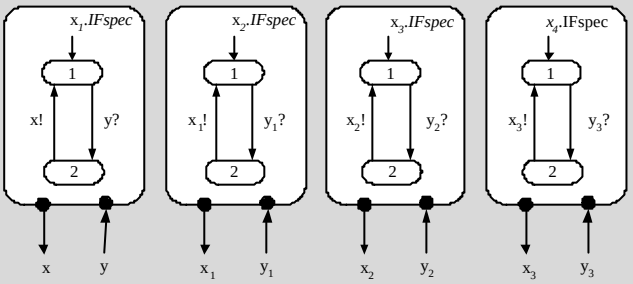


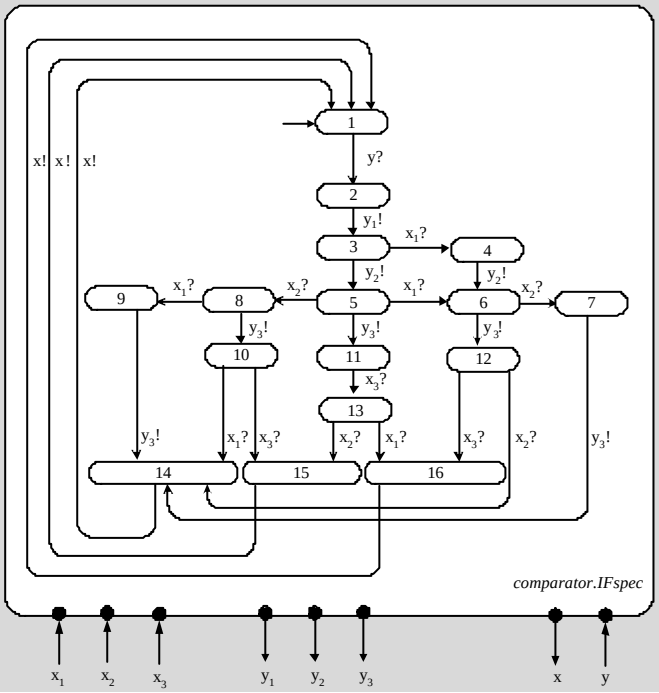
Abbildung 6-11 Graphtransformationsregel zum Löschen von mehrfachen redundanten Zustandsübergängen

Beispiel 6-16

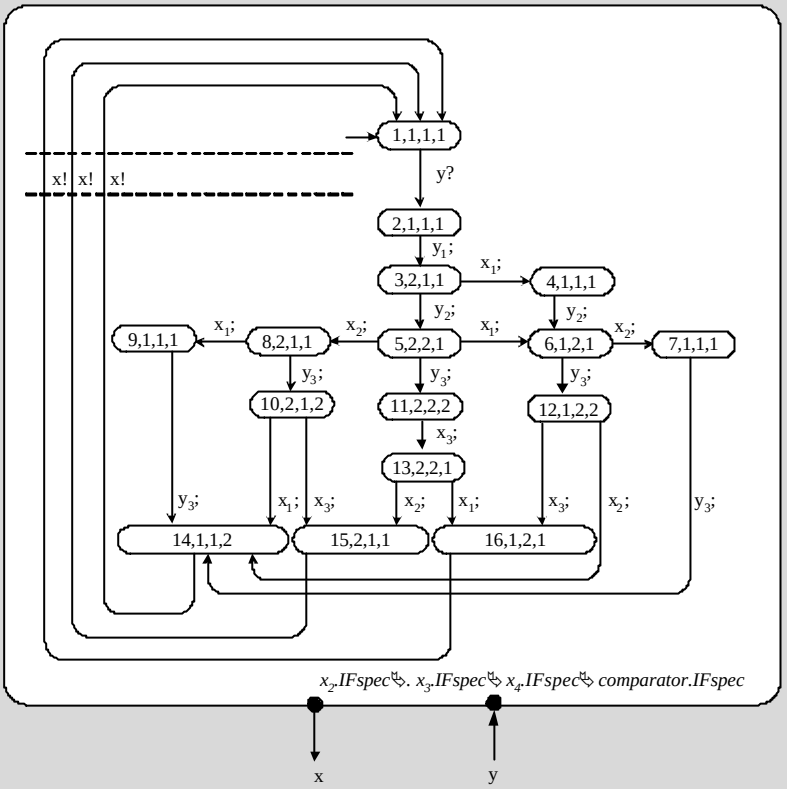
In dem bereits dargestellten Beispiel des Transformationsoperators der homogenen Redundanz mit Mehrheitsentscheid werden für die Komponenten x_1, x_2, x_3, und x_4 die folgenden Interfacespezifikationen angenommen. Diese Interfacespezifikationen sind bis auf die Benennungen der Ereignisse identisch.



Für die korrekte Realisierung des Mehrheitsentscheids wird die Interfacespezifikation der Komparatorkomponente v wie folgt erzeugt.



Ausgehend von der Spezifikation des Transformationsoperators entspricht der Interfaceautomaten I_L aller zu entfernenden Architekturelemente der Interfacespezifikation der Komponente x_1. Der Interfaceautomat I_R für die hinzugefügten Architekturelemente wird durch die Komposition der Interfacespezifikationen der Architekturelemente v, x_2, x_3, und x_4 bestimmt und entspricht somit dem folgenden Interfaceautomaten:



Bei der Reduktion auf das extern beobachtbare Verhalten werden alle Zustände oberhalb der ersten und unterhalb der zweiten gestrichelten Linie zu jeweils einem Zustand zusammengefasst. Zusätzlich werden die redundanten Ereignisse zwischen den beiden Zuständen entfernt. Dadurch ist der reduzierte Interfaceautomat I_R bis auf die Zustandsnamen identisch mit dem Interfaceautomaten I_L. Sie sind demnach isomorph.

6.4 Qualitätsverbessernde Transformationsoperatoren

Für die Realisierung des automatisierten SOQA-Prozesses ist es erforderlich, einen Satz von qualitätsverbessernden Transformationsoperatoren zu erstellen, die automatisch auf eine Architekturspezifikation in der Architekturbeschreibungssprache COOL anwendbar sind. Im vorangegangenen Abschnitt wurde dazu der generelle Aufbau eines Transformationsoperators sowie eine allgemeine Vorgehensweise zur Spezifikation der Transformationsoperatoren vorgestellt. Ausgehend davon wurden 11 Transformationsoperatoren erstellt, deren Grundlage bekannte Architekturentwurfsmuster /Douglass 99, 02/, /Saridakis 02/, /Grunske 03a/ bilden. Diese Architekturentwurfsmuster eignen sich im Besonderen für die Entwicklung von softwareintensiven technischen Systemen, sowie für die Verbesserung der Qualitätseigenschaften Sicherheit, Verfügbarkeit, Zuverlässigkeit und Wartbarkeit. Somit stellen sie einen Teil des Erfahrungswissens für die Qualitätsverbesserung im Architekturentwurf dar.

Ein Überblick über die erstellten Transformationsoperatoren wird in Tabelle 6-4 gegeben. Für die ausführliche Beschreibung der einzelnen Transformationsoperatoren wird auf den Anhang D verwiesen. Dort werden die erstellten Transformationsoperatoren ausführlich beschrieben und für jeden Transformationsoperator werden die Transformationsregeln, die Anwendungsbedingungen, die Verhaltensspezifikation aller hinzugefügten Architekturelementtypen und die Evaluationsmodelle aller hinzugefügten Architekturelementtypen spezifiziert.

Tabelle 6-4 Übersicht über die erstellen Transformationsoperatoren

Einfluss auf die Qualitätseigenschaften / Operatorname	Sicherheit	Zuverlässigkeit	Wartbarkeit	Verfügbarkeit	Echtzeitfähigkeit	Kosten
Mehrkanalige homogene Redundanz mit Mehrheitsentscheid	+	+/-	+	+/-	-	-
Zweikanalige homogene Redundanz	+	+	+	+	o/-	-
Mehrkanalige heterogene Redundanz mit Mehrheitsentscheid	+	+/-	+	+/-	-	-
Recovery Block	+	+	o	+	-	-
Watchdog	+	o	o	o	+	-
Heartbeat	+	+	o	+	o	-
Integritätscheck	+	+	o	o	-	-
Monitor	+	o	o	o	-	-
Hardwareelementaustausch	+/o	+/o	+/o	+/o	+/o	-
Neuzuweisung des Hardwareelementes	+/o	o	o	o	+	o
Vereinigung von Komponenten	+/-	o	o	o	+	+/-

Jedem Transformationsoperator wurde für jedes betrachtete Qualitätsmerkmal eine Bewertung gegeben, welche den Einfluss auf die Qualitätseigenschaften bei seiner Anwendung charakterisiert. Diese Bewertung basiert zum einen auf dem Erfahrungswissen bei der praktischen Anwendung des Transformationsoperators. Zum anderen wurde die Bewertung aus den Beschreibungen des zugrunde liegenden Architekturentwurfsmusters abgeleitet.
Die Bewertungsskala reicht dabei von einem + für einen positiven Einfluss bis zu einem - für einen negativen Einfluss auf die Qualitätseigenschaft. Eine exaktere Bestimmung der Beeinflussung der Qualitätseigenschaften kann vor der Anwendung der Architekturtransformation nicht erfolgen, da es auf den Kontext der Architekturtransformation ankommt, auf welche Weise sich die Qualitätseigenschaften der konkreten Architektur verändern.
Als Ergebnis der Bewertung der Transformationsoperatoren ist erkennbar, dass fast jedes Transformationsmuster die Kosten des zu erstellenden Systems negativ beeinflusst. Dies stützt die allgemein anerkannte These /Liggesmeyer 00,02/, /Meyer 97/, dass eine Verbesserung der Qualität mit einem erhöhten ökonomischen Aufwand verbunden ist.

6.5 Zusammenfassung

In diesem Kapitel wurde ein Formalismus beschrieben, der es ermöglicht, Architekturtransformationsoperatoren zu spezifizieren und automatisch auf eine Architekturspezifikation in der Architekturbeschreibungssprache COOL anzuwenden. Ein Transformationsoperator besteht dabei aus einer Ersetzungsregel, einer Menge von Anwendungsbedingungen, den Verhaltensspezifikationen für alle hinzuzufügenden Architekturelementtypen und den Evaluationsmodellen aller hinzuzufügenden Architekturelementtypen.
Die Grundlage für die Ersetzungsregeln bilden Hypergraphtransformationsregeln, welche sich mit dem double-pushout Ansatz automatisch anwenden und zurücknehmen lassen. Für die Spezifikation dieser Regeln wurde mit der T-Notation eine leicht verständliche graphische Notation vorgestellt. Mit dieser Notation lassen sich auch struktur- und typgenerische Graphtransformationsregeln spezifizieren, wodurch die Mächtigkeit dieser Graphtransformationsregeln erhöht wird.
Die Anwendungsbedingungen sind für die Auswahl eines geeigneten Transformationsoperators notwendig. Sie werden in Vor- und Nachbedingungen unterschieden und mit prädikatenlogischen Formeln spezifiziert. Für die Prüfbarkeit dieser Regeln mit einem Werkzeug wurden zudem Basisprädikate eingeführt, welche sich bei der Spezifikation von Architekturtransformationsoperatoren als sinnvoll erwiesen haben.
Ausgehend von den Verhaltensspezifikationen der hinzugefügten Architekturelementtypen ist die Überprüfung der Verhaltensäquivalenz einer Architektur vor und nach der Anwendung eines Transformations-

operators möglich. Dazu wurde für Interfaceautomaten ein genereller Algorithmus vorgestellt. Dieser Algorithmus bildet einen Kompositionsautomaten für alle gelöschten und alle hinzugefügten Architekturelemente. Anschließend werden die beiden Automaten auf das von außen sichtbare Verhalten reduziert. Sind beide Automaten isomorph, so ist die Verhaltensäquivalenz nachgewiesen und das Verhalten wird durch Transformation nicht verändert.
Die Evaluationsmodelle der hinzugefügten Architekturelementtypen werden für die Bestimmung der Qualitätseigenschaften der Architektur benötigt. Ausgehend davon kann bestimmt werden, ob eine Architekturtransformation erfolgreich war.
Sind alle Bestandteile eines Transformationsoperators korrekt spezifiziert, so ist der Transformationsoperator:

- automatisch auswählbar,
- automatisch anwendbar,
- reversibel
- und verhaltenserhaltend.

Somit erfüllt das in diesem Kapitel eingeführte Konzept der hypergraphbasierten Architekturtransformationsoperatoren die im Abschnitt 4.2.2 aufgestellte Anforderung A3 sowie deren Unteranforderungen für die Realisierung des automatisierten SOQA-Prozesses. Für einen praktischen Nachweis wurden elf Transformationsoperatoren spezifiziert. Die Grundlage dieser Transformationsoperatoren sind bewährte Architekturmuster, welche in /Douglass 02/, /Buschmann et al. 96/, /Randel 75/, /Saridakis 02/ und /Grunske 03a/ vorgestellt wurden und welche sich zur Verbesserung der Qualitätseigenschaften Sicherheit, Verfügbarkeit, Zuverlässigkeit, Wartbarkeit und Echtzeitfähigkeit eignen. Die Transformationsoperatoren lassen sich somit zur Optimierung dieser Qualitätseigenschaften im Architekturentwurf einsetzen.

7 Praktische Umsetzung

In den letzten beiden Kapiteln wurde mit der Architekturbeschreibungssprache COOL und den darauf anwendbaren qualitätsverbessernden Architekturtransformationsoperatoren die theoretische Umsetzbarkeit des Automatisierungsansatzes des SOQA-Prozesses demonstriert. In diesem Kapitel soll nun die praktische Realisierbarkeit dieses Ansatzes belegt werden. Dazu wird zunächst das Werkzeug BALANCE vorgestellt, welches mit dem Ziel entwickelt wurde, die Anwendung des automatisierten SOQA-Prozesses zu unterstützen. Des Weiteren werden drei industrielle Fallstudien vorgestellt, welche die industrielle Nutzbarkeit des Ansatzes demonstrieren. Diese Fallstudien wurden mit dem Werkzeug BALANCE durchgeführt.

7.1 Das Werkzeug BALANCE

Das Werkzeug BALANCE ermöglicht dem Anwender, mit einem Architektureditor eine Architekturspezifikation in der Architekturbeschreibungssprache COOL zu erstellen. Diese Architekturspezifikation kann mit Qualitätsanforderungen sowie mit den notwendigen Evaluationsmodellen für diese Qualitätsmerkmale annotiert werden. Somit kann die Erfüllung der Qualitätsanforderungen für die Architekturspezifikation mit kompositionsbasierten Evaluationstechniken automatisch bestimmt werden. Wird bei der Evaluation eine Verletzung der Qualitätsanforderungen erkannt, so werden die betroffenen Architekturelemente farblich hervorgehoben. Der Anwender kann nun manuell qualitätsverbessernde Transformationen durchführen oder sich vom Werkzeug geeignete Transformationsoperatoren vorschlagen lassen. Die Transformationsoperatoren lassen sich in einem Transformationsoperatoreditor spezifizieren. Die Auswahl sowie die automatisierte Anwendung erfolgen durch den Transformationsmanager. Die einzelnen Komponenten des Werkzeuges BALANCE werden im Folgenden vorgestellt.

7.1.1 Der Architektureditor

Der Editor für die Architekturspezifikation ermöglicht sowohl eine *top-down* als auch eine *bottom-up* Entwicklung der Architektur. Bei der *bottom-up*-Entwicklung müssen zunächst alle flachen Architekturelementtypen erstellt werden. Diese flachen Architekturelementtypen werden von den regulären COOL-Meta-Klassen *Komponente*, *Konnektor*, *System*, *Hardwareplattform*, *Hardwarekanal*, *Sensor* oder *Aktor* abgeleitet. Verwendet wird dazu ein Kommando des in Abbildung 7-1 dargestellten *Typerepository*, welches nach dem Start alle verfügbaren Meta-Klassen der Architekturbeschreibungssprache COOL enthält.

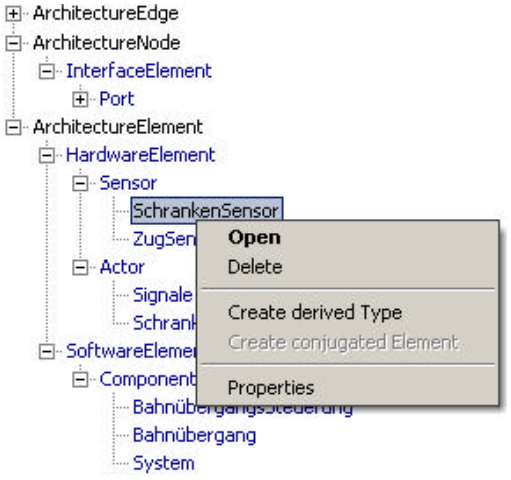


Abbildung 7-1 *Typerepository*

Für jeden flachen Architekturelementtypen werden anschließend die Interfaceelementklassen spezifiziert, welche zur Interaktion mit der Umgebung erforderlich sind. Dazu werden neue Interfaceelementklassen erzeugt, welche von den Meta-Klassen *Port* oder *Role* abgeleitet sind. Dies erfolgt ebenfalls über ein Kommando im Typerepository. Die Zuweisung eines Interfaceelementes zu einem Architekturelement erfolgt per Drag'n Drop. Dazu wird die entsprechende Interfaceelementklasse aus dem Typerepository ausgewählt und auf die Zeichenfläche des Architekturelementtypen gezogen. Bei einem flachen Architekturelementtypen werden durch das Werkzeug automatisch alle so zugewiesenen Interfaceelemente als extern betrachtet und durch eine Hyperkante vom Typ *Modul* verbunden.
Sind alle flachen Architekturelementtypen spezifiziert, so lassen sich ausgehend davon die hierarchischen Architekturelementtypen erzeugen. Diese werden ebenfalls im Typerepository erstellt und ihnen werden anschließend per Drag'n Drop die eingebetteten Architekturelemente zugewiesen. In der Zeichenfläche werden diese eingebetteten Architekturelemente inklusive der Interfaceelemente dargestellt, welche jedoch in dem hierarchischen Architekturelement als intern betrachtet werden. Soll zwischen den eingebetteten Architekturelementen eine Kommunikationsbeziehung bestehen, so lassen sich deren Interfaceelemente verschalten. Dazu wird eine Verschaltung aus dem *Typerepository* auf die Zeichenfläche gezogen, und die Andockpunkte an den Enden der Linie werden mit den jeweiligen Interfaceelementen verbunden. Für die Kommunikation mit der Umgebung des hierarchischen Architekturelementes werden externe Interfaceelemente benötigt. Diese werden ähnlich wie bei den flachen Architekturelementen im Typerepository erzeugt und auf der Zeichenfläche positioniert. Für die Verschaltung wird eine Bindung aus dem *Typerepository* auf die Zeichenfläche gezogen, und einer der beiden Andockpunkte wird mit dem externen Interfaceelement verbunden. Der andere Andockpunkt wird einem Interfaceelement eines eingebetteten Architekturelementes zugewiesen.
Für eine *top-down*-Entwicklung steht dem Bediener das in Abbildung 7-2 dargestellte Shapes-View-Fenster zur Verfügung.

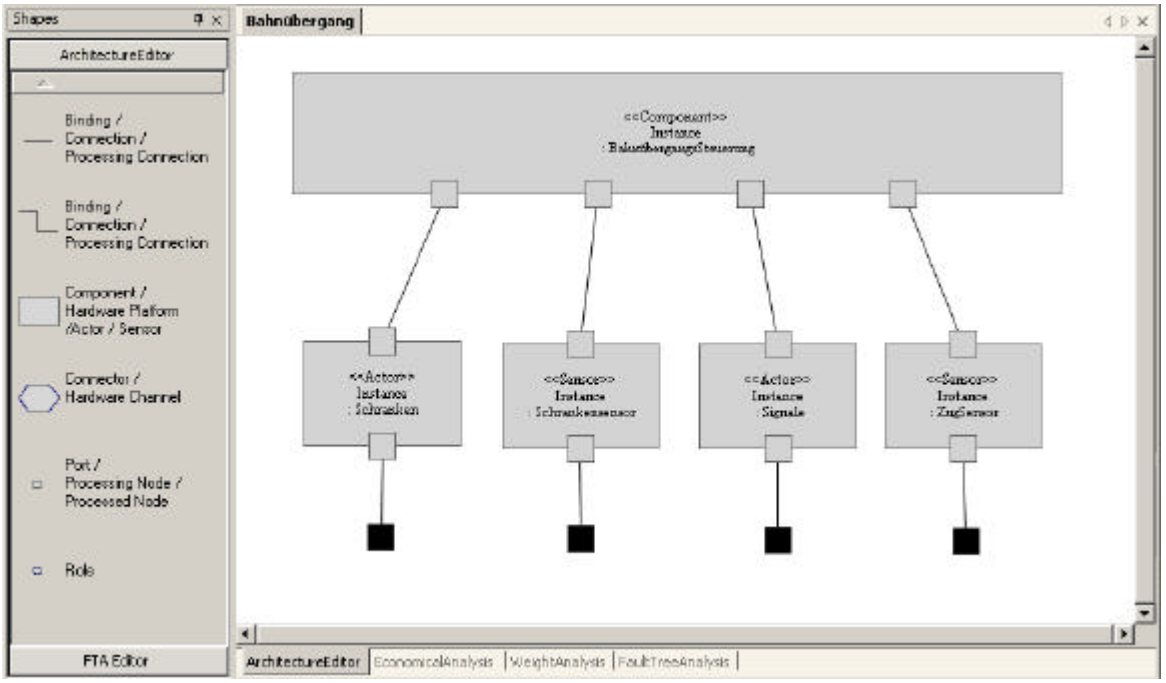


Abbildung 7-2 Shapes-View und Zeichenfläche

Dieses Fenster enthält für jede nutzbare COOL-Meta-Klasse ein graphisches Symbol. Wird ein solches Symbol per Drag'n Drop auf die Zeichenfläche gezogen, so wird ein neuer Architekturtyp von den COOL-Meta-Klassen abgeleitet. Anschließend wird eine Instanz des neuen Architekturtypen zum Architekturelement hinzugefügt und in dessen Zeichenfläche abgelegt.
Auf diese Weise lassen sich alle Instanzen und deren Typen erzeugen, welche zur Beschreibung eines komplexen Architekturelementtypen notwendig sind. Zur vollständigen Spezifikation müssen die Instanzen noch verschaltet werden. Dazu müssen zunächst alle erzeugten Architekturelementtypen durch Interfaceelemente erweitert werden. Dies erfolgt, indem in der Zeichenfläche ein Interfaceelement auf die Instanz des Architekturelementtypen gezogen wird. Dadurch wird der Architekturelementtyp verändert und das ent-

sprechende Interfaceelement in die Liste der externen Interfaceelemente hinzugefügt. Für eine syntaktisch korrekte Architekturspezifikation wird das hinzugefügte Interfaceelement zudem mit dem internen Modul verbunden. Diese Vorgehensweise ist daher nur bei flachen Architekturelementen anwendbar. Besitzt jedes Architekturelement die gewünschte Anzahl an Interfaceelementen, so können diese verschaltet werden. Dazu werden die Andockpunkte an den Enden einer Linie mit jeweils einem Interfaceelement verbunden. Sind beide Andockpunkte verbunden, so erkennt das Werkzeug in Abhängigkeit von den Interfaceelementtypen automatisch, ob die Linie eine *Abwicklungsverschaltung*, eine *Verschaltung* oder eine *Bindung* ist und fügt diese entsprechend zur Architektur hinzu. Ist die Spezifikation vollständig und verschaltet, so kann jedes der enthaltenen Architekturelemente weiter hierarchisch verfeinert werden. Dazu muss die Zeichenfläche des zu verfeinernden Typs geöffnet und, wie bereits beschrieben, weiter *top-down* bearbeitet werden.
Sowohl bei der *top-down* als auch *bottom-up*-Entwicklung lassen sich Architekturspezifikationen erstellen, welche zwar syntaktisch korrekt, jedoch nicht wohlgeformt sind. Daher enthält das Werkzeug eine Prüffunktion, welche die beschriebenen Wohlgeformtheitsregeln der Architekturbeschreibungssprache COOL überprüft. Werden von der Architekturspezifikation Wohlgeformtheitsregeln verletzt, so werden dem Anwender tabellarisch die Art und die Position der Verletzung angezeigt.
Für die Erleichterung der Spezifikation von anwendungsabhängigen Eigenschaften eines Elementes der Strukturspezifikation steht ein Eingabedialog zur Verfügung, welcher als Eigenschaftseingabefenster bezeichnet wird. Dieses ist in der Abbildung 7-3 dargestellt. Es kann unter anderem dazu genutzt werden, den Namen, die Supertypen, die Farbe und die Position eines Architekturelementes zu verändern.

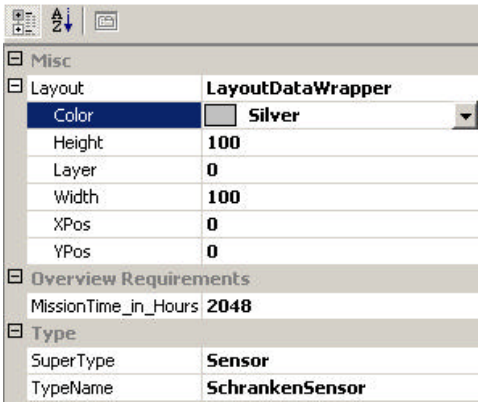


Abbildung 7-3 Eigenschaftseingabefenster

7.1.2 Die Architekturevaluation

Für die Architekturevaluation ermöglicht das Werkzeug BALANCE die Spezifikation der Qualitätsanforderungen und die Ermittlung der Qualitätseigenschaften auf Basis modularer Evaluationsmodelle. Als Qualitätsanforderungen lassen sich dabei Anforderungen an die in dieser Arbeit betrachteten Qualitätsmerkmale Sicherheit, Verfügbarkeit, Zuverlässigkeit, Wartbarkeit und Echtzeitfähigkeit sowie Kosten spezifizieren. Dies erfolgt für einzelne Qualitätsanforderungen mit einer Eingabemaske. Zur Anzeige aller Qualitätsanforderungen eines Architekturelementes wird eine Tabelle verwendet.
Zur Bestimmung der Qualitätseigenschaften einer Architektur mit den bereits beschriebenen kompositionsbasierten Evaluationstechniken ist es erforderlich, dass jedes flache Architekturelement mit einem geeigneten Evaluationsmodell annotiert wird. Im Werkzeug BALANCE wurde dazu bisher ein Fehlerbaumeditor realisiert. Der Fehlerbaumeditor ermöglicht die Modellierung der in Kapitel 5 vorgestellten Komponentenfehlerbäume. Die dazu verwendeten Symbole können per Drag'n Drop aus der in Abbildung 7-4 dargestellten Shapes-View auf die Zeichenfläche gezogen und anschließend verschaltet werden. Für die Eingabe der Auftrittshäufigkeit der internen Ereignisse steht dem Anwender ein angepasstes Eigenschaftseingabefenster zur Verfügung. Dieses kann auch zur Definition des Parameters *m* des *m/n* Gatters genutzt werden.

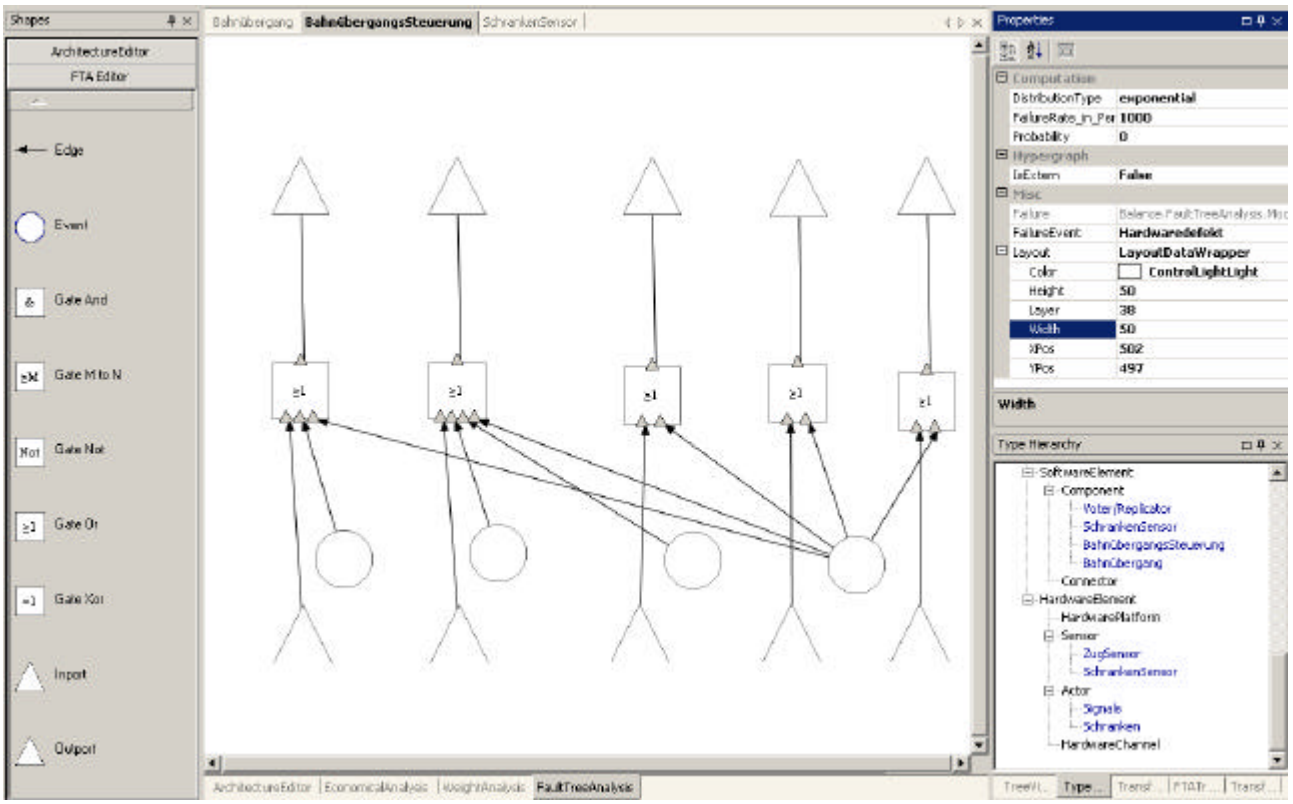


Abbildung 7-4 Annotierung eines Architekturelementes mit einem Komponentenfehlerbaum

Ausgehend von der Spezifikation der Evaluationsmodelle für alle flachen Architekturelemente ermöglicht das Werkzeug BALANCE eine Bestimmung der Qualitätseigenschaften einer Architekturspezifikation mit kompositionsbasierten Techniken. Für die Komponentenfehlerbäume wurde dazu der in Kapitel 5 vorgestellte Algorithmus umgesetzt. Dieser erzeugt rekursiv einen Komponentenfehlerbaum für die gesamte Architektur, in dem die Ausgabeports die Fehlverhalten des Systems charakterisieren. Für die Berechnung der Auftrittswahrscheinlichkeiten dieser Fehlverhalten wurde das Werkzeug UWG3 /Kaiser et al. 03/, /Grunske 03b/ genutzt, welches ebenfalls am Hasso-Plattner-Institut entwickelt wurde.

7.1.3 Der Transformationsoperatoreditor

Durch den Transformationsoperatoreditor wird dem Anwender die Möglichkeit gegeben, eigene Transformationsoperatoren zu erstellen. Dazu müssen die Transformationsregel und die Anwendungsbedingungen spezifiziert werden.
Für die Spezifikation der Transformationsregel ist es erforderlich, die einzelnen Architekturgraphen der Graphtransformationsregel zu erzeugen. Da diese grundsätzlich auch Architekturspezifikationen sind, basiert der Transformationsoperatoreditor auf dem Architektureditor. Erweitert wird er jedoch durch die Möglichkeit, Variablen und struktur-generische Elemente innerhalb einer Transformationsregel zu spezifizieren. Dazu kann für jede Instanz im Eigenschaftseingabefeld ein entsprechendes Flag gesetzt werden. Die graphische Repräsentation dieses Flag erfolgt für eine Variable durch das Symbol *v*. Dieses wird bei Interfaceelementen zentriert und bei Architekturelementen links unten angeordnet. Die graphische Repräsentation für struktur-generische Elemente erfolgt durch eine dreifach hintereinander geschachtelte Anordnung des entsprechenden Elementes.
Zur Spezifikation der Regeln für die Variableninstanziierungen werden die Variablen zusätzlich mit Variablennamen annotiert. Sind diese Variablennamen für zwei Architekturelemente identisch, so müssen bei der Regelanwendung auch identische Architekturelemente identifiziert werden.
Die Spezifikation der Anwendungsbedingungen erfolgt auf Basis von prädikatenlogischen Formeln. Diese können vom Anwender als Zeichenkette eingegeben werden. Dabei stehen Basisprädikate zur Verfügung, welche struktur- und parameterspezifische Bedingungen beschreiben, die für die komplette Anwendungsbe-

dingung entsprechend verknüpft werden müssen. Gibt der Anwender eine syntaktisch inkorrekte Formel ein, so weist das Werkzeug auf die Art und das Auftreten des Fehlers hin.
Zur Verdeutlichung der Spezifikation eines Transformationsoperators ist in Abbildung 7-5 ein Bildschirmabzug von BALANCE dargestellt. Dieser enthält den bereits vollständig spezifizierten Transformationsoperator Homogene Redundanz mit Mehrheitsentscheid.

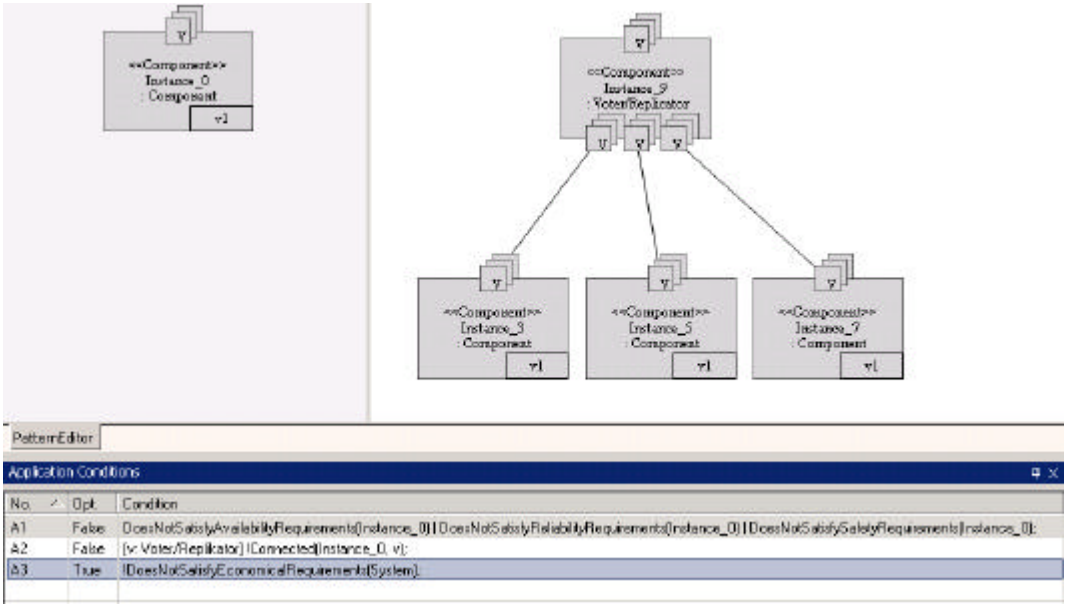


Abbildung 7-5 Transformationsoperatoreditor inklusive der Anwendungsbedingungen

7.1.4 Der Transformationsmanager

Die Auswahl und Anwendung der Transformationsoperatoren erfolgt im Werkzeug BALANCE durch den Transformationsmanager. Dieser prüft für jeden verfügbaren Transformationsoperator die Anwendungsbedingungen und versucht, den linken Teilgraphen der Ersetzungsregeln im Anwendungsgraphen zu identifizieren. Wird der Teilgraph identifiziert, so wird der entsprechende Transformationsoperator probeweise auf die Architektur angewendet. Dazu wird die Graphtransformationsregel mit dem *double-pushout*-Ansatz ausgeführt. In Abbildung 7-6 wird zur Veranschaulichung die Architektur des Bahnübergangsbeispiels nach der Anwendung des in Abbildung 7-5 spezifizierten Transformationsoperators auf den Schrankensensor dargestellt. Die hinzugefügten Elemente sind dabei farblich markiert.

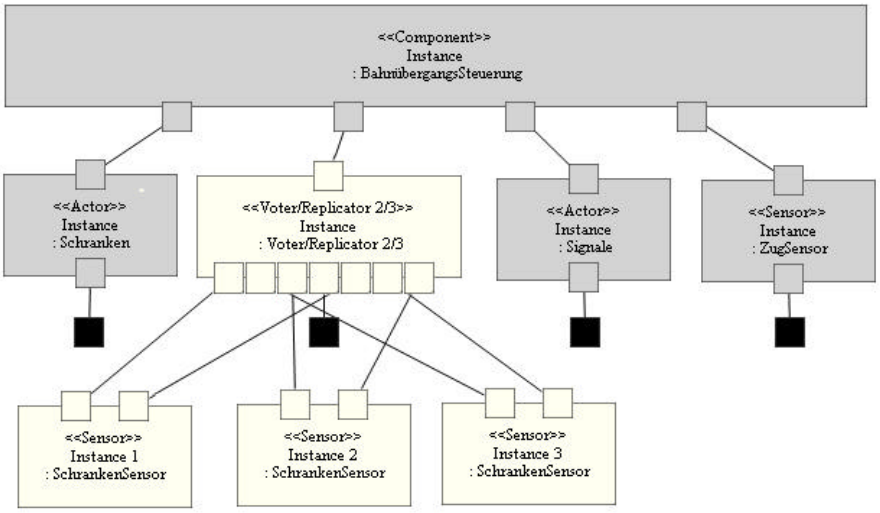


Abbildung 7-6 Architekturspezifikation der Anwendung eines Transformationsoperators

Nach der Anwendung überprüft der Transformationsoperatormanager, ob sich die Qualitätseigenschaften verbessert haben. Dazu wird eine erneute Architekturevaluation durchgeführt. Die Architekturtransformation wird anschließend wieder zurückgenommen und entweder ein weiteres Auftreten des linken Teilgraphen in der Anwendungsarchitektur gesucht oder ein neuer Transformationsoperator ausgewählt.
Auf diese Weise wird ein Suchbaum erstellt, welcher als Blattknoten die durch Transformation erzeugten Anwendungsarchitekturen und als Kanten die Transformationsoperatoren und deren Anwendungskontext enthält. Die Wurzel des Baumes entspricht der Startarchitektur. Die Architekturen der Blattknoten können wiederum als Ausgangspunkt für weitere Transformationen dienen. Somit realisiert BALANCE grundsätzlich eine Breitensuchstrategie. Diese kann vom Anwender parametrisiert werden.
Nach Durchführung der Transformationssuche werden dem Anwender in einer Tabelle die geeignetsten Architekturspezifikationen sowie die dazugehörigen Transformationsschritte präsentiert. Er besitzt die entgültige Entscheidungsbefugnis, welche Architekturtransformationen durchgeführt werden. Für die Auswahl enthält die Tabelle eine Auflistung der Qualitätseigenschaften, welche durch die Anwendung des Transformationsoperators verbessert oder verschlechtert wurden. Somit fungiert das Werkzeug BALANCE als Entscheidungsunterstützungssystem für den Architekten eines softwareintensiven technischen Systems.

7.2 Fallstudien

Neben dem bereits vorgestellten Beispiel des Bahnüberganges wurden drei industrielle Fallstudien in Zusammenarbeit mit der Siemens AG, der Deutschen Luft- und Raumfahrtgesellschaft (DLR) und dem Fraunhofer-Institut für Rechnerarchitektur und Softwaretechnik FIRST durchgeführt. Diese Fallstudien stammen aus unterschiedlichen Anwendungsbereichen, und die betrachteten Systeme besitzen unterschiedliche Anforderungen und Optimierungsziele. Dies zeigt die Universalität des Ansatzes und die Anwendbarkeit in verschiedenen Anwendungsbereichen.
Die einzelnen Fallstudien werden im Folgenden kurz vorgestellt und es wird ein Einblick in die Durchführung der Fallstudien gegeben. Abschließend werden die Ergebnisse der Fallstudien kritisch betrachtet.

7.2.1 Dampfturbinenschutz- und Turbinenschnellschlusssystem

Als erste Fallstudie wird ein System aus dem Anwendungsgebiet der Reaktorsicherheit betrachtet. Dieses System ist ein Schutzsystem, welches das Erreichen oder Überschreiten der Berstdrehzahl der Turbine verhindern soll. Da bei einem Bersten der Turbine eine hohe Gefährdung für die Umgebung besteht, besitzt das System entsprechende Sicherheitsanforderungen.
Das Erreichen der Berstdrehzahl kann in einem Reaktor in zwei unterschiedlichen Szenarien erfolgen. Im ersten Szenario wird der Reaktor durch Lastabschaltung vom frequenzstabilen Netz getrennt und der Turbine Frischdampf zugeführt. Im zweiten Szenario befindet sich der Reaktor im Inselbetrieb (Leerlaufbetrieb) und durch innere Anregung wird eine Überdrehzahl erzeugt.
Zur Vermeidung dieser beiden Szenarien besteht das Dampfturbinenschutz- und Turbinenschnellschlusssystem aus einem System zur Turbinenregelung (*Schnellgang TR*) und einem Turbinenschutzsystem (*TSS-System*). Die Turbinenregelung überprüft über Drehzahlsensoren die aktuelle Drehzahl der Turbine (*ÜbDrehzSchS Typ1* und *ÜbDrehzSchS Typ2*) und beeinflusst mit Stellventilen den Frischdampfzustrom (*FD-Einströmung*) zur Turbine. Somit kann die Drehzahl der Turbine in einem unkritischen Bereich gehalten werden. Neben der Turbinenregelung erhält auch das Turbinenschutzsystem die Drehzahlinformationen der Turbine. Befindet sich die Drehzahl in einem kritischen Bereich, so unterbindet das Schutzsystem über Schnellschlussventile sofort die Zufuhr von Frischdampf. Somit verhindert das Turbinenschutzsystem auch bei einem Ausfall der Turbinenregelung das Erreichen der Berstdrehzahl der Turbine.
Um eine Sicherheit gegenüber Einfachausfällen zu gewährleisten, wurden die Komponenten des Dampfturbinenschutz- und Turbinenschnellschlusssystems zweifach redundant ausgelegt. Diese Redundanz ist in den beiden Komponenten *Schnellgang TR* und *TSS-System* spezifiziert und in Abbildung 7-8 dargestellt. In Abbildung 7-9 wird des Weiteren die detaillierte Architektur der Komponenten *FD-Einströmung* dargestellt. Dabei ist erkennbar, dass die Frischdampfeinströmung durch ein Stellventil (*SteVtl*) oder durch eines von zwei Schnellschlussventilen (*SSV*) geregelt werden kann.

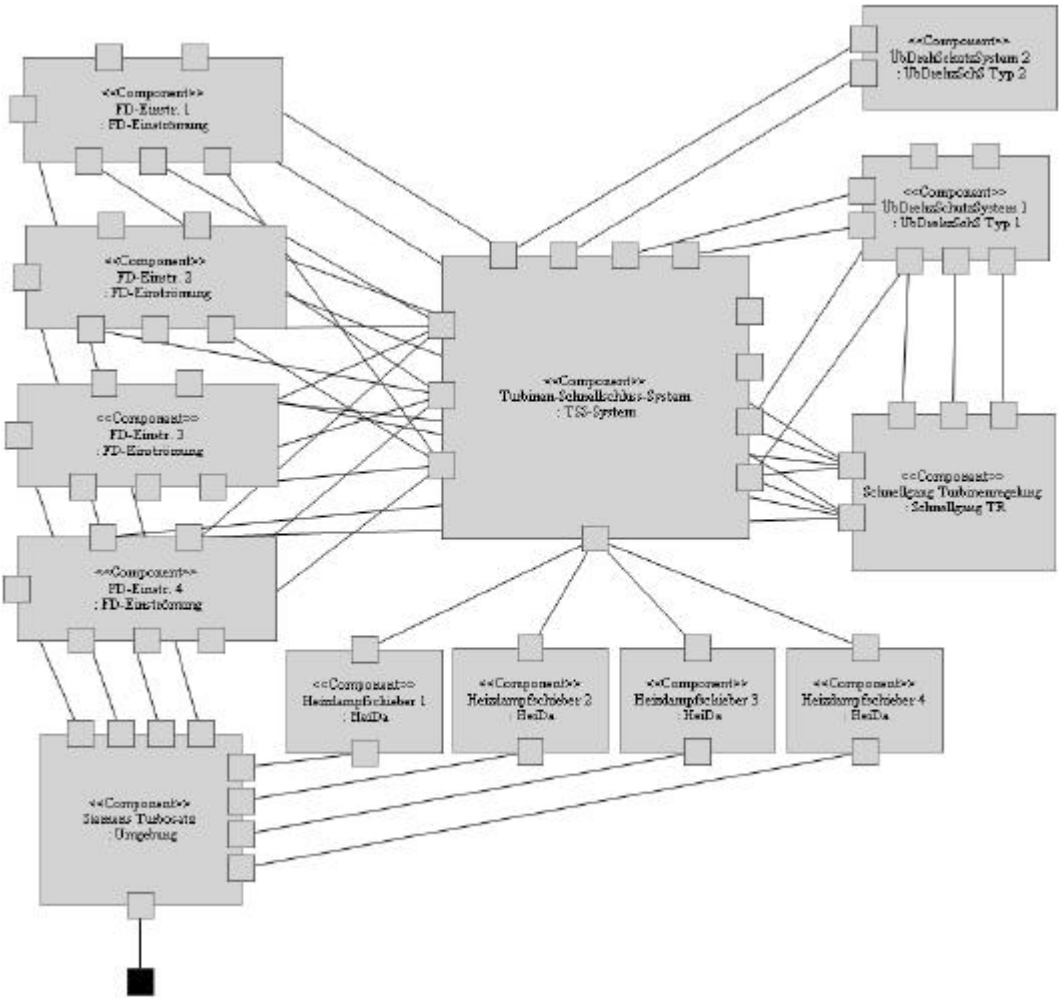


Abbildung 7-7 Strukturspezifikation des Dampfturbinenschutz- und Turbinenschnellschlusssystems

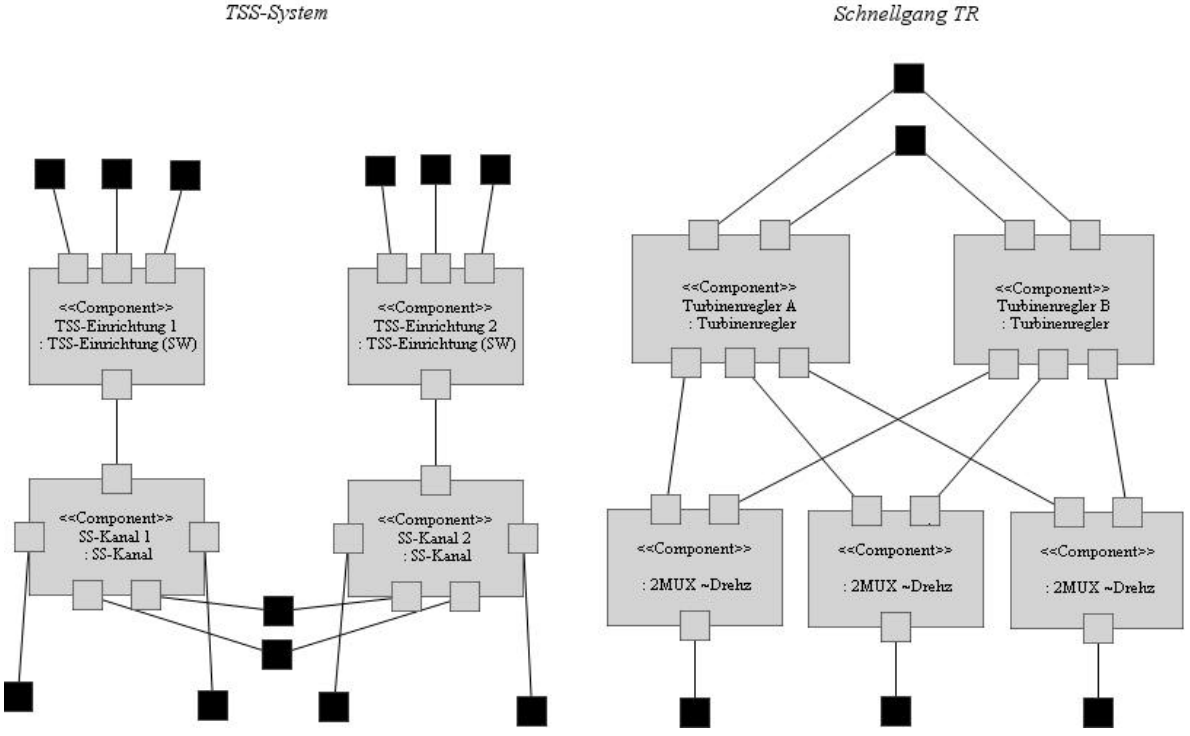


Abbildung 7-8 Detaillierte Architektur der Komponenten *Schnellgang TR* und *TSS-System*

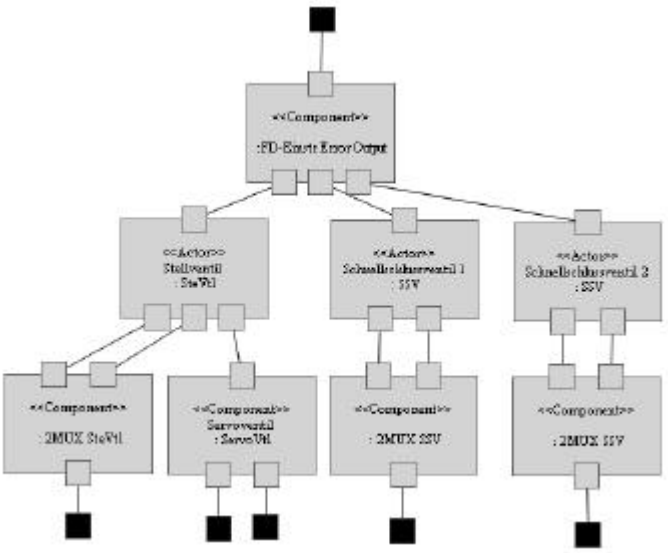


Abbildung 7-9 Detaillierte Architektur der Komponente FD-Einströmung

Zusätzlich zur Strukturspezifikation lagen für jedes Architekturelement ökonomische Daten und Fehlerbäume inklusive der Auftrittswahrscheinlichkeiten der Elementarfehler vor. Ausgehend von diesen Fehlerbäumen kann die Wahrscheinlichkeit bestimmt werden, mit welcher die Turbine die Berstdrehzahl erreicht. Auf dieser Grundlage wurde eine Optimierung der Sicherheitseigenschaften gegenüber den Erstellungskosten durch Anwendung des SOQA-Prozesses und des Werkzeuges BALANCE durchgeführt. Dabei wurden die in Tabelle 7-1 dargestellten Transformationsoperatoren für eine Verbesserung der Qualität der Architektur vorgeschlagen.

Tabelle 7-1 Geeignete Transformationsoperatoren zur Verbesserung der Qualität des Dampfturbinenschutz- und Turbinenschnellschlusssystems

Transformationsoperator	Anwendungskontext	Änderung der Sicherheitseigenschaft (Turbine erreicht die Berstdrehzahl nicht)	Änderung der ökonomischen Eigenschaften (Entwicklungskosten)
Homogene Redundanz mit Mehrheitsentscheid (Architekturelement)	Schnellschlussventil 1: *SSV*	+ 8,2 %	+ 2,2%
Zweikanalige homogene Redundanz	Schnellschlussventil 1: *SSV*	+ 3,9%	+1,8%
Homogene Redundanz mit Mehrheitsentscheid (Architekturelement)	Schnellschlussventil 2: *SSV*	+ 8,2 %	+ 2,2%
Zweikanalige homogene Redundanz	Schnellschlussventil 2: *SSV*	+ 3,9%	+1,8%
Homogene Redundanz mit Mehrheitsentscheid (Architekturelement)	Stellventil: *SteVtl*	+ 18,2%	+2,2%
Zweikanalige homogene Redundanz	Stellventil :*SteVtl*	+ 11,0%	+1,8%

Die vorgeschlagenen Architekturverbesserungen beziehen sich alle auf die Aktoren, welche die Dampfzufuhr in den beschriebenen gefährlichen Situationen regulieren könnten. Somit werden speziell die Schwachpunkte der Architektur behoben. Verbesserungen der Steuerungskomponenten und der Sensoren haben sich auf die Sicherheitseigenschaften nur wenig ausgewirkt. Der Grund dafür sind niedrige Ausfalloffenbarungszeiten und Redundanzkonzepte bei diesen Komponenten. Durch diese Eigenschaften ist die Wahrscheinlich-

keit, dass ein Fehlverhalten in den Steuerungskomponenten und den Sensoren zu einer Gefährdung führt, bereits sehr niedrig.

7.2.2 Türsteuerungssysteme in Personenverkehrszügen

Die zweite Fallstudie stammt aus dem Anwendungsgebiet des Schienenverkehrs und betrachtet ein Türsteuerungssystem in Personenverkehrszügen. Dieses System besitzt eine Sicherheitsfunktion, welche gewährleistet, dass die Türen geschlossen sind, wenn der Zug fährt. Die Verletzung der Sicherheitsfunktion kann zu einer Gefährdung beim Ein- und Ausstieg führen. Daher wird durch das Türsteuerungssystem eine Anfahrt des Zuges verhindert, wenn entweder die Türen noch offen sind oder eine Person oder ein Gegenstand in einer Tür eingeklemmt ist. Des Weiteren wird als zusätzliche Sicherungsfunktion ein Signal zum Schließen der Türen gegeben, wenn der Zug eine Geschwindigkeit höher als 5 km/h erreicht.
Wie in Abbildung 7-10 dargestellt, enthält das Türsteuerungssystem die Steuerungskomponenten *Zugsteuerungssystem*, *Bremssteuerungssystem* und *Antriebssteuerungssystem*. Diese Steuerungskomponenten erhalten Informationen über den aktuellen Zustand der Türen und über die Geschwindigkeit des Zuges von den Sensoren *Türsensor* und *Geschwindigkeitssensor*. Die Beeinflussung des Zuges bei offenen Türen erfolgt durch das Setzen einer Abfahrtssperre durch das *Antriebsteuerungssystem* und durch das Anlegen der Haltebremse durch das *Bremsteuerungssystem*. Außerdem wird dem Zugführer über eine *Warnlampe* mitgeteilt, dass eine der Türen offen ist. Zusätzlich zu den Systemkomponenten wurde die Umgebung des Zuges spezifiziert. Dies war notwendig, um externe Fehlereinflüsse zu beschreiben. Beispiele für solche Einflüsse sind Fehlverhalten des Zugführers oder des Zugbegleitpersonals. Des Weiteren wird durch die Umgebung modelliert, ob sich der Haltepunkt des Zuges an einem Gefälle befindet oder ob ein Passagier oder ein anderer Gegenstand beim Schließen der Türen eingeklemmt wird.

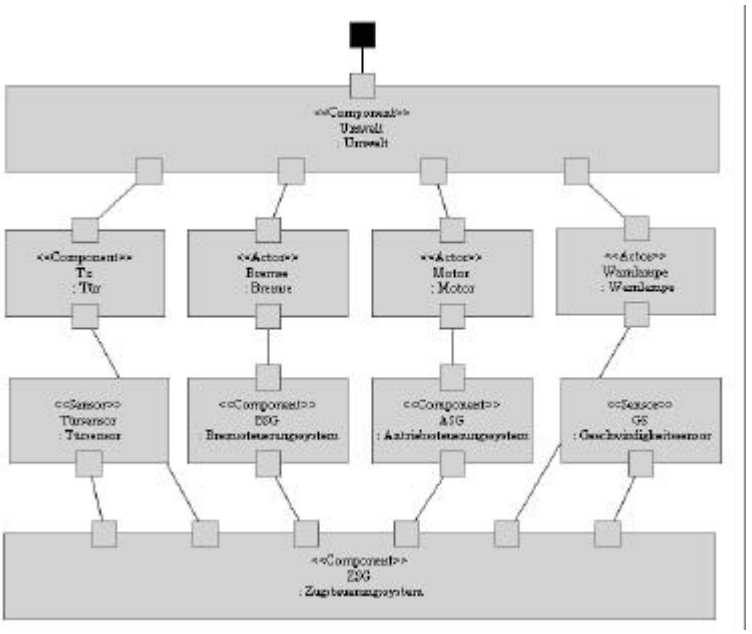


Abbildung 7-10 Strukturspezifikation des Türsteuerungssystems

Bei der Durchführung der Transformationssuche ergaben sich die in Tabelle 7-2 beschriebenen Verbesserungsvorschläge. Dabei eignete sich besonders die redundante Auslegung der Türsensoren, um die Sicherheit des Systems zu verbessern. Bei der Anwendung der entsprechenden Transformationsoperatoren ergaben sich signifikante Qualitätsverbesserungen. Der Grund dafür ist die verbesserte Erkennung von eingeklemmten Gegenständen in der Tür. Die Qualitätsverbesserung sollte jedoch noch durch eine Untersuchung der Fehlverhalten mit gemeinsamer Ursache (*common cause failure*) bestätigt werden, da selbst durch redundante Sensoren kleine Gegenstände, wie z. B. Hundeleinen, nicht erkannt werden.
Neben der redundanten Auslegung der Türsensoren wurde auch die redundante Auslegung des Zugsteuerungssystems zur Verbesserung der Sicherheitseigenschaften vorgeschlagen. Durch die Anwendung der

daraus resultierenden Transformationsoperatoren ergab sich jedoch keine ähnlich signifikante Qualitätsverbesserung.

Tabelle 7-2 Geeignete Transformationsoperatoren zur Verbesserung der Qualität des Türsteuerungssystems

Transformationsoperator	Anwendungskontext	Änderung der Sicherheitseigenschaft (Türen sind geschlossen, wenn der Zug fährt) in %	Änderung der ökonomischen Eigenschaften (Entwicklungskosten)
Homogene Redundanz mit Mehrheitsentscheid (Architekturelement)	TS: Türsensor	+ 40,2%	< +0,01%
Zweikanalige homogene Redundanz	TS: Türsensor	+ 27.2%	< +0,01%
Homogene Redundanz mit Mehrheitsentscheid (Architekturelement)	ZSG: Zugsteuerungssystem	+ 1,2%	+ 9,3%
Zweikanalige homogene Redundanz	ZSG: Zugsteuerungssystem	+ 0,8%	+5,1%

7.2.3 Steuerung des Satelliten BIRD

In der dritten Fallstudie wird der Satellit BIRD (Bi-spectral InfraRed Detection) betrachtet, dessen Aufgabe unter anderem die Branderkennung in unbewohnten Regionen auf der Erdoberfläche ist. Für diese Branderkennung nutzt der Satellit zwei Infrarotkameras, welche die Temperaturvorhersage ermöglichen und somit zusammen ein sicheres Klassifizierungskriterium zur Branderkennung bilden. Wird ein Brand erkannt, so können auf der Erde geeignete Gegenmaßnahmen ergriffen und eine Ausbreitung des Feuers auf bewohnte Regionen verhindert werden.
Da eine Reparatur von Komponenten des Satelliten im Orbit nahezu unmöglich ist, wurde eine Architektur gewählt, welche mehrfache Redundanzen und Recoverykonzepte enthält. Diese Architektur hat sich in der bisherigen Missionszeit bewährt und ist in Abbildung 7-11 dargestellt.
Kern des Satelliten ist die *Satellitensteuerungssoftware.* Diese interpretiert zur Branderkennung die Bilddaten der beiden *Infrarotkameras* und übermittelt die Ergebnisse der Analyse mit Hilfe der *Telemetrieeinheit* an die Bodenstation. Eine weitere Aufgabe der Steuerungssoftware ist die Lagesteuerung. Diese Lagesteuerung ist notwendig, um die Umlaufbahn zu halten und den Satelliten so zu positionieren, dass die *Infrarotkameras* auf die Erdoberfläche ausgerichtet sind. Dazu werden zwei *Sternensensoren*, ein *Gyroscope* und ein *GPS Sensor* genutzt, um die Ist-Position/Lage des Satelliten zu ermitteln. Ausgehend von der berechneten Ist-Lage/Position wird die Abweichung von der Soll-Lage/Position ermittelt. Daraus leitet die Software dann Steuerungsbefehle ab, die den Satelliten mit Hilfe der Aktoren in die Sollposition bringen sollen. Die zu steuernden Aktoren sind vier *ReactionWheels* und zwei *Magnetspulenpaare*. Ausgeführt wird die Steuerungssoftware redundant auf vier *Bordrechnern*, die durch eine zweifach redundante *Stromversorgung* mit Energie versorgt werden. Von diesen vier *Bordrechnern* sind jedoch nur zwei gleichzeitig aktiv. Die anderen beiden dienen als kalte Reserve.
Ziel der Studie war es, die Erfahrungen der Mission zur Verbesserung der Zuverlässigkeit zukünftiger Satelliten zu nutzen. Daher wurde ausgehend von den während der Missionszeit registrierten Ausfällen und Fehlverhalten ein Fehlerbaummodell entwickelt. Dieses Fehlerbaummodell berücksichtigt Fehlerursachen, welche zum Beispiel durch die Bestrahlung des Satelliten mit kosmischen Partikeln entstehen, die diesen Satelliten, ungehindert von einer schützenden Atmosphäre, treffen.
Bei der Optimierung durch Anwendung des SOQA-Prozesses wurden neben der Zuverlässigkeit auch Kosten und Gewicht des Satelliten berücksichtigt. Das Gewicht ist relevant für den Transport des Satelliten in den Orbit und sollte nach den Anforderungen eine Obergrenze von 100 kg nicht überschreiten. Damit ist der Satellit ein sogenannter Kleinsatellit und kann günstig zusammen mit einem großen Satelliten befördert werden. Bei der Konstruktion des Satelliten wurde das Gewicht fast optimal ausgenutzt.

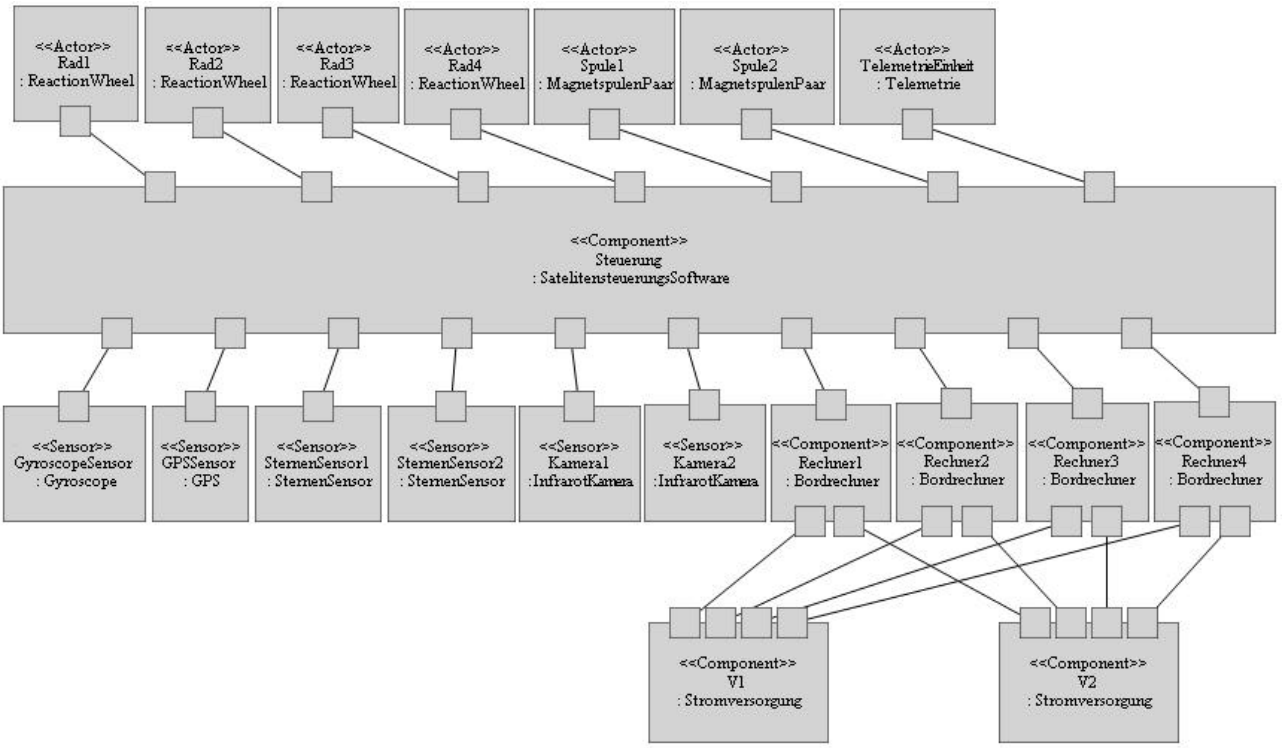


Abbildung 7-11 Strukturspezifikation des Satelliten BIRD

Bei der Transformationssuche wurde die Anwendung der Transformationsoperatoren *Recovery Block* und *Heterogene Redundanz mit Mehrheitsentscheid* auf die Steuerungssoftware als geeignet ermittelt, die Qualität der Architektur zu verbessern. Diese Transformationsoperatoren fügen heterogen implementierte Softwarekomponenten zur Architektur hinzu, welche nach einer identischen Interfacespezifikation, aber von unterschiedlichen Entwicklungsteams erstellt wurden. Diese heterogene Implementierung der Softwarekomponenten wird im BIRD Satelliten bereits realisiert, da über die Telemetrie-Einheit neue Softwareversionen auf die Bordrechner überspielbar sind.
Alle weiteren Transformationsoperatoren fügten stets zusätzliche Hardwareelemente zur Architektur hinzu und erhöhten somit das Gewicht des Satelliten über die vorgeschriebene Gewichtsobergrenze. Daraus resultiert, dass diese Transformationsoperatoren auf Grund der Gewichtsanforderungen nicht anwendbar waren.

7.2.4 Kritische Betrachtung der Fallstudien

In allen drei Fallstudien waren die zugrunde liegenden Software- und Systemarchitekturen gut durchdacht und auf die jeweiligen Anforderungen abgestimmt. Des Weiteren waren alle Systeme bereits im Einsatz, und vor Inbetriebnahme wurden Nachweise erbracht, welche die Erfüllung der jeweiligen Anforderungen belegen. Aus diesem Grund kann anhand der Fallstudien keine Aussage getroffen werden, inwiefern sich der automatisierte SOQA-Prozess bei der Neuentwicklung eines Systems und bei unausgereiften Architekturen eignet. Um diese Aussagen zu ermöglichen, müssten entsprechenden Langzeitstudien, währende der Entwicklung eines softwareintensiven technischen Systems durchgeführt werden.
Die betrachten Fallstudien beziehen sich weiterhin nur auf eine Verbesserung der Qualitätsmerkmale Zuverlässigkeit und Sicherheit. Aus diesem Grund wurde nur ein Teil der vorgeschlagenen Transformationsoperatoren benötigt. Um die Praktikabilität der anderen Transformationsoperatoren nachzuweisen, müssten daher weitere Fallstudien erstellt werden, welche sich auf die entsprechenden Qualitätsmerkmale beziehen.
Zur Durchführung dieser Fallstudien müssten weitere Evaluationsmodelle einbezogen werden. Ausgehend von den in Kapitel 3 vorgestellten Techniken zur Architekturevaluation, erscheinen besonders probabilistische Zustandsautomaten (Markov-Modelle) und Schedulinganalysen (RMA, EDF) geeignet, die Qualität des Werkzeuges BALANCE und des SOQA-Prozesses zu verbessern. Durch die Einbindung von probabilistischen Zustandsautomaten würde auch die Qualität der bisher erstellten Fallstudien verbessert werden, da einige Systemeigenschaften durch Fehlerbäume nur unzureichend spezifiziert werden. Ein Beispiel ist die

kalte Reserve bei den Bordrechnern des Satelliten BIRD. Diese Bordrechner besitzen im ausgeschalteten Zustand eine Ausfallwahrscheinlichkeit, welche von den beiden aktiven Bordrechnern abweicht. Aus diesem Grund wäre ein probabilistisches zustandsbasiertes Modell besser geeignet, die Ausfallwahrscheinlichkeiten in Abhängigkeit des aktuellen Zustandes zu beschreiben.

7.3 Zusammenfassung

Durch das Werkzeug BALANCE und die durchgeführten Fallstudien wurde gezeigt, dass der automatisierte SOQA-Prozess auch praktisch anwendbar ist. Dabei konnten Architekturalternativen mit signifikanten Qualitätsverbesserungen ohne großen zeitlichen und personellen Aufwand gefunden werden. Somit konnten die Annahmen aus Kapitel 4 über die Vorteile der automatisierten und werkzeugunterstützten Prozessanwendung auch praktisch belegt werden.

8 Zusammenfassung und Ausblick

In diesem Kapitel werden die Ergebnisse der vorliegenden wissenschaftlichen Arbeit abschließend zusammengefasst und ihr praktischer Nutzen beschrieben. Darüber hinaus wird ein Ausblick auf Forschungsthemen gegeben, welche sich zur Weiterentwicklung dieser Forschungsarbeit anbieten.

8.1 Ergebnisse

In dieser Dissertation wurde ein automatisierbares Verfahren zur Verbesserung der Qualitätseigenschaften eines softwareintensiven technischen Systems vorgestellt. Dabei werden qualitätsverbessernde verhaltenserhaltende Transformationsoperatoren (semi)-automatisch auf die Architekturspezifikationen angewendet. Dieses Verfahren entspricht der gegebenen Zieldefinition und leistet somit einen Beitrag zur Lösung der in Kapitel 1 beschriebenen Probleme softwareintensiver technischer Systeme.
Für die technische Realisierung und Umsetzung des Verfahrens wird eine neuartige Architekturbeschreibungssprache vorgeschlagen, welche folgende Konzepte integriert:

- Als Spezifikationsformalismus für die Strukturspezifikation einer Architektur werden Hypergraphen vorgeschlagen. Dabei werden insbesondere hierarchische typisierte Hypergraphen genutzt, um die Beschreibung von hierarchisch strukturierten Architekturspezifikationen zu ermöglichen.
- Die flachen Architekturelemente werden in der Architekturbeschreibungssprache mit modularen Evaluationsmodellen für die betrachteten Qualitätseigenschaften annotiert. Mit diesen modularen Evaluationsmodellen und den darauf aufbauenden kompositionsbasierten Techniken ist eine automatische Evaluation von Architekturspezifikationen möglich.
- Die Hardwareelemente eines softwareintensiven technischen Systems werden in die Architekturspezifikation integriert, um eine vollständige Architekturevaluation zu ermöglichen.
- Die flachen Architekturelemente werden in der Architekturbeschreibungssprache mit Interfaceautomaten zur Beschreibung des funktionalen Verhaltens annotiert. Mit diesen Interfaceautomaten und den darauf aufbauenden kompositionsbasierten Techniken kann das Verhalten der gesamten Architektur bestimmt werden.

Ein weiteres Ergebnis dieser Dissertation ist ein Formalismus zur automatischen Anwendung von Architekturtransformationen auf Basis der Architekturbeschreibungssprache. Dieser Formalismus basiert auf der Graphtransformation in der Kategorie der hierarchischen typisierten Hypergraphen. Zur Spezifikation der Architekturtransformationsoperatoren werden Graphtransformationsregeln um die folgenden Konzepte und Verfahren erweitert:

- Die verwendeten Graphtransformationsregeln werden durch die Spezifikation von Anwendungsbedingungen erweitert. Dies ermöglicht die Einschränkung des Anwendungskontextes durch die Spezifikation von Vorbedingungen. Des Weiteren wird eine gezielte Auswahl von geeigneten Regeln auf Basis von Nachbedingungen ermöglicht, welche die Ziele der Regelanwendung spezifizieren. Als Notation der Anwendungsbedingungen werden dabei prädikatenlogische Formeln verwendet.
- Die Graphtransformationsregeln werden durch struktur- und typgenerischen Konzepte erweitert, um die Mächtigkeit und die Praktikabilität der Architekturtransformationen zu erhöhen.
- Zusätzlich wird ein Algorithmus auf Basis der annotierten Interfaceautomaten der Architekturspezifikation vorgeschlagen, welcher die Verhaltensäquivalenz der Architektur vor und nach der Anwendung eines Transformationsoperators prüft. Somit kann der Nachweis erbracht werden, dass durch die Anwendung eines Architekturtransformationsoperator die funktionalen Eigenschaften nicht verändert werden.

Ausgehend von dem vorgestellten Formalismus wurden elf Transformationsoperatoren vorgeschlagen, welche sich zur Anwendung des im Kapitel 4 beschriebenen SOQA-Prozesses eignen.

8.2 Praktischer Nutzen

Durch den beschriebenen Automatisierungsansatz und das entwickelte Werkzeug BALANCE ist das vorgestellte Verfahren auch in industriellen Projekten einsetzbar. Dies wurde unter anderem durch industrielle Fallstudien aus dem Bereich des Personennahverkehrs, der Reaktorsicherheit und der Luft- und Raumfahrt nachgewiesen. Bei der Durchführung dieser Fallstudien ergaben sich ökonomische Vorteile, welche durch die Reduktion des zeitlichen Aufwandes gegenüber einer nicht automatisierten Qualitätsoptimierung begründet sind.
Des Weiteren wird durch die spezifizierten Transformationsoperatoren Expertenwissen zur Verfügung gestellt, welches speziell unerfahrenen Softwarearchitekten die Entwicklung einer qualitativ hochwertigen Architekturspezifikation erleichtert. Dieses Expertenwissen kann durch die Spezifikation von weiteren Transformationsoperatoren noch erweitert werden. Für Unternehmen und Branchen kann somit ein einheitlicher Satz von Transformationsoperatoren erstellt werden. Dieser Transformationsoperatorsatz kann in jedem Entwicklungsprojekt angewendet werden und zur Verbesserung der Qualität der erstellten softwareintensiven technischen Systeme beitragen.

8.3 Ausblick

Bei der Erstellung dieser wissenschaftlichen Arbeit wurden einige offene Forschungsthemen identifiziert, welche in weiteren Forschungstätigkeiten betrachtet werden sollten. Diese offenen Forschungsfragen werden im Folgenden beschrieben und kurz diskutiert.

Behandlung von weiteren Qualitätsmerkmalen
Die Architekturspezifikation und die beschriebenen Transformationsoperatoren sind im Speziellen für die Spezifikation und Verbesserung der Qualitätsmerkmale Sicherheit, Verfügbarkeit, Zuverlässigkeit, Wartbarkeit und Echtzeitfähigkeit geeignet. Für eine Verbesserung der Anwendbarkeit des Verfahrens sollten jedoch auch weitere Qualitätsmerkmale berücksichtigt werden. Im Besonderen eignet sich die Sicherheit, im Sinne des englischen Begriffes *security*, für eine mögliche Erweiterung. Aber auch Qualitätseigenschaften wie zum Beispiel Robustheit, Effektivität, Testbarkeit, Benutzerfreundlichkeit, Erweiterbarkeit, Portabilität und Wiederverwendbarkeit sollten in Betracht gezogen werden.
Für die Erweitung des Verfahrens um neue Qualitätsmerkmale sind vor allem neue Evaluationsmodelle für die Bestimmung der Qualitätseigenschaften zu spezifizieren. Diese neuen Evaluationsmodelle sollten in die Architekturbeschreibungssprache und in das Werkzeug BALANCE eingebunden werden, um den praktischen Nutzen zu erweitern.

Erstellung von weiteren Transformationsoperatoren
Durch die Behandlung von zusätzlichen Qualitätsmerkmalen werden auch neue Architekturtransformationsoperatoren benötigt, um die entsprechenden Qualitätseigenschaften zu verbessern.
Aber auch die bisher spezifizierte Sammlung an Architekturtransformationsoperatoren für die Qualitätsmerkmale Sicherheit, Verfügbarkeit, Zuverlässigkeit, Wartbarkeit und Echtzeitfähigkeit besitzt keinen Anspruch auf Vollständigkeit. Daher ist eine Erweiterung des Transformationsoperatorsatzes auch für diese Qualitätsmerkmale denkbar. Besonderes Interesse sollte dabei Architekturtransformationsoperatoren gelten, welche sich in einer speziellen Anwendungsdomäne bewährt haben. Dadurch kann ein Satz von effektiven Architekturtransformationsoperatoren für diesen Anwendungsbereich erstellt werden.

Anwendung der erstellten Konzepte auf andere Spezifikations - und Modellierungssprachen
Der in dieser wissenschaftlichen Arbeit beschriebene Ansatz fokussiert die Anwendung auf Architekturspezifikationen in der Architekturbeschreibungssprache COOL. Für eine Erweiterung ist es jedoch auch denkbar, das Konzept der qualitätsverbessernden Architekturtransformationen für andere Spezifikations- und Modellierungssprachen zu realisieren. Besonders geeignet scheint dazu UML RT und UML 2.0, da diese Spezifikations- und Modellierungssprachen einen ähnlichen Aufbau der Strukturspezifikation besitzen, wie die ADL COOL /Grunske 04/.

Architekturtransformation zur Laufzeit des Systems
Eine weitere denkbare Erweiterung ist die Anwendung von Architekturtransformationen zur Systemlaufzeit. Wenn zum Beispiel ein System erkennt, dass ein Server ausgelastet ist, kann der Architekturtransformationsoperator „Neuzuweisung der Hardwareplattform“ angewendet werden. Dadurch werden die Softwarekomponenten auf andere Server verteilt. Somit wird in einem System eine optimale Lastenverteilung erreicht.

Architekturverfeinerungen
In dieser Arbeit wurden speziell horizontale Architekturtransformationen behandelt, welche es ermöglichen, eine Architektur unter Beibehaltung der funktionalen Eigenschaften zu verändern. Die vorgestellten Grundlagen eignen sich jedoch auch für die Neuentwicklung von Architekturen. Dazu ist es erforderlich, Architekturverfeinerung zu definieren, welche eine Hyperkante des Typs *Modul* durch feingranularere Architekturen und Architekturelemente ersetzt. Diese Architekturverfeinerungen lassen sich somit formal als Hyperkantenersetzungsregeln spezifizieren.
Bei der Architekturverfeinerung ergibt sich aber das Problem des Nachweises der funktionalen Korrektheit. Oft wird eine Architektur mit dem Ziel verfeinert, Funktionalität hinzuzufügen. Dazu müsste der in dieser Dissertation beschriebene Ansatz überdacht werden, da er nur verhaltenserhaltene Architekturtransformationen zulässt.

Architekturtransformation im Rahmen des MDA Ansatzes
Ein weiterer Ansatzpunkt für eine weitergehende Forschung ist die Architekturtransformation zwischen verschiedenen Modelltypen und die Codegenerierung aus der Architektur im Rahmen des *MDA*-Projektes (*model driven architecture*) der OMG /OMG 03/.
Die Architekturtransformation zwischen verschiedenen Modelltypen wird als horizontale Architekturtransformation bezeichnet und ist für die Spezifikation von verschiednen Sichten auf die Architektur erforderlich /Agrawal et al. 03/. Dabei werden Architekturtransformationen dazu verwendet die Sichten untereinander konsistent zu halten. Problematisch dabei ist die Spezifikation von Graphtransformationen zwischen Sichten mit unterschiedlichen Metamodellen. Die in dieser Dissertation beschriebenen Transformationsoperatoren sind nur geeignet, Architekturtransformationen innerhalb des COOL-Metamodelles zu spezifizieren.
Die Codegenerierung aus der Architektur entspricht einer vertikalen Architekturtransformation /Agrawal et al. 03/. Dabei ist es erforderlich, die visuelle Sprache der Architektur zu parsen und in die entsprechenden Bestandteile einer textbasierten, kompilierbaren Spezifikation umzusetzen. Dies erfordert eine Erweiterung der in dieser Arbeit vorgestellten Graphtransformationsregeln und kann daher ebenfalls ein Ansatzpunkt für die weitere Forschung sein.

Anhang A Mathematische Grundlagen

Im Folgenden werden ausgewählte mathematische Konzepte und Notationen vorgestellt, welche im Rahmen der Dissertationsschrift verwendet wurden.

A.1 Mengen

Die Grundlage für die meisten in dieser Dissertation verwendeten mathematischen Konzepte bilden Mengen. Definiert wird eine Menge als eine Zusammenfassung von wohlunterschiedenen Objekten. Des Weiteren werden die folgenden Notationen verwendet.

- $|A|$ bezeichnet die Anzahl der Elemente einer *endlichen* Menge A
- $\subseteq$ bezeichnet die Teilmengenbeziehung zwischen zwei Mengen, während $\subset$ die strikte Teilmengenbeziehung bezeichnet
- $A \times B$ ist das kartesische Produkt zweier Mengen A und B. Es definiert Tupel aus Elementen beider Mengen. Wenn $a \in A$ und $b \in B$, dann ist $(a,b) \in A \times B$.
- Die Menge aller Teilmengen einer Menge A wird Potenzmenge $\mathrm{P}(A)$ genannt. $\mathrm{P}(A) = \{B \mid B \subseteq A\}$

A.2 Relationen

Eine Relation $R \subseteq A \times B$ ist eine Teilmenge eines kartesischen Produktes von zwei endlichen Mengen A und B. Für $(a,b) \in R$ kann man auch $a\,R\,b$ schreiben. Eine Relation $R \subseteq A \times B$ heißt binär, wenn $A = B$. Die inverse Relation $R^{-1} \subseteq B \times A$ ist formal wie folgt definiert: $R^{-1} = \{(b,a) \in B \times A \mid (a,b) \in R\}$.
In Analogie zu den noch beschriebenen Funktionen schreibt man anstatt $R \subseteq A \times B$ auch $R : A \to B$. Wichtig ist dann aber, dass $R(a)$ die Menge aller $b \in B$ bezeichnet, die mit a in Relation R stehen. Formal: $R(a) = \{b \in B \mid (a,b) \in R\}$.
Relationen können miteinander komponiert werden. Die Komposition von Relationen $R : A \to B$ und $S : B \to C$ ist definiert als $S \circ R = \{(a,c) \mid \exists b \in B \; so \; dass \; (a,b) \in R \; und \; (b,c) \in S\}$.
Ein binäre Relation, die in der Praxis nützlich ist, da sie es erlaubt, Elemente miteinander zu vergleichen, ist die Halbordnung. Eine binäre Relation $R \subseteq A \times A$ ist eine Halbordnung, g.d.w.

- R ist reflexiv: $\forall a \in A : a\,R\,a$
- R ist antisymmetrisch: $\forall a,b \in A : if \; aRb \; and \; bRa \; then \; a = b$
- R ist transitiv: $\forall a,b,c \in A : if \; aRb \; and \; bRc \; then \; aRc$

A.3 Funktionen

Funktionen sind spezielle Relationen $R \subseteq A \times B$, bei denen jedes Element aus der Menge A maximal einem Element aus der Menge B zugeordnet ist.
Eine Funktion $f : A \to B$ muss nicht notwendigerweise für alle $a \in A$ definiert sein. Der Funktionsbereich $dom(f)$ bezeichnet die Menge aller $a \in A$, für die f definiert ist. Wenn $dom(f) \subset A$, dann wird f als partielle Funktion bezeichnet. Der Bildbereich einer $ran(f)$ einer Funktion $f : A \to B$ bezeichnet die Menge aller $b \in B$, auf die f ein $a \in A$ abbildet. Formal: $ran(f) = \{b \in B \mid \exists a \in A \, so \, dass \; f(a) = b\}$
Man nennt eine Funktion $f : A \to B$ injektiv, wenn man von jedem Bild $b \in ran(f)$ auf das $a \in A$ schließen kann, für das $f(a) = b$ gilt. Formal: $\forall a,b \in dom(f) : (f(a) = f(b)) \Rightarrow a = b$.
Wenn der Bildbereich der Funktion gleich der Menge ist, auf die abgebildet wird, wird eine Funktion $f : A \to B$ als surjektiv bezeichnet. Formal: $dom(f) = B$. Ist eine Funktion sowohl injektiv als auch surjektiv, dann ist sie bijektiv.

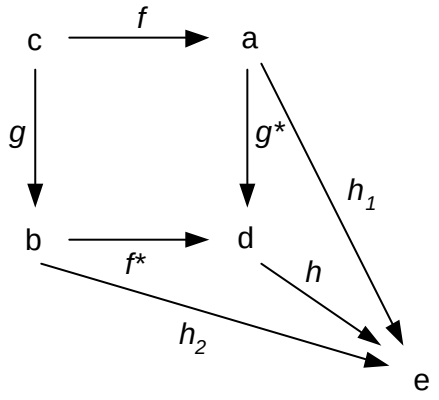
A.4 Grundlagen der Kategorientheorie

Die Kategorientheorie erlaubt es, die abstrakten, strukturelle Gemeinsamkeiten zwischen konkreten mathematischen Strukturen wie Mengen, Funktionen, Relationen, Graphen, etc. zu formalisieren. Eine Kategorie umfasst Objekte, die Abstraktionen von mathematischen Strukturen darstellen, und Morphismen, die die Beziehungen zwischen Objekten ausdrücken.

Formal:
Eine Kategorie C besteht aus Objekten Obj_C und Morphismen $Morph_C$. Jeder Morphismus $f \in Morph_C$ setzt ein Objekt $o_1 \in Obj_C$ mit einem Objekt $o_2 \in Obj_C$ in Beziehung. Dies wird auch $o_1 \to o_2$ geschrieben. Die folgenden vier Axiome gelten stets:
– $\forall a \in Obj_C : \exists$ Identitätsmorphimus $id_a : a \to a$
– Wenn $f : a \to b$ und $g : b \to c$ mit $f, g \in Morph_C$, dann existiert $g \circ f : a \to c$ mit $g \circ f \in Morhp_C$.
– $\forall f : a \to b \in Morhp_C : f \circ id_a = id_b \circ f = f$
– $\forall f : a \to b, g : b \to c, h : c \to d \in Morhp_C : \quad (h \circ g) \circ f = h \circ (g \circ f)$
Sei C eine beliebige Kategorie, dann wird $f : a \to b \in Morph_C$ Isomorphismus genannt, g.d.w. ein $g : b \to a \in Morph_C$ existiert mit $f \circ g = id_a$ und $g \circ f = id_b$.

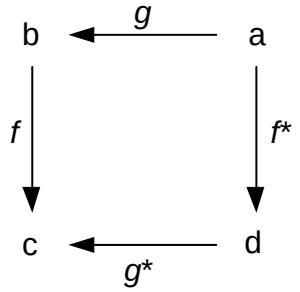
A.4.1 Pushout

Mit Hilfe der Kategorientheorie kann man Strukturen von Strukturen beschreiben. Pushouts beschreiben, wie zwei Objekte miteinander verschmolzen werden können. Formal:
Gegeben sei eine Kategorie C.
Der Pushout von zwei Morphismen $f, g \in Morph_C$, mit $f : c \to a$ und $g : c \to b$, ist das Tripel $(d, f^* : b \to d, g^* : a \to d)$, für das gilt:
(1) $d \in Obj_C$, $f^*, g^* \in Morph_C$
(2) Kommunitativität: $g^* \circ f = f^* \circ g$
(3) Universalität: $\forall e \in Obj_C : \forall h_1, h_2 \in Morph_C$ mit $h_1 : a \to e$, $h_2 : b \to e$ und $h_1 \circ f = h_2 \circ g$: $\exists! h \in Morph_C$ mit $h : d \to e$, so dass $h \circ g^* = h_1$ und $h \circ f^* = h_2$
Die Bedingung (2) stellt sicher, dass ein *pushout*-Objekt existiert. Die Bedingung (3) stellt sicher, dass dieses *pushout*-Objekt minimal ist.



A.4.2 Pushout Complement

Gegeben sei eine Kategorie C.
Das Pushout-Complement von zwei Morphismen $f, g \in Morph_C$, mit $f : b \to c$ und $g : a \to b$, ist das Tripel $(d, f^* : a \to d, g^* : d \to c)$, wenn das Tripel $(c, f : b \to c, g^* : d \to c)$ ein Pushout der Morphismen $f^* : a \to d$ und $g : a \to b$ ist.



Anhang B Spezifikation der entwickelten Algorithmen

B.1 Erstellung einer Interfacespezifikation für ein hierarchisches Architekturelement

Sind für ein hierarchisches Architekturelement in der Architekturbeschreibungssprache COOL die Interfaceautomaten der enthaltenen Komponenten bekannt, so ist mit dem folgenden Algorithmus der Interfaceautomat für das hierarchische Architekturelement konstruierbar. Für die Anwendung des Algorithmus ist jedoch erforderlich, dass ein Ereignistyp über genau ein Interfaceelement mit der Umgebung ausgetauscht wird.

Algorithmus erstelleInterfaceAutomat(k)

Input: Komponente k, für die die Verhaltensspezifikation erzeugt werden soll
Output: Verhaltensspezifikation der Komponente k

```
anzahlSubKomponenten = k.subKomponenten.length;

if ( anzahlSubKomponenten == 0 ) then
        return k.InterfaceAutomat;

InterfaceAutomat result = erstelleInterfaceAutomat ( k.subKomponenten[ 0 ] );
for (k1 = 1; k1 < anzahlSubKomponenten; ++k1 )
        List ereignisMenge;

        /* Bestimme die Menge der Ereignisse, die Komponente k1 mit allen bereits
           komponierten Komponenten austauscht */
        for (k2=0; k2 < k1; ++k2 )
                for each v in k.Verschaltungen
                        if ( v.verschaltet( k1, k2 ) ) then
                                ereignisMenge.add( v.gemeinsameEreignisse() );

        result = komponiere( result,
                        erstelleInterfaceAutomat( k.subKomponenten[ k1 ] ),
                        ereignisMenge );

return result
```

B.2 Erstellung eines Komponentenfehlerbaumes für ein hierarchisches Architekturelement

Sind für ein hierarchisches Architekturelement in der Architekturbeschreibungssprache COOL die Fehlerbäume der enthaltenen Komponenten bekannt, so ist mit dem folgenden Algorithmus der Fehlerbaum für das hierarchische Architekturelement konstruierbar. Sind die Fehlerbäume nicht bekannt, so erfolgt die Konstruktion rekursiv. Somit ist für eine Architektur, in der alle flachen Architekturelemente mit Fehlerbäumen annotiert sind, ein vollständiger Fehlerbaum konstruierbar.

Algorithmus erzeugeFehlerbaum(k)

Input: k Komponente, für die ein Fehlerbaum erzeugt werden soll
Output: Fehlerbaum der Komponente k

```
if ( k.subKomponenten.count == 0 ) then
        return k.FehlerBaum;

f = leererFehlerBaum();
for each s in k.subKomponenten
        f.add( erzeugeFehlerbaum( s ) )

// Iteriere über alle Bindungen und verbinde die entsprechenden Ports miteinander
// sofern Fehler für gemeinsame Aktionen existieren
for each b in k.Bindungen
        fk = f.findKomponentenFehlerbaum( b );
        gemeinsameAktionen = b.getAktionen();
        for each a in gemeinsameAktionen
                if ( fk.hasFehler ( a ) ) then
                        if ( b.istAusgabe( a ) ) then
                                e = f.addEingangsPort( a );
                                f.verbinde( e, fk.getEingangFürAktion( a ) );
                        else
                                e = f.addAusgangsPort( a );
                                f.verbinde( fk.getAusgangFürAktion( a ), e );


// Iteriere über die Verschaltungen aller Subkomponenten und verbinde Ports, für die
// Fehler für gemeinsame Aktionen existieren
for each v in k.Verschaltungen
        gemeinsameAktionen = v.getAktionen();
        for each a in gemeinsameAktionen
                ft1 = f.findKomponentenFehlerbaum1( v, a );
                ft2 = f.findKomponentenFehlerbaum2( v, a );
                if ( ft2.hasFehler( a ) ) then
                        f.verbinde( ft1.getAusgangFürAktion( a ),
                        ft2.getEingangFürAktion( a ) );
                else if ( ft1.hasFehler( a ) ) then
                        f.verbinde( ft2.getAusgangFürAktion( a),
                        ft1.getEingangFürAktion( a ) );
```

B.3 Generierung der Interfacespezifikation des Architekturelementes Voter/Replicator

Der folgenden Algorithmus erstellt einen Interaktionsbaum welcher alle möglichen und zulässigen Ereignissequenzen eines zu erstellenden n/m-Voters beschreibt. Aus diesem Interaktionsbaum kann der Interfaceautomat wie in Abschnitt 6.2.4.3 dargestellt generiert werden:

Algorithmus ErzeugeKnoten(*n, m, s*)

Input: Anzahl n identischer Antworten homogener Komponenten, Gesamtanzahl m der homogenen Komponenten, Ereignismenge s des zu erzeugenden Knoten,

Output: Knoten, der für jede alternativen Zustandsübergang Kindknoten aufweist, denen die aus dem jeweiligen Zustandsübergang resultierenden Ereignismenge zugeordnet ist. Für die Kindknoten gilt rekursiv das gleiche.

Erstelle neuen leeren Knoten *k*

Ergänze Ereignismenge *s* zum Knoten *k*

If Anzahl der vorliegenden Ausgaben gleich *n* ist **then** *// Rekursionsende*

return *k*;

If (*e* = Anzahl($a \in s \mid a$ ist Eingabe)) < m **then** *// Erzeuge ggf. eine weitere Eingabe*

k_1 = ErzeugeKnoten($n, m, s \cup \{ y_{e+1}? \}$);

Verbinde k_1 mit *k* und beschrifte die Kante mit y_{e+1}?

For each $y_i? \in \{ a_i \in s \mid a_i$ ist Eingabe und $x_i! \notin s \}$ *// Erzeuge Nachfolgeknoten für Ausgaben*

k_i = ErzeugeKnoten($n, m, s \cup \{ x_i! \}$);

Verbinde k_i mit *k* und beschrifte die Kante mit x_i!

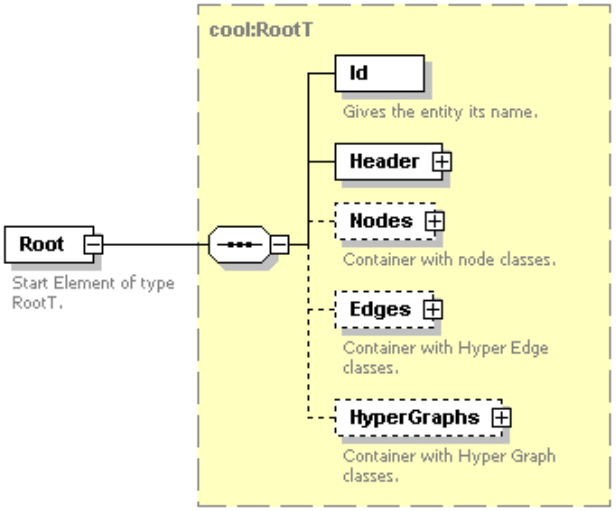
return *k;*

Anhang C Das XML-Schema der Architekturbeschreibungssprache COOL

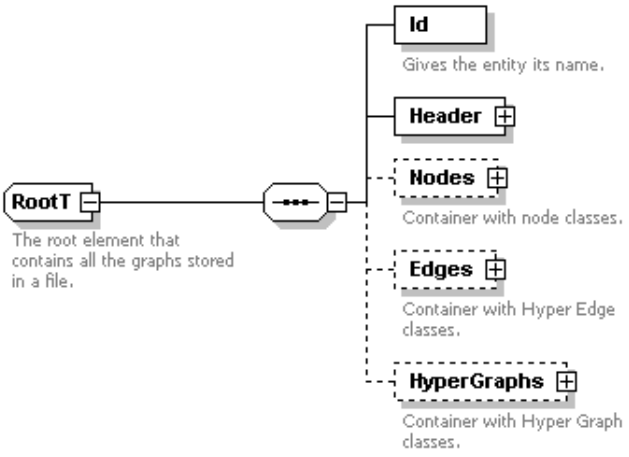
Im Folgenden wird eine Beschreibung des COOL XML Schemas vorgestellt. Dieses Schema beschreibt das Format einer Architekturspezifikation und wird im Werkzeug BALANCE verwendet.

C.1 Cool.xsd

Element Root

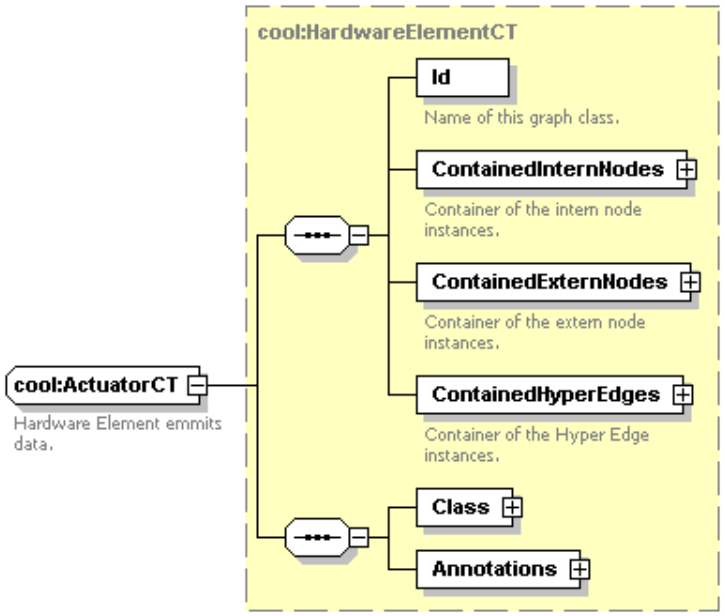
diagram	
type	cool:RootT
children	Id Header Nodes Edges HyperGraphs

complexType RootT

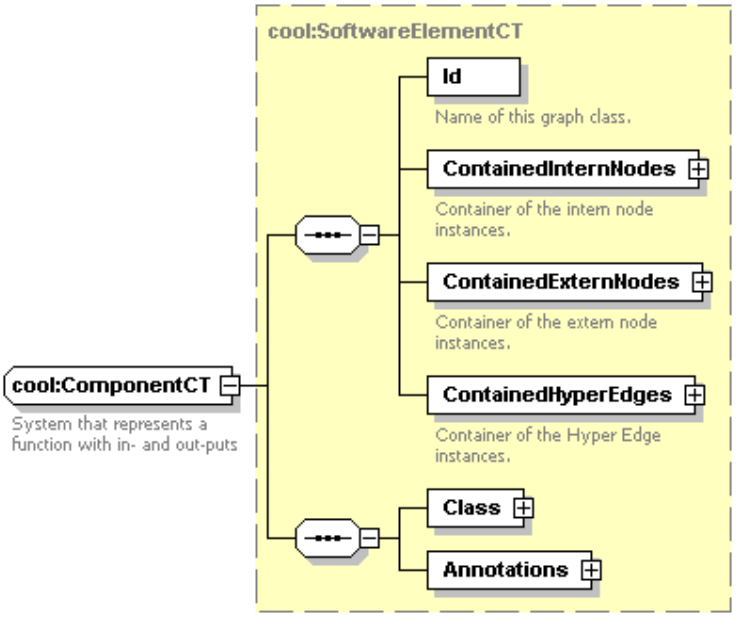
diagram	
type	restriction of cool:ContainerT
children	Id Header Nodes Edges HyperGraphs

C.2 Abstract_cool_types.xsd

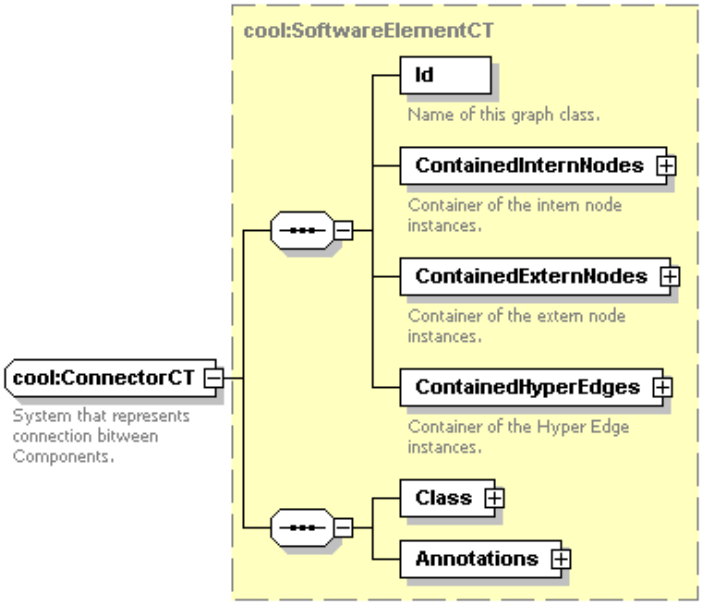
complexType cool:ActuatorCT

diagram	
type	extension of cool:HardwareElementCT
children	Id ContainedInternNodes ContainedExternNodes ContainedHyperEdges Class

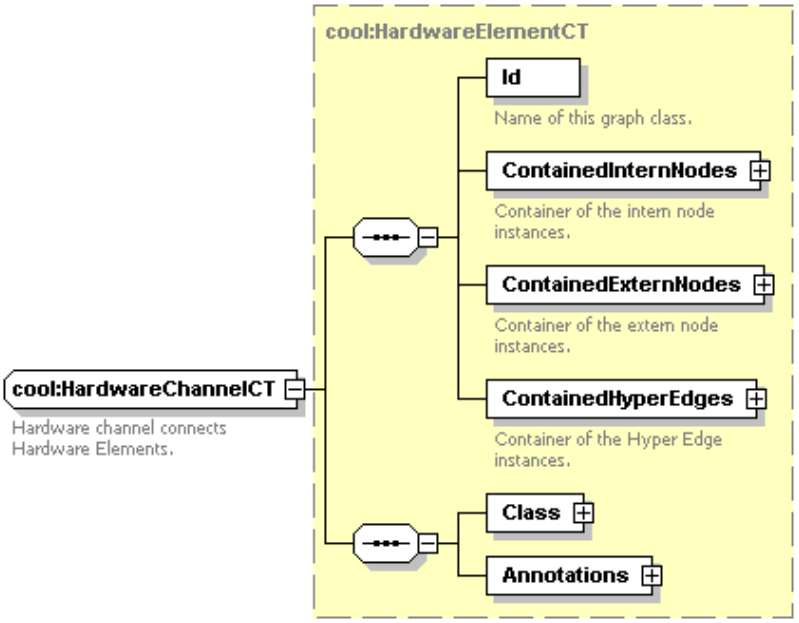
complexType cool:ComponentCT

diagram	
type	extension of cool:SoftwareElementCT
children	Id ContainedInternNodes ContainedExternNodes ContainedHyperEdges Class

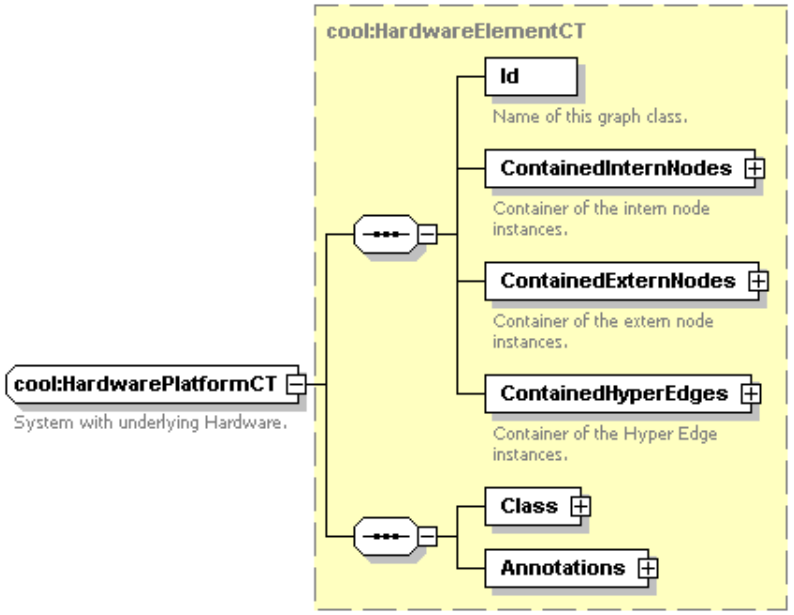
complexType cool:ConnectorCT

diagram	
type	extension of cool:SoftwareElementCT
children	Id ContainedInternNodes ContainedExternNodes ContainedHyperEdges Class

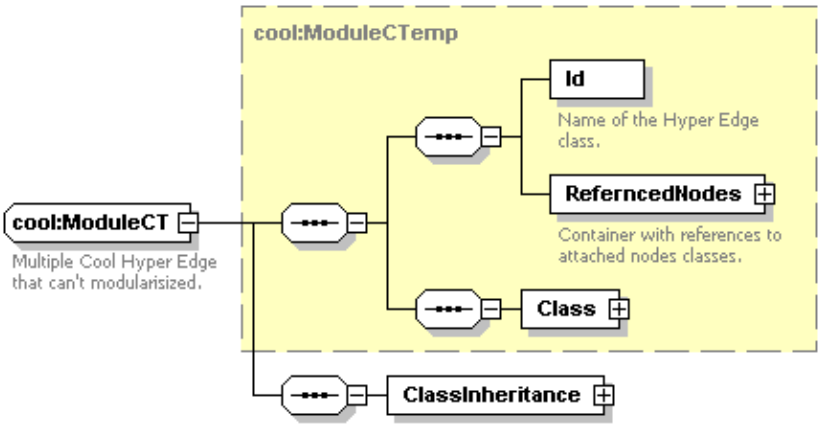
complexType cool:HardwareChannelCT

diagram	
type	extension of cool:HardwareElementCT
children	Id ContainedInternNodes ContainedExternNodes ContainedHyperEdges Class

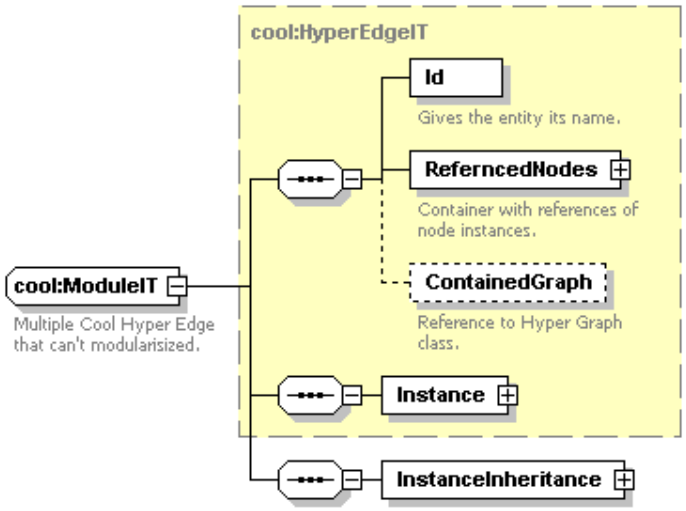
complexType cool:HardwarePlatformCT

diagram	
type	extension of cool:HardwareElementCT
children	Id ContainedInternNodes ContainedExternNodes ContainedHyperEdges Class

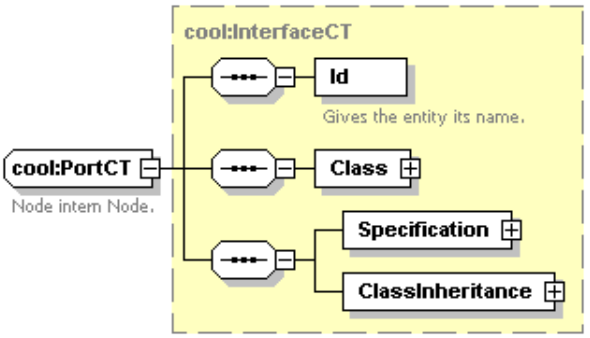
complexType cool:ModuleCT

diagram	
type	extension of cool:ModuleCTemp
children	Id ReferncedNodes Class ClassInheritance

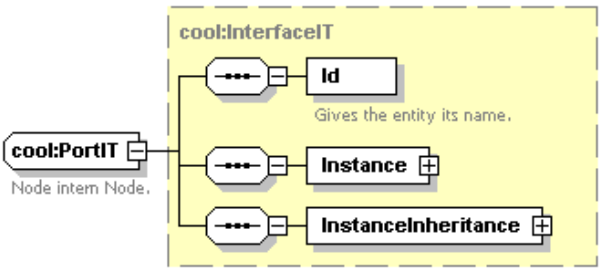
complexType cool:ModuleIT

diagram	
type	extension of cool:HyperEdgeIT
children	Id ReferncedNodes ContainedGraph Instance InstanceInheritance

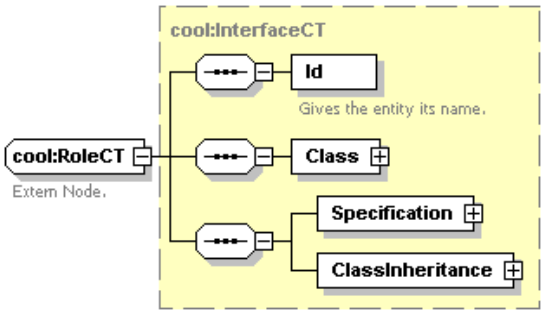
complexType cool:PortCT

diagram	
type	extension of cool:InterfaceCT
children	Id Class Specification ClassInheritance

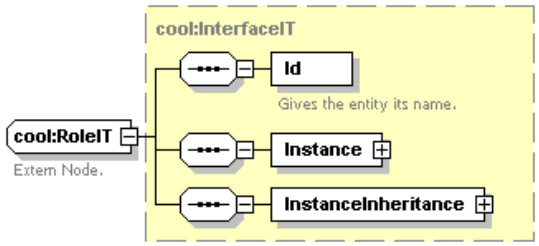
complexType cool:PortIT

diagram	
type	extension of cool:InterfaceIT
children	Id Instance InstanceInheritance

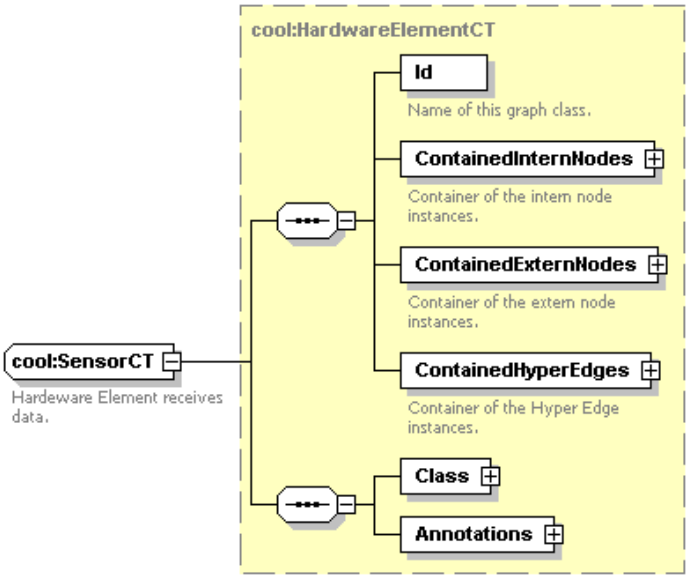
complexType cool:RoleCT

diagram	
type	extension of cool:InterfaceCT
children	Id Class Specification ClassInheritance

complexType cool:RoleIT

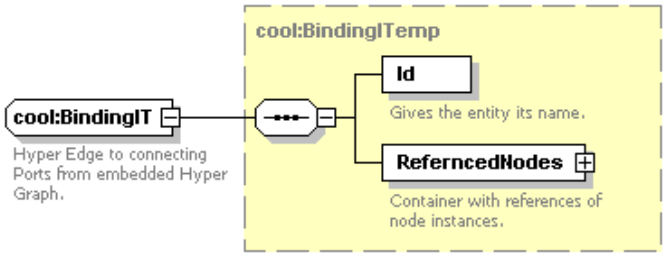
diagram	
type	extension of cool:InterfaceIT
children	Id Instance InstanceInheritance

complexType cool:SensorCT

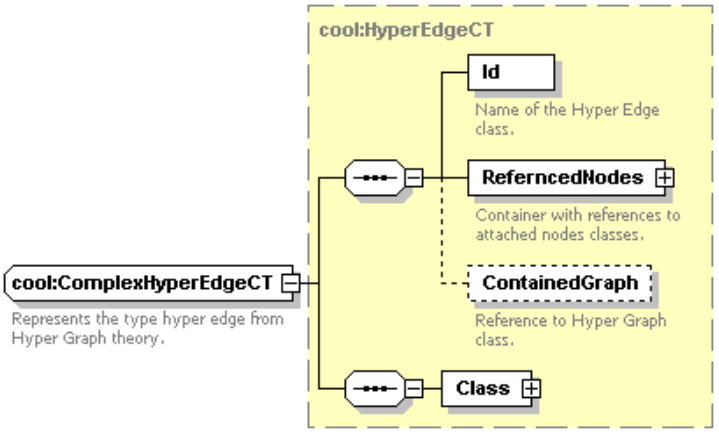
diagram	
type	extension of cool:HardwareElementCT
children	Id ContainedInternNodes ContainedExternNodes ContainedHyperEdges Class

C.3 Usable_cool_types.xsd

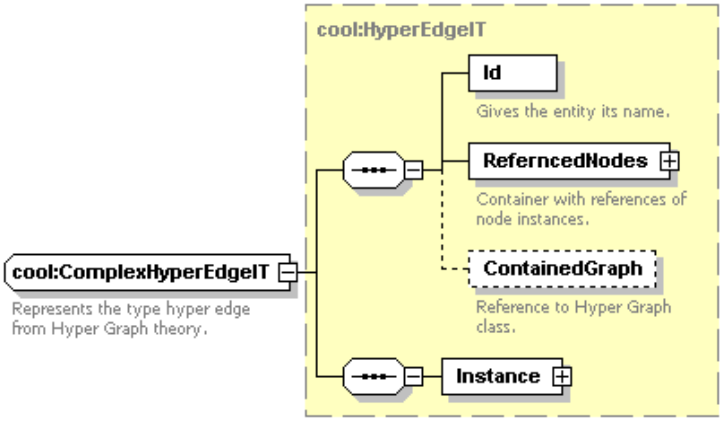
complexType cool:BindingIT

diagram	
type	extension of cool:BindingITemp
children	Id ReferncedNodes

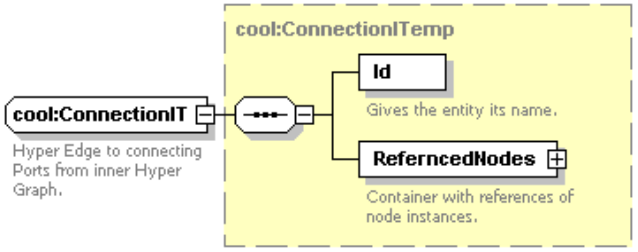
complexType cool:ComplexHyperEdgeCT

diagram	
type	extension of cool:HyperEdgeCT
children	Id ReferncedNodes ContainedGraph Class

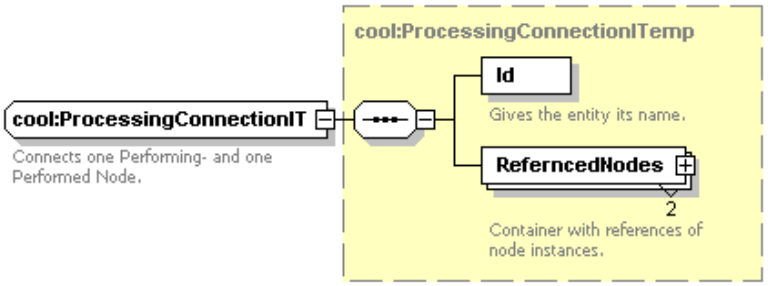
complexType cool:Comple xHyperEdgeIT

diagram	
type	extension of cool:HyperEdgeIT
children	Id ReferncedNodes ContainedGraph Instance

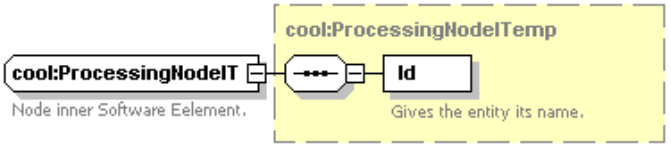
complexType cool:ConnectionIT

diagram	
type	extension of cool:ConnectionITemp
children	Id ReferncedNodes

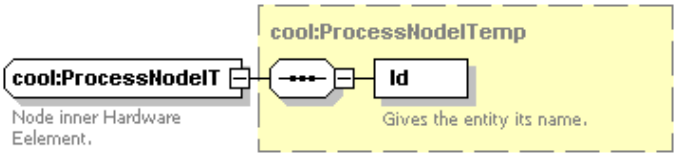
complexType cool:ProcessingConnectionIT

diagram	
type	extension of cool:ProcessingConnectionITemp
children	Id ReferncedNodes

complexType cool:ProcessingNodeIT

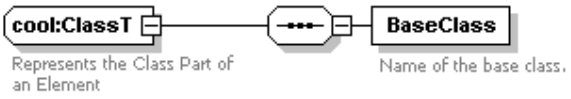
diagram	
type	extension of cool:ProcessingNodeITemp
children	Id

complexType cool:ProcessNodeIT

diagram	
Type	extension of cool:ProcessNodeITemp
children	Id

C.4 Hyper_graph_classes_and_instances.xsd

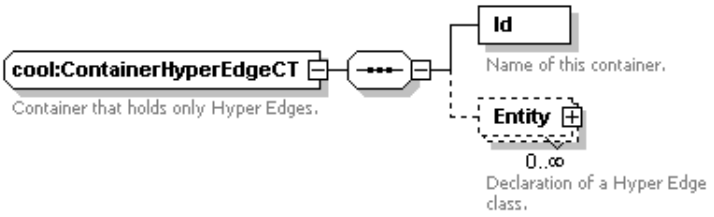
complexType cool:ClassT

diagram	
children	BaseClass

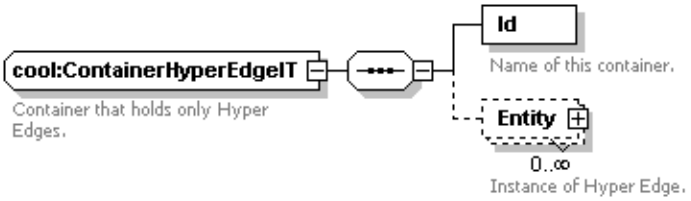
attributes

Name	Type	Use	Default	Fixed
Name	xs:string	required		
IsSubstitutableByInheritor	xs:boolean	required		

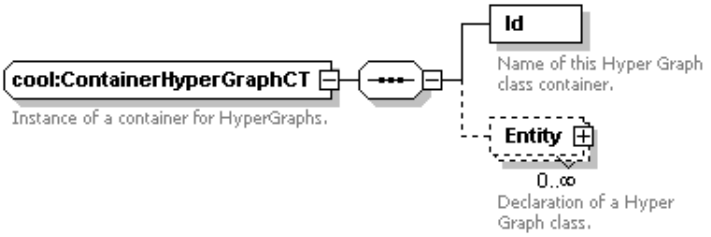
complexType cool:ContainerHyperEdgeCT

diagram	
type	restriction of cool:ContainerHyperEdgeT
children	Id Entity

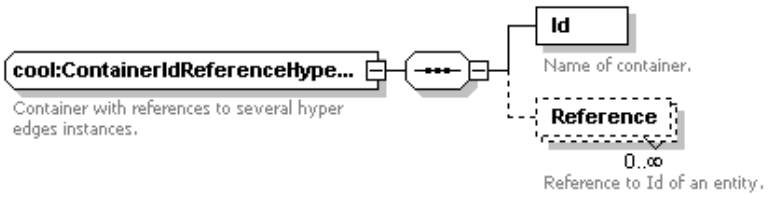
complexType cool:ContainerHyperEdgeIT

diagram	
type	restriction of cool:ContainerHyperGraphT
children	Id Entity

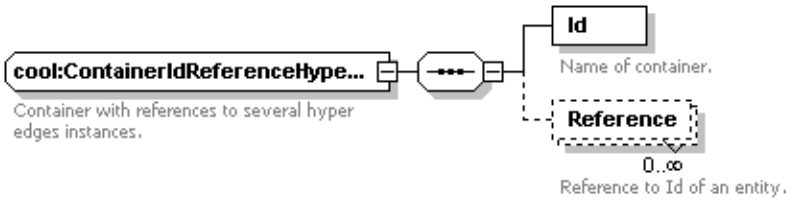
complexType cool:ContainerHyperGraphCT

diagram	
type	restriction of cool:ContainerHyperGraphT
children	Id Entity

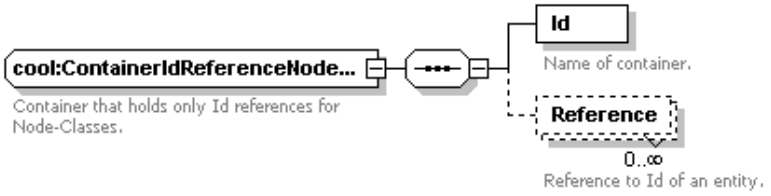
complexType cool:ContainerIdReferenceHyperEdgeClassT

diagram	
type	restriction of cool:ContainerIdReferenceHyperEdgeT
children	Id Reference

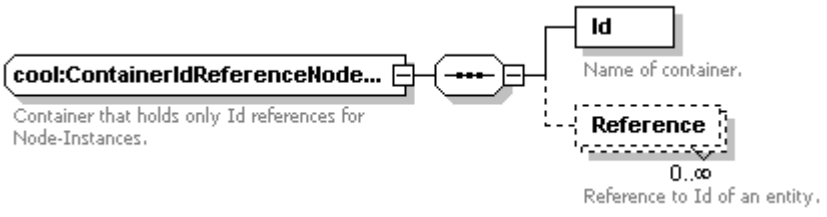
complexType cool:ContainerIdReferenceHyperEdgeInstancesT

diagram	
type	restriction of cool:ContainerIdReferenceHyperEdgeT
children	Id Reference

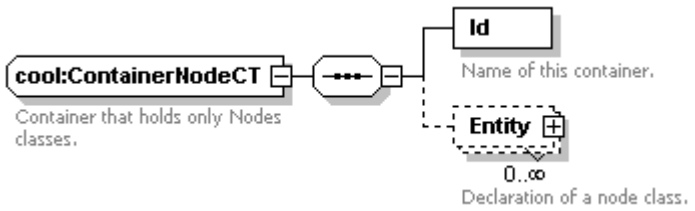
complexType cool:ContainerIdReferenceNodeClassT

diagram	
type	restriction of cool:ContainerIdReferenceNodeT
children	Id Reference

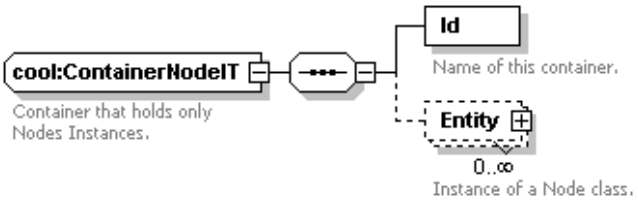
complexType cool:ContainerIdReferenceNodeInstanceT

diagram	
type	restriction of cool:ContainerIdReferenceNodeT
children	Id Reference

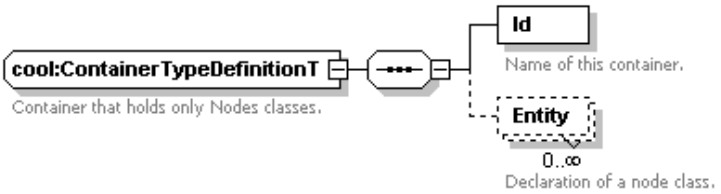
complexType cool:ContainerNodeCT

diagram	
type	restriction of cool:ContainerNodeT
children	Id Entity

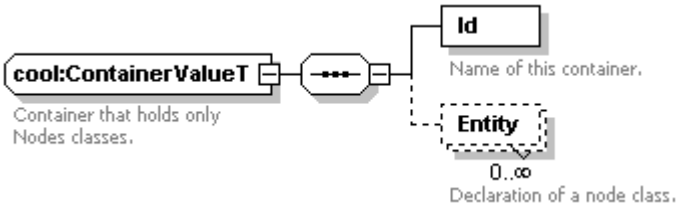
complexType cool:ContainerNodeIT

diagram	
type	restriction of cool:ContainerNodeT
children	Id Entity

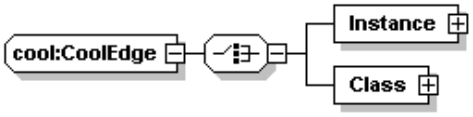
complexType cool:ContainerTypeDefinitionT

diagram	
type	restriction of cool:ContainerNodeT
children	Id Entity

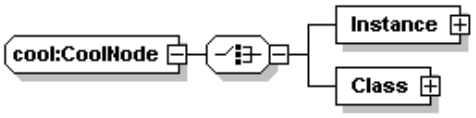
complexType cool:ContainerValueT

diagram	
type	restriction of cool:ContainerNodeT
children	Id Entity

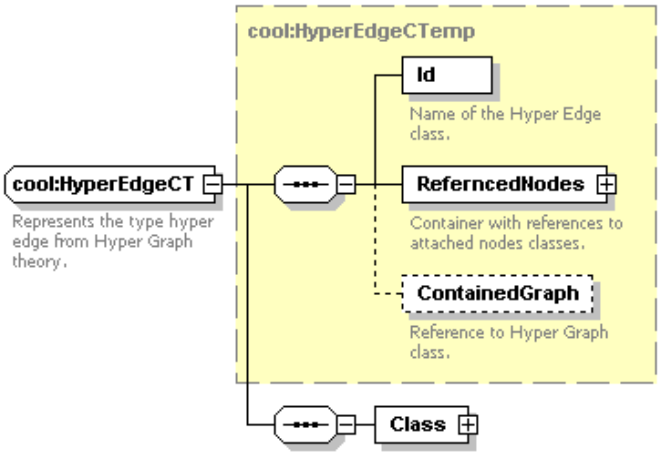
complexType cool:CoolEdge

diagram	
children	Instance Class

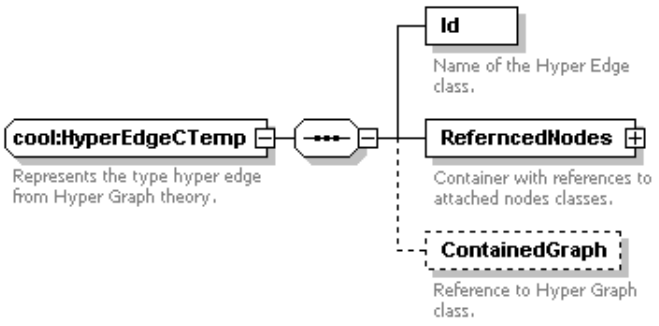
complexType cool:CoolNode

diagram	
children	Instance Class

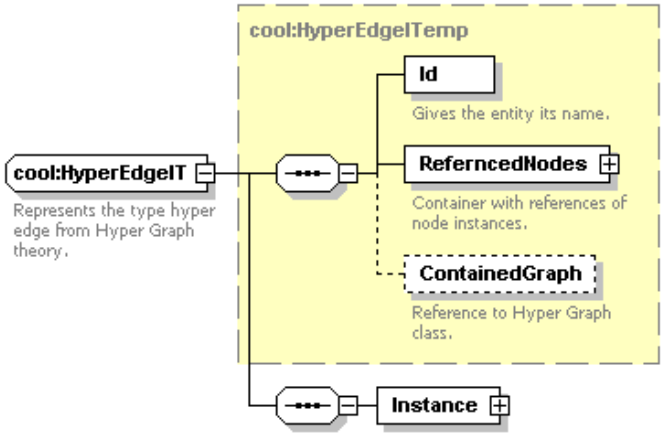
complexType cool:HyperEdgeCT

diagram	
type	extension of cool:HyperEdgeCTemp
children	Id ReferncedNodes ContainedGraph Class

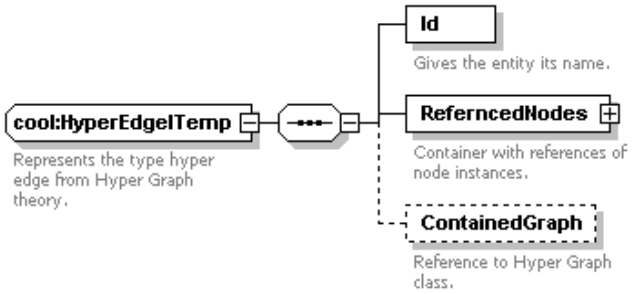
complexType cool:HyperEdgeCTemp

diagram	
type	restriction of cool:HyperEdgeT
children	Id ReferncedNodes ContainedGraph

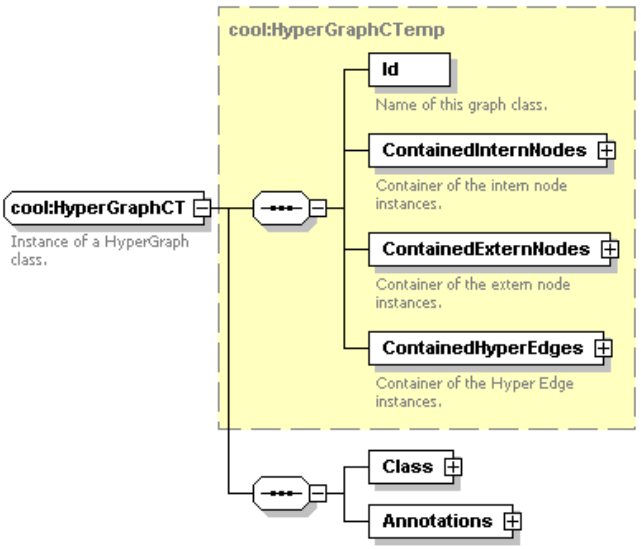
complexType cool:HyperEdgeIT

diagram	
type	extension of cool:HyperEdgeITemp
children	Id ReferncedNodes ContainedGraph Instance

complexType cool:HyperEdgeITemp

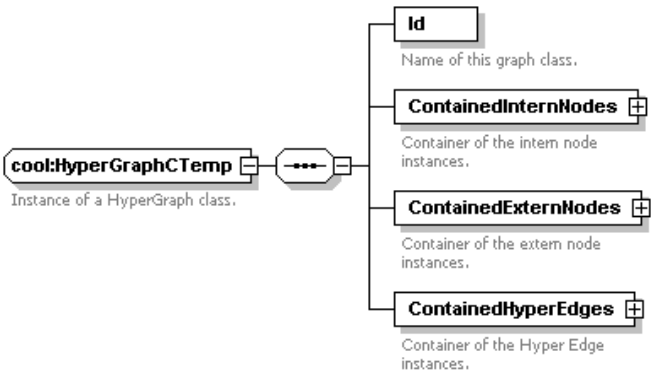
diagram	
type	restriction of cool:HyperEdgeT
children	Id ReferncedNodes ContainedGraph

complexType cool:HyperGraphCT

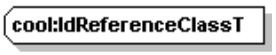
diagram	

type	extension of cool:HyperGraphCTemp
children	Id ContainedInternNodes ContainedExternNodes ContainedHyperEdges Class

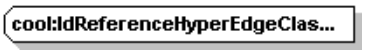
complexType cool:HyperGraphCTemp

diagram	
type	restriction of cool:HyperGraphT
children	Id ContainedInternNodes ContainedExternNodes ContainedHyperEdges

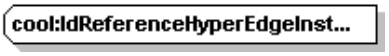
complexType cool:IdReferenceClassT

diagram	
type	extension of cool:IdReferenceHyperEdgeT
attributes	Name Type Use Default Fixed GloblID cool:IdT required URI cool:IdT required

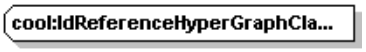
complexType cool:IdReferenceHyperEdgeClassT

diagram	
type	extension of cool:IdReferenceHyperGraphT
attributes	Name Type Use Default Fixed GloblID cool:IdT required URI cool:IdT required

complexType cool:IdReferenceHyperEdgeInstanceT

diagram	
type	extension of cool:IdReferenceHyperGraphT
attributes	Name Type Use Default Fixed GloblID cool:IdT required URI cool:IdT required

complexType cool:IdReferenceHyperGraphClassT

diagram	
type	extension of cool:IdReferenceHyperGraphT
attributes	Name Type Use Default Fixed GloblID cool:IdT required URI cool:IdT required

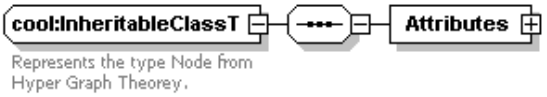
complexType cool:IdReferenceNodeClassT

diagram	
type	extension of cool:IdReferenceNodeT
attributes	Name Type Use Default Fixed GloblID cool:IdT required URI cool:IdT required

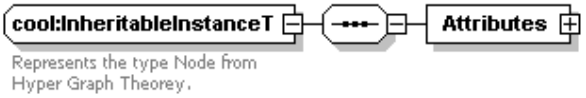
complexType cool:IdReferenceNodeInstanceT

diagram	
type	extension of cool:IdReferenceNodeT
attributes	Name Type Use Default Fixed GloblID cool:IdT required URI cool:IdT required

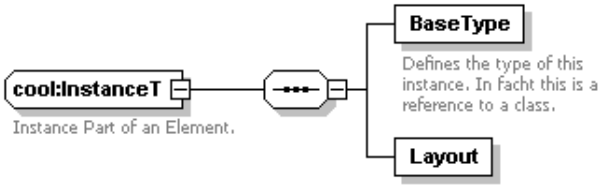
complexType cool:InheritableClassT

diagram	
children	Attributes

complexType cool:InheritableInstanceT

diagram	
children	Attributes

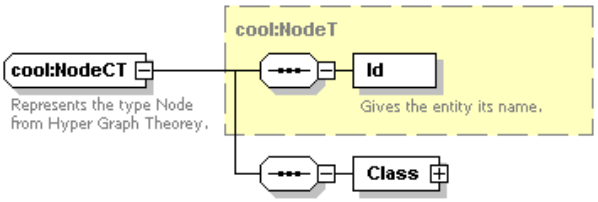
complexType cool:InstanceT

diagram	
children	BaseType Layout
attributes	Name Type Use Default Fixed Name xs:string required

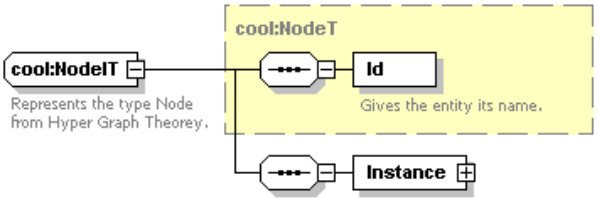
complexType cool:LayoutT

diagram	
attributes	Name Type Use Default Fixed Height xs:int required Width xs:int required XPos xs:int required YPos xs:int required Layer xs:int required Color xs:int required ReadOnly xs:boolean required ThrowEvent xs:boolean required

complexType cool:NodeCT

diagram	
type	extension of cool:NodeT
children	Id Class

complexType cool:NodeIT

diagram	
type	extension of cool:NodeT
children	Id Instance

complexType cool:TypeDefinitionT

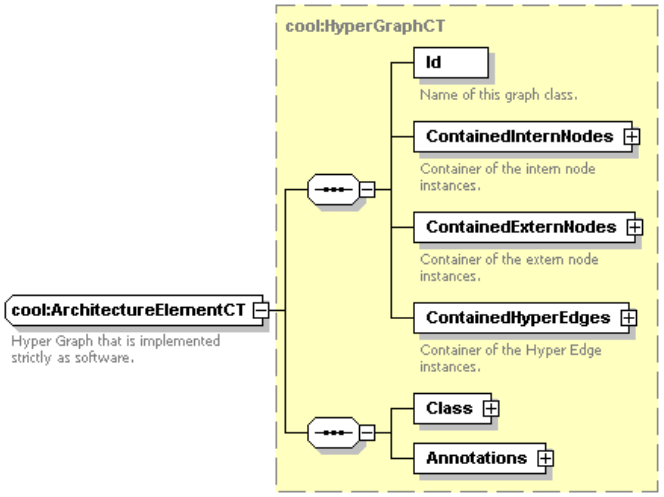
diagram	
attributes	Name Type Use Default Fixed Name xs:string required Type xs:string required

complexType cool:ValueT

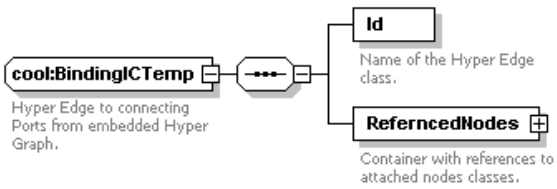
diagram	
attributes	Name Type Use Default Fixed Name xs:string required Value xs:string required

C.5 Cool_type_derivation.xsd

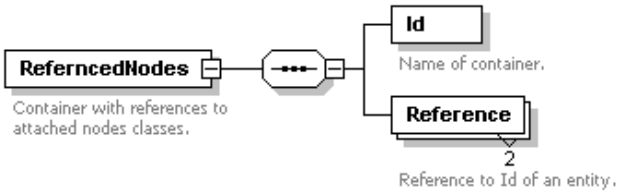
complexType cool:ArchitectureElementCT

diagram	
Type	extension of cool:HyperGraphCT
children	Id ContainedInternNodes ContainedExternNodes ContainedHyperEdges Class

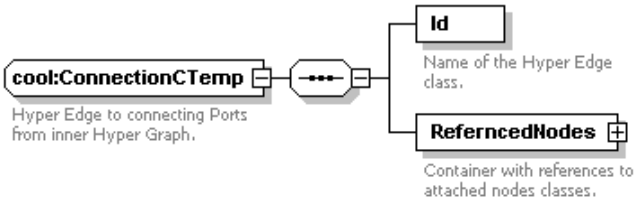
complexType cool:BindingICTemp

diagram	
type	restriction of cool:HyperEdgeCT
children	Id ReferncedNodes

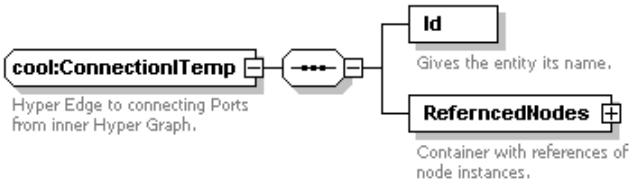
complexType cool:BindingITemp

diagram	
type	restriction of cool:HyperEdgeIT
children	Id ReferncedNodes

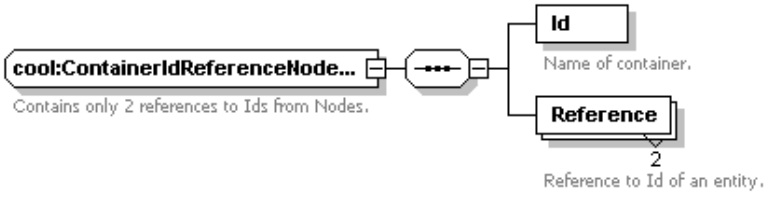
complexType cool:ConnectionCTemp

diagram	
type	restriction of cool:HyperEdgeCT
children	Id ReferncedNodes

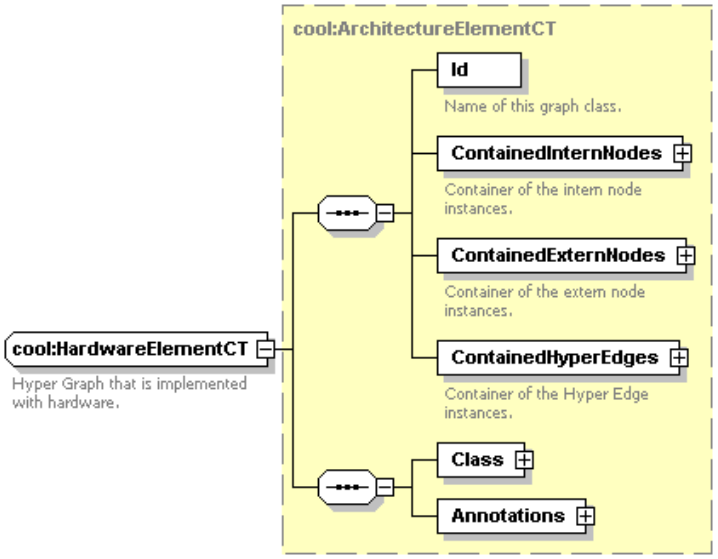
complexType cool:ConnectionITemp

diagram	
type	restriction of cool:HyperEdgeIT
children	Id ReferncedNodes

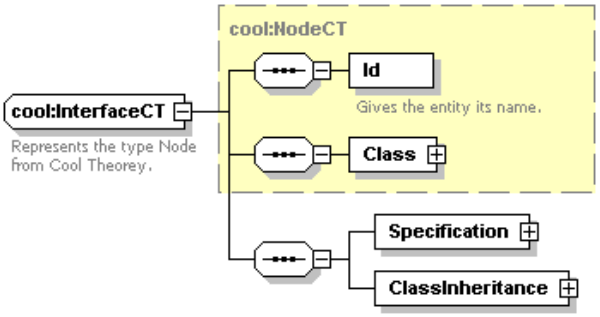
complexType cool:ContainerIdReferenceNodeClassBinaryT

diagram	
type	restriction of cool:ContainerIdReferenceNodeT
children	Id Reference

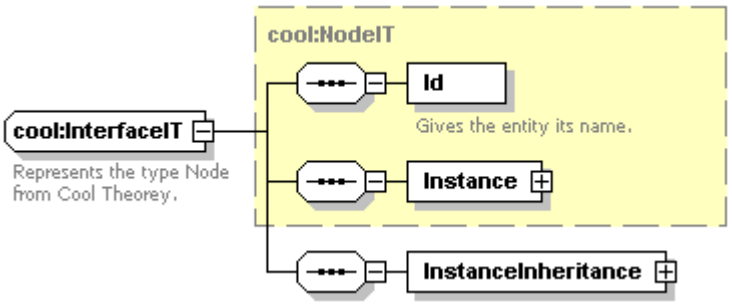
complexType cool:HardwareElementCT

diagram	
type	extension of cool:ArchitectureElementCT
children	Id ContainedInternNodes ContainedExternNodes ContainedHyperEdges Class

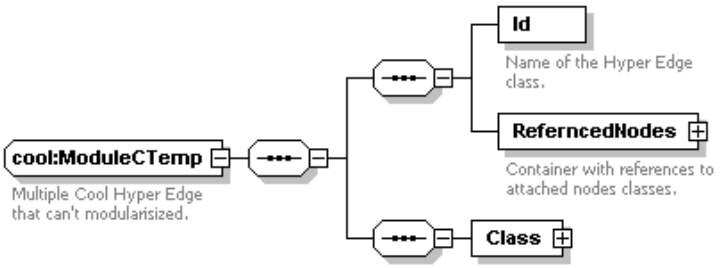
complexType cool:InterfaceCT

diagram	
type	extension of cool:NodeCT
children	Id Class Specification ClassInheritance

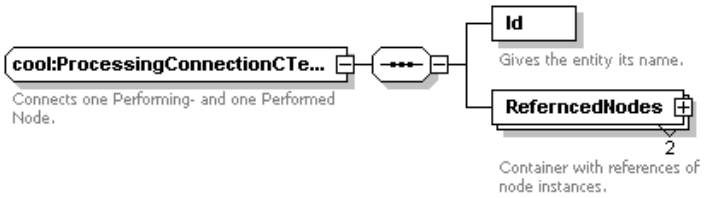
complexType cool:InterfaceIT

diagram	
type	extension of cool:NodeIT
children	Id Instance InstanceInheritance

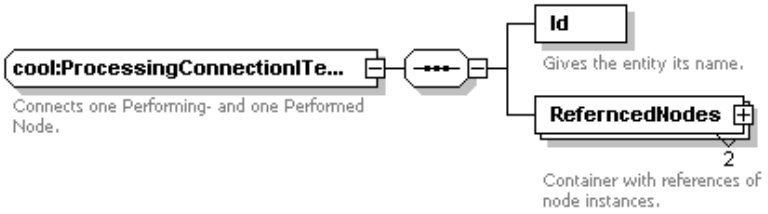
complexType cool:ModuleCTemp

diagram	
type	restriction of cool:HyperEdgeCT
children	Id ReferncedNodes Class

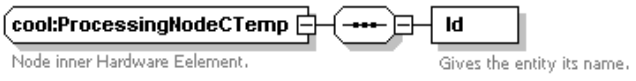
complexType cool:ProcessingConnectionCTemp

diagram	
type	restriction of cool:HyperEdgeIT
children	Id ReferncedNodes

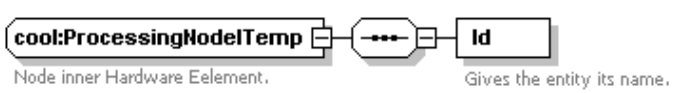
complexType cool:ProcessingConnectionITemp

diagram	
type	restriction of cool:HyperEdgeIT
children	Id ReferncedNodes

complexType cool:ProcessingNodeCTemp

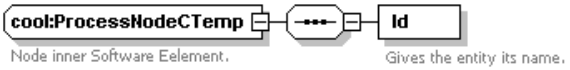
diagram	
type	restriction of cool:NodeCT
children	Id

complexType cool:ProcessingNodeITemp

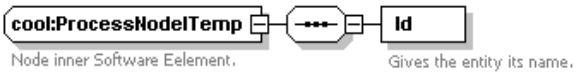
diagram	

type	restriction of cool:NodeIT
children	Id

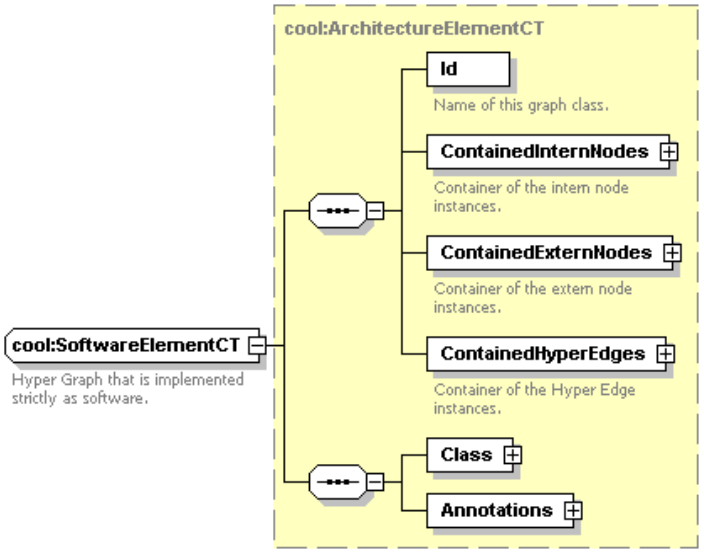
complexType cool:ProcessNodeCTemp

diagram	
type	restriction of cool:NodeCT
children	Id

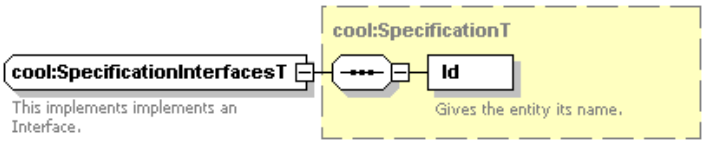
complexType cool:ProcessNodeITemp

diagram	
type	restriction of cool:NodeIT
children	Id

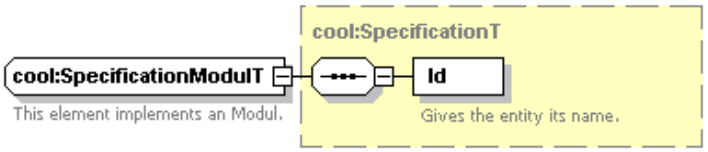
complexType cool:SoftwareElementCT

diagram	
type	extension of cool:ArchitectureElementCT
children	Id ContainedInternNodes ContainedExternNodes ContainedHyperEdges Class

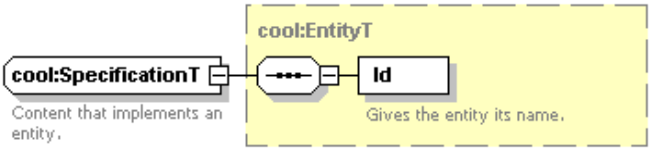
complexType cool:SpecificationInterfacesT

diagram	
type	extension of cool:SpecificationT
children	Id

complexType cool:SpecificationModulT

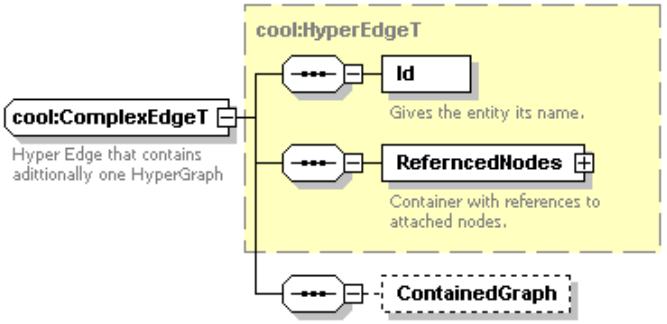
diagram	
type	extension of cool:SpecificationT
children	Id

complexType cool:SpecificationT

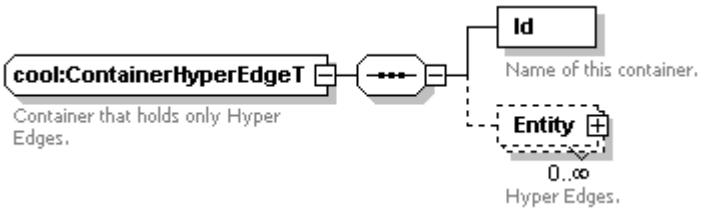
diagram	
type	extension of cool:EntityT
children	Id

C.6 Hyper_graph_elements.xsd

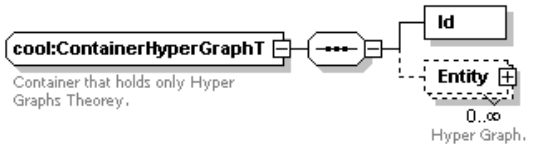
complexType cool:ComplexEdgeT

diagram	
type	extension of cool:HyperEdgeT
children	Id ReferncedNodes ContainedGraph

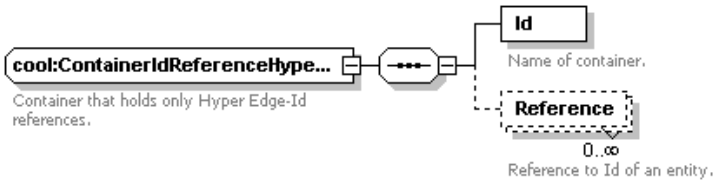
complexType cool:ContainerHyperEdgeT

diagram	
type	restriction of cool:ContainerT
children	Id Entity

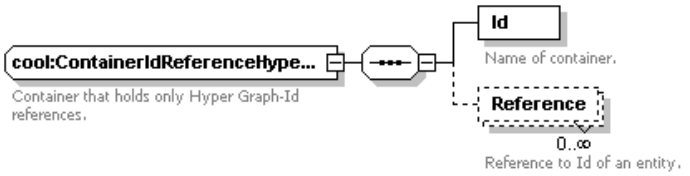
complexType cool:ContainerHyperGraphT

diagram	
type	restriction of cool:ContainerT
children	Id Entity

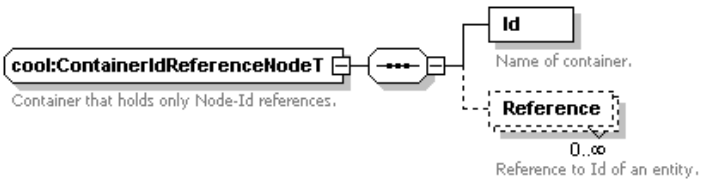
complexType cool:ContainerIdReferenceHyperEdgeT

diagram	
type	restriction of cool:ContainerIdReferenceT
children	Id Reference

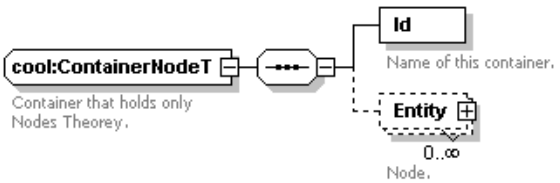
complexType cool:ContainerIdReferenceHyperGraphT

diagram	
type	restriction of cool:ContainerIdReferenceT
children	Id Reference

complexType cool:ContainerIdReferenceNodeT

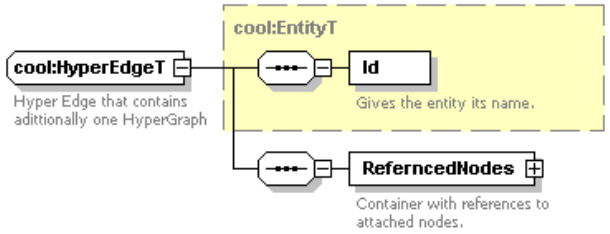
diagram	
type	restriction of cool:ContainerIdReferenceT
children	Id Reference

complexType cool:ContainerNodeT

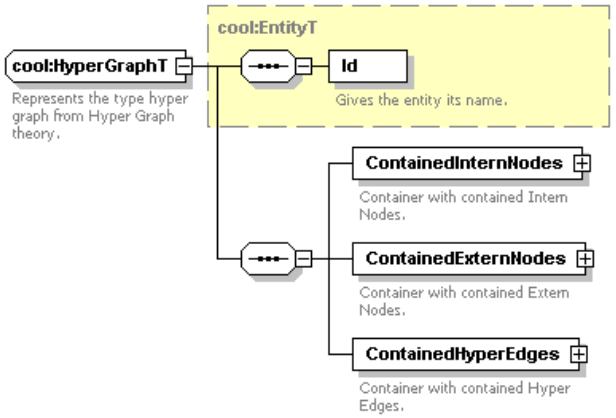
diagram	

type	restriction of cool:ContainerT
children	Id Entity

complexType cool:HyperEdgeT

diagram	
type	extension of cool:EntityT
children	Id ReferncedNodes

complexType cool:HyperGraphT

diagram	
type	extension of cool:EntityT
children	Id ContainedInternNodes ContainedExternNodes ContainedHyperEdges

complexType cool:IdReferenceHyperEdgeT

diagram	
type	extension of cool:IdReferenceT
attributes	Name Type Use Default Fixed GloblID cool:IdT required URI cool:IdT required

complexType cool:IdReferenceHyperGraphT

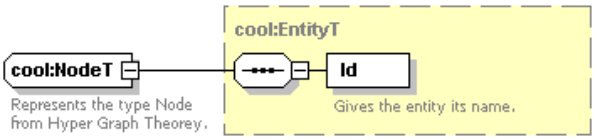
diagram	

type	extension of cool:IdReferenceT
attributes	Name Type Use Default Fixed GloblID cool:IdT required URI cool:IdT required

complexType cool:IdReferenceNodeT

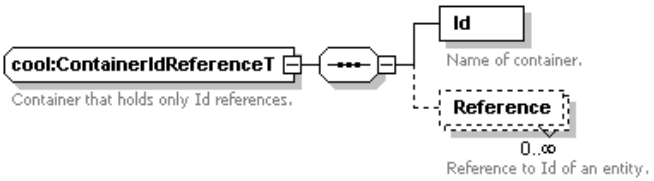
diagram	
type	extension of cool:IdReferenceT
attributes	Name Type Use Default Fixed GloblID cool:IdT required URI cool:IdT required

complexType cool:NodeT

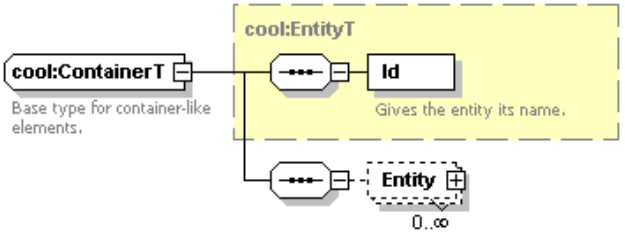
diagram	
type	extension of cool:EntityT
children	Id

C.7 Atomic_elements.xsd

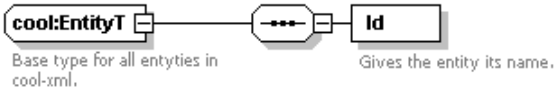
complexType cool:ContainerIdReferenceT

diagram	
children	Id Reference

complexType cool:ContainerT

diagram	
type	extension of cool:EntityT
children	Id Entity

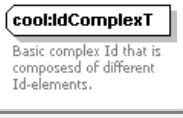
complexType cool:EntityT

diagram	
children	Id

complexType cool:HistoryItemT

diagram	
attributes	Name: Type: xs:date Use: required Default: Fixed: Name: Type: xs:string Use: required Default: Fixed: Description: Type: xs:string Use: optional Default: Fixed:

complexType cool:IdComplexT

diagram	
attributes	Name: Type: cool:IdT Use: required Default: Fixed: LocalID: Type: cool:IdT Use: required Default: Fixed: ParentID: Type: cool:IdT Use: required Default: Fixed: GlobalID: Type: cool:IdT Use: required Default: Fixed:

complexType cool:IdReferenceT

diagram	
attributes	Name: Type: cool:IdT Use: required Default: Fixed: GloblID: Type: cool:IdT Use: required Default: Fixed: URI: Type: cool:IdT Use: required Default: Fixed:

simpleType cool:FormatVersionT

type	restriction of xs:string
facets	pattern cool build (\.\d+)+(\.(\d{0,1}((alpha)|(beta)|(\p{Ll})))){0,1} pattern balance rc (\.\d+)+

simpleType cool:IdT

type	xs:string

XML Schema documentation generated with XMLSPY Schema Editor http://www.altova.com/xmlspy

Anhang D Musteroperatorensatz

D.1 Aufbau des Musterkataloges

Der im Folgenden dargestellte Musterkatalog enthält elf Transformationsmuster, welche bei ihrer Anwendung die Qualitätseigenschaft einer Softwarearchitektur positiv beeinflussen können. Eine Übersicht über alle enthaltenen Muster wird in der folgenden Tabelle gegeben:

Einfluss auf die Qualitätseigenschaften / Operatorname	Sicherheit	Zuverlässigkeit	Wartbarkeit	Verfügbarkeit	Echtzeitfähigkeit	Kosten
Mehrkanalige homogene Redundanz mit Mehrheitsentscheid	+	+/-	+	+/-	-	-
Zweikanalige homogene Redundanz	+	+	+	+	o/-	-
Mehrkanalige heterogene Redundanz mit Mehrheitsentscheid	+	+/-	+	+/-	-	-
Recovery Block	+	+	o	+	-	-
Watchdog	+	o	o	o	+	-
Heartbeat	+	+	o	+	o	-
Integritätscheck	+	+	o	o	-	-
Monitor	+	o	o	o	-	-
Hardwareelementaustausch	+/o	+/o	+/o	+/o	+/o	-
Neuzuweisung des Hardwareelementes	+/o	o	o	o	+	o
Vereinigung von Komponenten	+/-	o	o	o	+	+/-

Jedem Transformationsoperator wurde für jedes betrachtete Qualitätsmerkmal eine Bewertung gegeben, welche den Einfluss auf die Qualitätseigenschaften bei seiner Anwendung charakterisiert. Diese Bewertung basiert zum einen auf dem Erfahrungswissen bei der praktischen Anwendung des Transformationsoperators und zum anderen wurde die Bewertung aus den Beschreibungen des zugrunde liegenden Architekturentwurfsmusters abgeleitet.

D.2 Aufbau einer Transformationsmusterbeschreibung

Die Beschreibung der Transformationsoperatoren des Kataloges erfolgt in einer festgelegten Form. Diese Form entspricht einer Musterschablone (*pattern template*), welche in Anlehnung an /Gamma et al. 95/ und /Buschmann et al 96/ erstellt wurde und die in Tabelle 8-1 dargestellten Elemente enthält.
Die Spezifikation von Transformationsoperatoren in einer Musterschablone ermöglicht eine Vergleichbarkeit der einzelnen Muster sowie einen schnellen Überblick über die Inhalte des jeweiligen Musters. Je nach Intention sind für den Leser nur partielle Aspekte des Transformationsoperators relevant. Durch eine strukturierte Musterschablone wird das Finden dieser Aspekte erleichtert.

Tabelle 8-1 Bestandteile der Musterschablone

Element der Musterschablone	**Kurze Beschreibung**
Transformationsoperatorname	Eindeutiger aussagekräftiger Bezeichner des Transformationsoperators. Dieser wird im Folgenden durch die Überschrift des Transformationsoperators gegeben.
Problembeschreibung und Kontext	Beschreibung der Vorbedingungen für die Anwendung des Transformationsoperators und des Problems, welches durch die Operatoranwendung behoben werden soll.
Lösung	Beschreibung der zugrunde liegenden Lösungsidee des Transformationsoperators.
Implementierung	Spezifikation des implementierbaren Transformationsoperators mit einer

	Transformationsregel in der T-Notation, einer Beschreibung der Anwendungsbedingungen, einer Beschreibung der Regeln zur Variableninstanziierung, sowie einer Beschreibung eines Konstruktionsalgorithmus für die Interfacespezifikationen und Evaluationsmodelle für die hinzugefügten Architekturelementtypen.
Konsequenzen	Positive und negative Wechselwirkungen mit anderen Qualitätseigenschaften bei der Anwendung des Transformationsoperators
Referenzen auf verwandte Muster und Varianten	Spezialisierungen und Verweise auf ähnliche Transformationsoperatoren

D.3 Mehrkanalige homogene Redundanz mit Mehrheitsentscheid

Problembeschreibung und Kontext

Eine Softwarekomponente erfüllt seine Sicherheits-, Verfügbarkeits- oder Zuverlässigkeitsanforderungen nicht. Die Ursachen dafür sind Ausfälle der Hardwareplattform, auf welcher die Softwarekomponente ausgeführt wird.

Lösung

Die Idee zur Behebung des Problems besteht darin, durch den Transformationsoperator mehrere Softwarekomponenten redundant auf unabhängigen Hardwareplattformen zu realisieren. Zusätzlich wird in die Architektur eine *Voter/Replikator* Komponente eingefügt. Diese *Voter/Replikator* Komponente wird zwischen die redundanten Softwarekomponenten und den Rest der Architektur geschaltet und leitet alle Nachrichten von der Umgebung direkt an die Softwarekomponenten weiter. Im Gegensatz dazu werden jedoch alle erzeugten Nachrichten der redundanten Softwarekomponenten nur dann an die Umgebung übermittelt, wenn eine vorher festgelegte Anzahl der Softwarekomponenten identische Nachrichten sendet. Verwendet wird dazu ein Mehrheitsentscheid, welcher eine ungerade Anzahl an redundanten Softwarekomponenten voraussetzt. In der Praxis wird dabei aus ökonomischen Gründen ein 2-aus-3 Mehrheitsentscheid verwendet /Douglass 02/.

Implementierung

Positive Anwendungsbedingung:

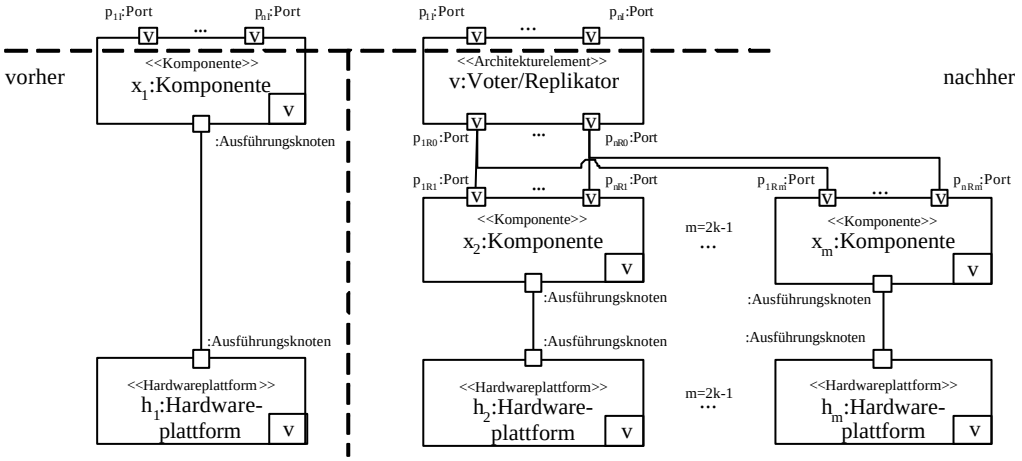
$A1^{+e}$: ErfülltVerfügbarkeitsAnforderungenNicht (x_1) $\vee$ ErfülltZuverlässigkeitsAnforderungenNicht (x_1) $\vee$ ErfülltSicherheitsAnforderungenNicht (x_1)

Negative Anwendungsbedingung:

$A2^{-e}$: $\exists$ v$\in$ Anwendungsarchitektur: (v.Typ=Voter/Replikator) $\wedge$Verschaltet(x_1,v)

$A3^{-o}$: ErfülltÖkonomischeAnforderungenNicht(System)

Transformationsoperator:



Regeln zur Variableninstanziierung:

$h_1=h_2=...=h_m$

$x_1=x_2=...=x_m$

$p_{1I}=p_{1R0}= p_{1R1}= \ldots=p_{1Rm}$

...
$p_{nI}=p_{nR0}= p_{nR1}= \ldots=p_{nRm}$

Konsequenzen

Eigenschaft	Auswirkung	Begründung
Sicherheit	+	Durch die Anwendung des Transformationsoperators wird die Wahrscheinlichkeit eines Ausfalles einer Hardwareplattform reduziert und somit auch die durch einen solchen Ausfall verursachten Gefährdungen. Dadurch wird die Sicherheit des Systems potentiell erhöht.
Zuverlässigkeit	+/-	Durch die Struktur nach der Operatoranwendung wird die Wahrscheinlichkeit eines einzelnen Ausfalles der Hardwareplattform reduziert. Vorraussetzung dafür ist, dass das Architekturelement Voter/Replikator eine geringere Ausfallwahrscheinlichkeit besitzt als die Hardwareplattformen der homogenen Softwarekomponenten. Des Weiteren müssen aber auch die Ausfallwahrscheinlichkeiten der Übertragungskanäle zwischen den verschiedenen Hardwareplattformen berücksichtigt werden.
Wartbarkeit	+	Die Wartbarkeit des Systems wird durch die Transformationsoperatoranwendung positiv beeinflusst, da Wartungsarbeiten bei einer Softwarekomponente oder einer Hardwareplattform in der Betriebsphase des Systems unabhängig von den anderen Kanälen durchführbar sind.
Verfügbarkeit	+/-	Wenn sich sowohl die Wartbarkeit als auch die Zuverlässigkeit durch die Anwendung des Transformationsoperators verbessern, so verbessert sich auch die Verfügbarkeit.
Echtzeitfähigkeit	-	Das Laufzeitverhalten wird durch den Zeitaufwand für die Weiterleitung der Nachrichten und für den Mehrheitsentscheid negativ beeinflusst.
Kosten	-	Die Kosten für die Systemerstellung erhöhen sich durch die mehrfache Realisierung der redundanten Hardwareplattformen.

Referenzen auf verwandte Muster und Varianten

Multi-Channel Homogeneous Redundancy /Douglass 99/

D.4 Zweikanalige homogene Redundanz

Kontext und Problembeschreibung

Das zugrunde liegende Problem ist ähnlich dem Problem des Musters mehrkanalige homogene Redundanz. Dabei erfüllt eine Softwarekomponente seine Sicherheits-, Verfügbarkeits- oder Zuverlässigkeitsanforderungen nicht. Die Ursache dafür ist ein Ausfall der Hardwareplattform, auf welcher die Softwarekomponente ausgeführt wird. Zusätzlich bestehen jedoch Einschränkungen bei den Entwicklungs- und Herstellungskosten. Aus diesem Grund ist es nicht möglich, mehrere Hardwareplattformen, wie bei der mehrkanaligen homogenen Redundanz, einzufügen.

Lösung

Zur Lösung des Problems werden im Gegensatz zur mehrkanaligen homogenen Redundanz nur zwei Kanäle auf unabhängigen Hardwareplattformen ausgeführt. Einer der beiden Kanäle übernimmt dabei die Rolle des aktiven Kanals und der andere die Rolle des passiven Kanals. Der passive Kanal wird auch als Reservekanal bezeichnet. Ein Kanal besteht aus der Softwarekomponente und einem Architekturelement, welches als *Kanalvalidation* bezeichnet wird.

Die Aufgabe der *Kanalvalidations*-Komponente des aktiven Kanals ist die Überprüfung der Nachrichten, welche mit der Umgebung ausgetauscht werden. Wird dabei ein sicherheitsrelevantes Fehlverhalten erkannt, so tauschen der aktive und der passive Kanal seine Rollen.

Die Aufgabe der *Kanalvalidations*-Komponente des passiven Kanals ist zu überprüfen, ob der aktive Kanal sich noch korrekt verhält. Wird dabei ein Ausfall oder Abweichen vom korrekten Verhalten erkannt, so übernimmt der bisherige passive Kanal die Aufgaben des aktiven Kanals.

Implementierung

Positive Anwendungsbedingung:

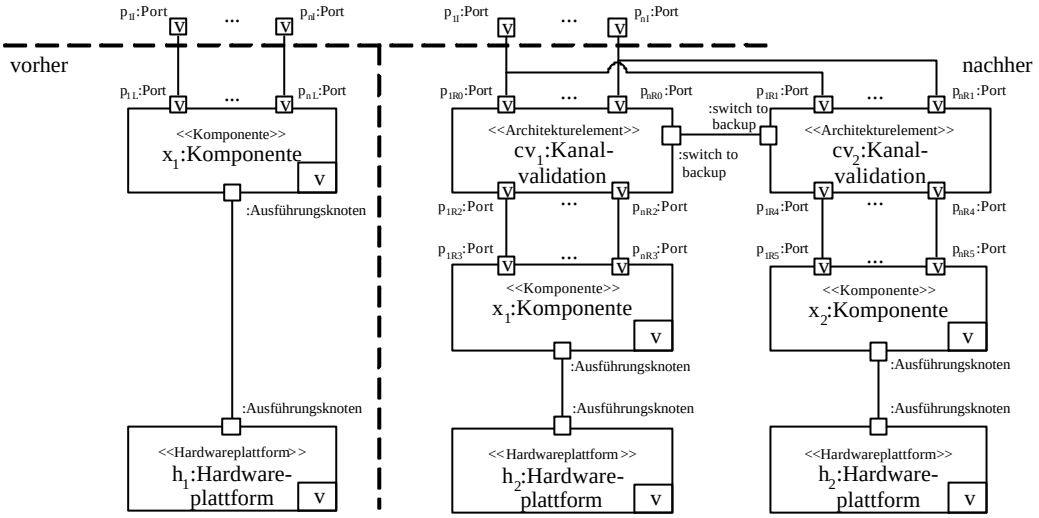
$A1^{+e}$: ErfülltVerfügbarkeitsAnforderungenNicht (x_1) $\vee$ ErfülltZuverlässigkeitsAnforderungenNicht (x_1) $\vee$ ErfülltSicherheitsAnforderungenNicht (x_1)

Negative Anwendungsbedingung:

$A2^{-e}$: $\exists$ $cv_1 \in$ Anwendungsarchitektur: (cv_1.Typ=Kanalvalidation) $\wedge$Verschaltet(x_1, cv_1)

$A3^{-o}$: ErfülltÖkonomischeAnforderungenNicht(System)

Transformationsoperator:



Regeln zur Variableninstanziierung:

$h_1=h_2=...=h_m$

$x_1=x_2=...=x_m$

$p_{1I}= p_{1L}=p_{1R0}= p_{1R1}= p_{1R2}= p_{1R3}= p_{1R4}= p_{1R5}$

...

$p_{nI}= p_{nL}=p_{nR0}= p_{nR1}= p_{nR2}= p_{nR3}= p_{nR4}= p_{nR5}$

Konsequenzen

Eigenschaft	Auswirkung	Begründung
Sicherheit	+	Die Sicherheit des Systems erhöht sich, wenn vor der Musteranwendung durch einen Ausfall der Hardwareplattform eine Gefährdung verursacht wird. Durch das Einfügen der *Kanalvalidations*-Komponente ist es zudem möglich, ein sicherheitskritisches Verhalten einer Komponente zu erkennen und mit dem Umschalten auf den Reservekanal eine geeignete Maßnahme zur Risikominimierung einzuleiten. Dadurch erhöht sich ebenfalls die Sicherheit des Systems.
Zuverlässigkeit	+	Die Zuverlässigkeit des Systems erhöht sich, da durch die Struktur der Einfluss eines Hardwareausfalles reduziert wird.
Wartbarkeit	+	Die Wartbarkeit des Systems wird durch die Transformationsoperatoranwendung positiv beeinflusst, da Wartungsarbeiten bei einer Softwarekomponente des passiven Kanals während der Betriebsphase des Systems durchführbar sind.
Verfügbarkeit	+	Siehe Zuverlässigkeit und Wartbarkeit
Echtzeitfähigkeit	o/-	Durch das Hinzufügen zusätzlicher aktiver Softwarekomponenten kann das Schedulingverhalten negativ beeinflusst werden.
Kosten	-	Die Kosten für die Systemerstellung erhöhen sich, auf Grund der zusätzlich zu entwickelnden Komponenten und der zusätzlichen Hardwareplattform für den zweiten Kanal.

Referenzen auf verwandte Muster und Varianten

Switch to back up /Saridakis 02/ Two-Channel-Redundancy /Douglass 02/

D.5 Mehrkanalige heterogene Redundanz mit Mehrheitsentscheid

Kontext und Problembeschreibung

Das Problem, welches es durch den Transformationsoperator „Mehrkanalige heterogene Redundanz mit Mehrheitsentscheid“ zu lösen gilt, ist eine Softwarekomponente, die ihre Sicherheits-, Zuverlässigkeits- und Verfügbarkeitsanforderungen nicht einhält. Der Grund dafür sind Fehlverhalten auf Basis systematischer Fehlerr.

Lösung

Zur Problemlösung wird die Strukturspezifikation ähnlich wie bei der homogenen Redundanz verändert. Die Softwarekomponente wird auf verschiedenen Hardwareplattformen realisiert. Die Kommunikation zwischen der Komponentenumgebung und den redundanten Softwarekomponenten erfolgt über eine *Voter/Replikator* Komponente, welche einen Mehrheitsentscheid implementiert. Zur Erhöhung der Sicherheit gegenüber systematischen Fehlern werden jedoch keine homogenen Komponenten verwendet, da diese unter den gleichen Umgebungsbedingungen die gleichen Fehlverhalten offenbaren, sondern es werden heterogene Softwarekomponenten verwendet. Die heterogenen Softwarekomponenten werden durch neue Softwarekomponententypen beschrieben, welche in die Architekturspezifikation integriert und von dem ursprünglichen Softwarekomponententyp abgeleitet werden. Da sich die Softwarekomponententypen unterscheiden, müssen diese jedoch getrennt entwickelt werden. Erfolgt dies durch unterschiedliche Entwicklungsteams, so wird die Wahrscheinlichkeit des Auftretens eines systematischen Fehlers reduziert /Avizienis 85/, da die Fehlerverhalten durch die *Voter/Replikator* Komponente erkannt werden. Nach einer Studie in /Knight, Leveson 86/ lassen sich durch eine heterogene Implementierung jedoch nicht alle auf systematischen Fehlern beruhende Fehlverhalten erkennen. Dies ist der Fall, da selbst bei der Entwicklung der heterogenen Softwarekomponenten mit verschiedenen Entwicklungsteams und Entwicklungsmethoden identische Fehler auftreten. Der Grund dafür liegt zum einen in der Komplexität der Komponente und Tatsache, dass unterschiedliche Entwicklungsteams oft dieselben systematischen Fehler erzeugen /Knight, Leveson 86/. Zum anderen kann aber auch die Komponentenspezifikation selbst fehlerhaft sein. Daher ist das Ausfallverhalten auf Basis systematischer Fehler bei heterogenen Komponenten nicht unabhängig, und durch den Transformationsoperator „Mehrkanalige heterogene Redundanz mit Mehrheitsentscheid“ kann keine vollständige Sicherheit gegenüber systematischen Fehlern gewährleistet werden.

Implementierung

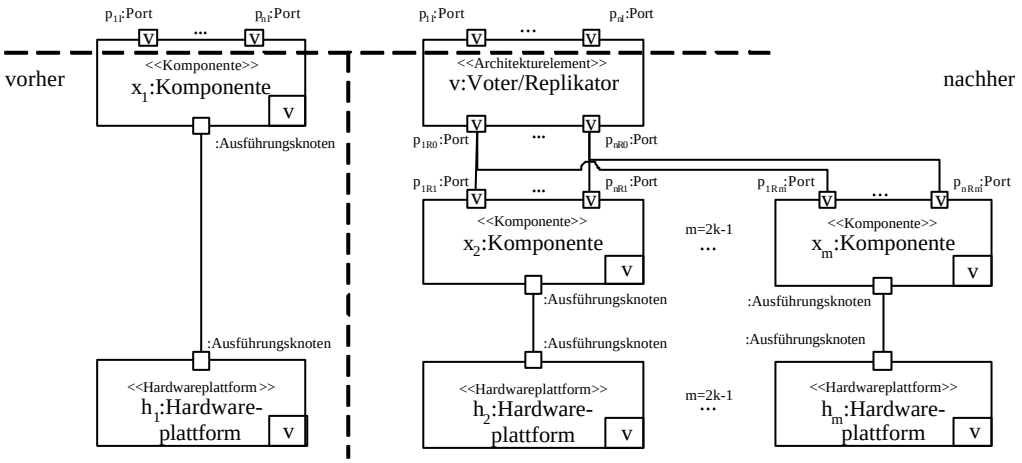
Positive Anwendungsbedingung:

$A1^{+e}$: ErfülltVerfügbarkeitsAnforderungenNicht (x_1) $\vee$ ErfülltZuverlässigkeitsAnforderungenNicht (x_1) $\vee$ ErfülltSicherheitsAnforderungenNicht (x_1)

Negative Anwendungsbedingung:

$A2^{-e}$: $\exists$ v$\in$ Anwendungsarchitektur: (v.Typ=Voter/Replikator) $\wedge$Verschaltet(x_1,v)

$A3^{-o}$: ErfülltÖkonomischeAnforderungenNicht(System)



Regeln zur Variableninstanziierung:

$h_1=h_2=...=h_m$

$x_2 \mid x_1,..., x_m \mid x_1$

$p_{1I}=p_{1R0}= p_{1R1}= \ldots=p_{1Rm}$

...

$p_{nI}=p_{nR0}= p_{nR1}= \ldots=p_{nRm}$

Die Strukturtransformationsregel sowie die Evaluationsmodelle und Interfacespezifikationen des Architekturelementes Voter/Replikator entsprechen denen des Transformationsoperators Homogene Redundanz mit Mehrheitsentscheid. Unterschiede ergeben sich jedoch bei den Instanziierungsregeln des Transformationsoperators. Durch sie werden die Komponenten x_2 bis x_m von der Komponente x_1 abgeleitet. Dadurch werden neue Architekturtypen zur Architekturspezifikation hinzugefügt, welche durch unterschiedliche Entwicklungsteams realisiert werden müssen.

Konsequenzen

Eigenschaft	Auswirkung	Begründung
Sicherheit	+	Die Sicherheit des Systems kann sich bei der Operatoranwendung erhöhen, da die Auswirkung von systematischen Fehlern auf das Systemverhalten reduziert wird. Ebenfalls werden die Auswirkungen eines Hardwareausfalles durch die Operatoranwendung reduziert.
Zuverlässigkeit	+	Siehe Sicherheit
Wartbarkeit	o	
Verfügbarkeit	+	Siehe Sicherheit
Echtzeitfähigkeit	o	
Kosten	-	Die Kosten für die Systemerstellung erhöhen sich, da für jede heterogene Softwarekomponente zusätzliche Entwicklungskosten anfallen. Des Weiteren entstehen Kosten durch die redundante Verwendung der Hardwareplattformen.

Referenzen auf verwandte Muster und Varianten

Heterogene-Redundanz /Douglass 99/

D.6 Recovery Block

Kontext und Problembeschreibung

Eine Softwarekomponente hält seine Sicherheitsanforderungen nicht ein. Grund dafür sind Fehlverhalten, welche von systematischen Fehlern verursacht wurden. Für diese Komponente bestehen hohe Verfügbarkeitsanforderungen, so dass bei einem identifizierten Fehlverhalten die Softwarekomponente weiterhin ihre Funktion erfüllen muss. Zudem ist das Budget bei der Auswahl dieses Transformationsoperators relevant.

Lösung

Zur Lösung des Problems wird das Recovery Block Konzept verwendet, welches unter anderem in /Randel 75/, /Avizienis 85/ und /Randell, Xu 95/ beschrieben ist. Die Idee dieses Konzeptes ist es, anstelle der Softwarekomponente mehrere heterogene Softwarekomponenten zu verwenden, welche nach einer identischen Interfacespezifikation, aber von unterschiedlichen Entwicklungsteams erstellt wurden. Im Gegensatz zur heterogenen Redundanz laufen diese Softwarekomponenten jedoch auf einer Hardwareplattform ab und nur eine dieser heterogenen Komponenten ist aktiv. Die anderen Softwarekomponenten dienen als Backup.

Zur Identifizierung eines Fehlverhaltens bei der aktiven Komponente wird eine spezielle Fehlererkennungs-Komponente verwendet, welche als *Akzeptanztest*-Komponente bezeichnet wird. Zur Realisierung dieser Komponente lassen sich die Muster Actuation-Monitor /Douglass 02/, /Grunske 03a/, Integrity Check /Grunske 03a/ und Watchdog /Douglass 02/, /Grunske 03a/, /Pont, Ong 02/, /Saridakis 02/ verwenden. Wird ein Fehlverhalten bei der aktiven Komponente identifiziert, so wird aus dem Pool der Backup-Komponenten eine neue aktive Komponente bestimmt. Bei einer zustandslosen Komponente ist dieser Wechsel ohne Probleme möglich. Bei zustandsbehafteten Komponenten kann in der ersten Variante der Zustand der bisherigen aktiven Komponente extern gespeichert werden. Dieser Zustand wird von der neuen aktiven Komponente geladen und für die Bestimmung der nachfolgenden ausgehenden Nachrichten genutzt. In der zweiten Variante erhalten alle heterogenen Softwarekomponenten von der Komponentenumgebung identische Nachrichten. Die Komponenten müssen diese Nachrichten parallel bearbeiten und sollten somit den aktuell korrekten Zustand haben. Das Problem, welches sich bei dieser Variante ergibt, ist eine Verschlechterung der Performanz.

Implementierung

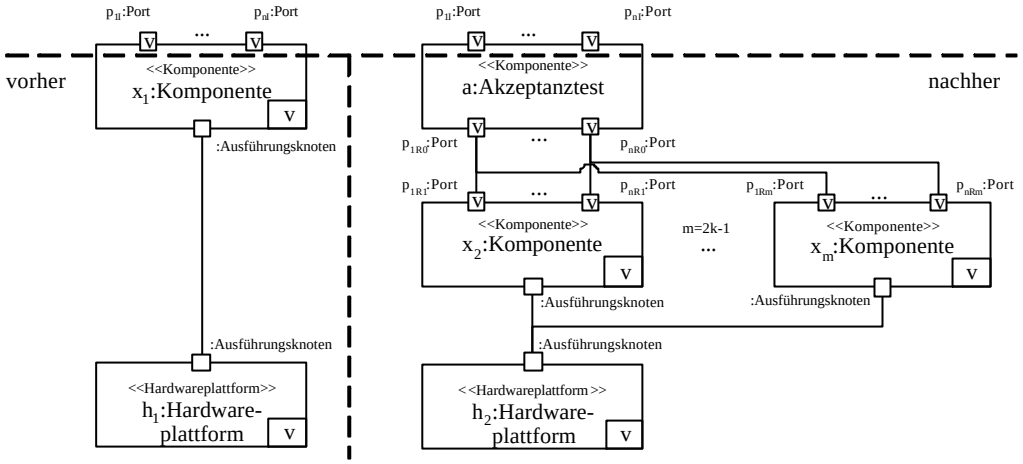
Positive Anwendungsbedingung:

$A1^{+e}$: ErfülltVerfügbarkeitsAnforderungenNicht (x_1) $\vee$ ErfülltZuverlässigkeitsAnforderungenNicht (x_1) $\vee$ ErfülltSicherheitsAnforderungenNicht (x_1)

Negative Anwendungsbedingung:

$A2^{-e}$: $\exists$ v$\in$ Anwendungsarchitektur: (v.Typ=Akzeptanztest) $\wedge$Verschaltet(x_1,v)

$A3^{-o}$: ErfülltÖkonomischeAnforderungenNicht(System)



Regeln zur Variableninstanziierung:

$h_1=h_2=...=h_m$

$x_2 \mid x_1,..., x_m \mid x_1$

$p_{1I}=p_{1R0}= p_{1R1}= \ldots=p_{1Rm}$

...

$p_{nI}=p_{nR0}= p_{nR1}= \ldots=p_{nRm}$

Anmerkung:

Ein Integritätstest der Komponenten wird durch den Operator nicht hinzugefügt, da dies die Schnittstellen der einzelnen Komponenten verändern würde. Daher muss die Integration des Integritätstestmechanismus manuell nach der Operatoranwendung erfolgen.

Konsequenzen

Eigenschaft	Auswirkung	Begründung
Sicherheit	+	Die Sicherheit des Systems kann sich bei der Operatoranwendung erhöhen, da durch die Transformatoranwendung die Auswirkung von Fehlverhalten auf Basis systematischer Fehler auf die Systemumwelt reduziert wird.
Zuverlässigkeit	+	Die Zuverlässigkeit des Systems kann sich bei der Operatoranwendung erhöhen, da durch die Transformatoranwendung die Auswirkung von Fehlverhalten auf Basis systematischer Fehler reduziert wird.
Wartbarkeit	o	
Verfügbarkeit	+	Da die Zuverlässigkeit durch die Transformatoranwendung verbessert wird, wird auch die Verfügbarkeit erhöht.
Echtzeitfähigkeit	-	Das Laufzeitverhalten wird durch die Mechanismen zur Identifizierung von Fehlverhalten in der *Akzeptanztest*-Komponente negativ beeinflusst. Zusätzlich wird durch das Hinzufügen von aktiven Softwarekomponenten das Schedulingverhalten negativ beeinflusst.
Kosten	-	Die Kosten für die Systemerstellung erhöhen sich, da für jede heterogene Softwarekomponente Entwicklungskosten anfallen. Zudem muss die *Akzeptanztest*-Komponente zusätzlich erstellt werden.

Referenzen auf verwandte Muster und Varianten

Recovery Block /Randel 75/, /Randell, Xu 95/

D.7 Watchdog

Kontext und Problembeschreibung

Das zugrunde liegende Problem des Watchdog-Musters ist eine Softwarekomponente, welche zu früh oder zu spät auf ein Ereignis der Komponentenumgebung reagiert. Somit verletzt diese Softwarekomponente primär die Anforderungen an die Echtzeitfähigkeit. Sekundär verletzt die Softwarekomponente auch ihre Zuverlässigkeits-, Verfügbarkeits- oder Sicherheitsanforderungen, wenn sie gar nicht, also unendlich zu spät reagiert, oder wenn aus dem temporalen Fehlverhalten eine Gefährdung der Systemumgebung resultiert.

Lösung

Zur Lösung des Problems werden zwei zusätzliche Softwarekomponenten in die Architekturspezifikation integriert. Die erste Komponente wird als *Watchdog* bezeichnet. Diese Komponente enthält einen unabhängigen Zeitgeber und kann somit überprüfen, ob die beobachtete Komponente zu früh oder zu spät reagiert. Dazu empfängt die *Watchdog*-Komponente alle Ereignisse, welche an die beobachtete Softwarekomponente gesendet werden. Muss die beobachtete Komponente auf ein Ereignis nach der Interfacespezifikation innerhalb einer definierten Zeitschranke reagieren, so wird der Zeitgeber gestartet. Reagiert die beobachtete Komponente außerhalb dieser Zeitschranke, so wird eine *Fehlerbehebungs*-Komponente benachrichtigt. Diese Komponente implementiert eine geeignete Strategie zur Reduktion der Auswirkungen des Fehlverhaltens. Die Auswahl der Strategie ist jedoch abhängig vom Anwendungsgebiet, in welchem das System eingesetzt wird. Als Beispiel kann diese Komponente das System durch ein Herunterfahren in einen sicheren Zustand bringen. Dabei ist klar, dass dies nur für Systeme funktioniert, die einen sicheren Zustand besitzen. Bei lebenserhaltenden Systemen in der Medizin oder Systemen in der Avionik ist die Anwendung der Strategie nicht möglich.

Implementierung

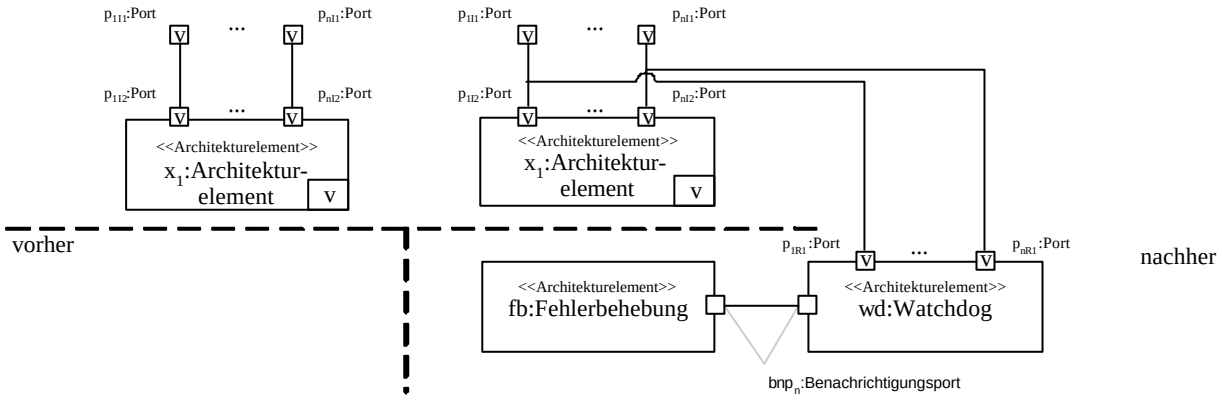
Positive Anwendungsbedingung:

$A1^{+e}$: ErfülltVerfügbarkeitsAnforderungenNicht (x_1) $\vee$ ErfülltZuverlässigkeitsAnforderungenNicht (x_1) ErfülltEchtzeitAnforderungenNicht (x_1) $\vee$ ErfülltSicherheitsAnforderungenNicht (x_1)

Negative Anwendungsbedingung:

$A2^{-e}$: $\exists$ wd$\in$ Anwendungsarchitektur: (wd.Typ=Watchdog) $\wedge$Verschaltet(x_1,wd)

$A3^{-o}$: ErfülltÖkonomischeAnforderungenNicht(System)



Konsequenzen

Eigenschaft	Auswirkung	Begründung
Sicherheit	+	Die Sicherheit des Systems kann sich bei der Operatoranwendung erhöhen, wenn ein Fehlverhalten, bei dem die Komponente zu früh oder zu spät reagiert, zu einer Gefährdung führt. Diese Fehlverhalten können nach der Operatoranwendung erkannt und geeignet behandelt werden.
Zuverlässigkeit	+	Durch die *Watchdog*-Komponente kann ein Ausfall eines Architekturelementes erkannt werden, und geeignete Maßnamen zur Behebung des Ausfalles können ergriffen werden.
Wartbarkeit	o	
Verfügbarkeit	+	Da die Zulässigkeit durch die Transformatoranwendung verbessert wird, wird auch die Verfügbarkeit erhöht.

Echtzeitfähigkeit	+	Durch Anwendung des Watchdog-Musters werden temporale Fehlverhalten erkannt und lassen sich geeignet beheben.
Kosten	-	Die Kosten für die Systemerstellung erhöhen sich, da für die *Watchdog*-Komponente zusätzliche Entwicklungskosten anfallen.

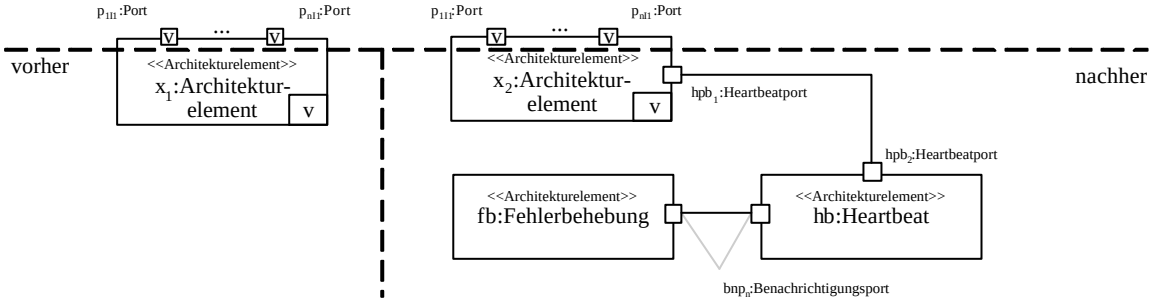
Referenzen auf verwandte Muster und Varianten
Watchdog Timer /Mahmood, McCluskey 88/, Watchdog Processor und andere Realisierungen des Watchdog-Musters /Pont, Ong 02/

D.8 Heartbeat

Kontext und Problembeschreibung
Das zugrunde liegende Problem des Heartbeat-Musters sind Ausfälle von Architekturelementen und die damit verbundene Verletzung von Zuverlässigkeits- und Verfügbarkeitsanforderungen sowie die Nichteinhaltung der Sicherheitsanforderungen, wenn durch einen solchen Ausfall eine Gefährdung verursacht wird.

Lösung
Zur Erkennung eines Ausfalls eines Architekturelementes wird mit der *Heartbeat*-Komponente eine zusätzliche Komponente in die Architektur integriert. Für die Realisierung dieser Komponente existieren zwei Varianten. Bei der ersten Variante sendet die *Heartbeat*-Komponente in wohldefinierten Abständen Anfragen an das beobachtete Architekturelement. Sobald dieses in einer festgelegten Zeit nicht reagiert, wird ein Ausfall angenommen. In der zweiten Variante muss die beobachtete Komponente von selbst in wohldefinierten Abständen eine Nachricht an die *Heartbeat*-Komponente senden. Unterlässt die beobachtete Komponente diese Nachricht, so wird sie als ausgefallen betrachtet. Beide Varianten weisen eine starke Ähnlichkeit zum *Watchdog*-Muster auf. Daher kann auch für die *Heartbeat*-Komponente eine ähnliche Implementierung verwendet werden wie bei der *Wachdog*-Komponente.
Wird durch die *Heartbeat*-Komponente ein Ausfall erkannt, so wird über den Benachrichtigungsport ein Alarmsignal an eine *Fehlerbehebungs*-Komponente gesendet. Diese wird ähnlich dem Watchdog-Muster mit einer geeigneten anwendungsabhängigen Fehlerbehebungsstrategie implementiert.

Implementierung
Positive Anwendungsbedingung:
$A1^{+e}$: ErfülltVerfügbarkeitsAnforderungenNicht (x_1) $\vee$ ErfülltZuverlässigkeitsAnforderungenNicht (x_1) $\vee$ ErfülltSicherheitsAnforderungenNicht (x_1)
Negative Anwendungsbedingung:
$A2^{-e}$: $\exists$ hb$\in$ Anwendungsarchitektur: (hb.Typ=Heartbeat) $\wedge$Verschaltet(x_1,hb)
$A3^{-o}$: ErfülltÖkonomischeAnforderungenNicht(System)



Regeln zur Variableninstanziierung:
$x_2 \mid x_1$

Konsequenzen

Eigenschaft	Auswirkung	Begründung
Sicherheit	+	Die Sicherheit des Systems erhöht sich bei der Operatoranwendung, da sicherheitskritische Fehlverhalten der Komponente erkennbar und reduzierbar sind.
Zuverlässigkeit	+	Die Zuverlässigkeit des Systems kann sich erhöhen, da Ausfälle der Komponente erkennbar sind und geeignet behoben werden können.

Wartbarkeit	o	
Verfügbarkeit	+	Auf Basis der besseren Zuverlässigkeit erhöht sich auch die Verfügbarkeit.
Echtzeitfähigkeit	o	
Kosten	-	Die Kosten für die Systemerstellung erhöhen sich, da für die *Heartbeat*-Komponente zusätzliche Entwicklungskosten anfallen.

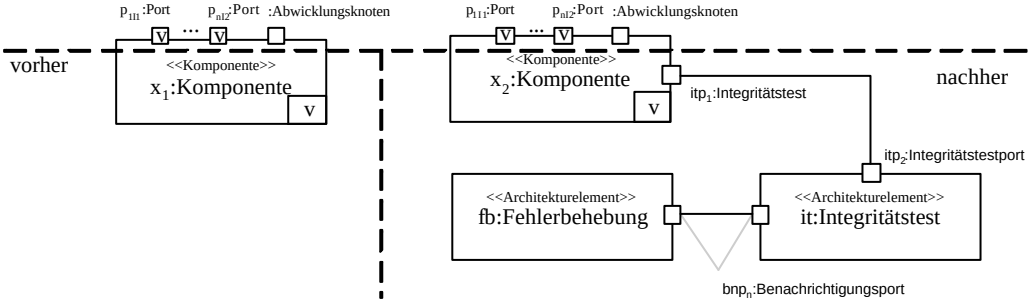
Referenzen auf verwandte Muster und Varianten
Watchdog Timer /Mahmood, McCluskey 88/, Heartbeat, I am Alive /Saridakis 02/, Are You Alive /Saridakis 02/

D.9 Integritätscheck

Kontext und Problembeschreibung
Eine Softwarekomponente hält ihre Sicherheitsanforderungen nicht ein. Der Grund dafür ist ein Abweichen vom korrekten Verhalten während des Betriebs der Komponente. Dieses Abweichen wird durch eine Veränderung des Objektcodes im Code- oder Datensegment der Komponente verursacht. Eine solche Objektcodeveränderung kann durch externe Einflüsse wie zum Beispiel elektromagnetische Felder oder kosmische Strahlung erfolgen. Des Weiteren ist auch durch eine unsaubere Programmierung einer anderen Komponente und ein unzureichender Schutz des Code- oder Datensegments der Komponente eine Beeinflussung möglich.

Lösung
Zur Lösung des Problems wird das Code- und/oder Datensegment um zusätzliche redundante Bits erweitert. Diese Bits sind Prüfbits, mit denen die Integrität des Objektcodes überprüfbar ist. Verwendet werden dazu Prüfsummen, CRC-Bits oder Paritäts-Bits /Stone et al 98/. Die Integration der Prüfbits unterscheidet sich für das Code- und das Datensegment. Das Codesegment ist statisch und ändert sich während des Betriebs einer Komponente nicht. Daher lassen sich die Prüfbits bereits vor der Auslieferung der Komponente bestimmen und integrieren. Das Datensegment ändert sich während des Betriebs einer Komponente bei jedem Schreibzugriff auf die Daten. Daher ist die Bestimmung der Prüfbits erst während des Betriebs der Komponente möglich. Folglich müssen die Prüfbits bei jedem schreibenden Zugriff auf ein Datum neu berechnet werden.
Für die Überprüfung der Integrität wird eine zusätzliche Komponente in die Architektur eingefügt. Diese Komponente wird als *Integritätstest* bezeichnet. Erkennt die *Integritätstest*-Komponente eine Veränderung des Objektcodes, so wird eine weitere Komponente benachrichtigt. Diese wird als *Fehlerbehebungs*-Komponente bezeichnet und implementiert je nach Anwendungsgebiet eine Strategie zur Fehlerbeseitigung. Die einfachste Strategie wäre, die Komponente neu zu instanziieren. Dies ist aber nur bei zustandslosen Komponenten möglich. Für zustandsbasierte Komponenten ist ein fehlerbehebender Prüfcode geeignet, welcher jedoch mit einer größeren Anzahl an redundanten Bits verbunden ist.

Implementierung
Positive Anwendungsbedingung:
$A1^{+e}$: ErfülltZuverlässigkeitsAnforderungenNicht (x_1)$\vee$ ErfülltVerfügbarkeitsAnforderungenNicht (x_1) $\vee$ ErfülltEchtzeitAnforderungenNicht (x_1) $\vee$ ErfülltSicherheitsAnforderungenNicht (x_1)
Negative Anwendungsbedingung:
$A2^{-e}$: $\exists$ it$\in$ Anwendungsarchitektur: (it.Typ=Integritätstest) $\wedge$Verschaltet(x_1,it)
$A3^{-o}$: ErfülltÖkonomischeAnforderungenNicht(System)



Regeln zur Variableninstanziierung:
$x_2 \mid x_1$

Konsequenzen

Eigenschaft	Auswirkung	Begründung
Sicherheit	+	Die Sicherheit des Systems kann sich bei der Operatoranwendung erhöhen, wenn ein Fehlverhalten auf Basis der Veränderung des Daten- und Codesegmentes zu einer Gefährdung führt. Diese Fehlverhalten können nach der Operatoranwendung erkannt und geeignet behandelt werden.
Zuverlässigkeit	+	Die Zuverlässigkeit des Systems kann sich erhöhen, da zufällige Fehler entdeckt und behoben werden können.
Wartbarkeit	o	
Verfügbarkeit	o	
Echtzeitfähigkeit	-	Das Laufzeitverhalten wird durch die Überprüfung der Integrität in der *Integritätstest*-Komponente negativ beeinflusst. Zusätzlich wird durch das Hinzufügen von aktiven Softwarekomponenten auch das Schedulingverhalten negativ beeinflusst.
Kosten	-	Die Kosten für die Systemerstellung erhöhen sich, da für die zusätzlichen Komponenten und die Integration der Prüfbits Entwicklungskosten anfallen.

Referenzen auf verwandte Muster und Varianten

Keine bekannten Muster

D.10 Monitor

Kontext und Problembeschreibung

Das Problem ist ein Architekturelement, welches die Sicherheitsanforderungen nicht einhält. Dieses Architekturelement beeinflusst direkt sicherheitskritische Systemfunktionen, so dass ein Fehlverhalten der Komponente oder fehlerhafte Informationen der Sensoren stets zu einer Gefährdung der Systemumwelt führt.

Lösung

Zur Überwachung der Systemumwelt und des sicherheitskritischen Architekturelementes führt das Monitor-Muster mit der *Monitor*-Komponente und den *Monitor*-Sensoren zusätzliche Architekturelemente in die Architektur ein. Die *Monitor*-Sensoren dienen dabei als redundante Sensoren, welche relevante Informationen über den Zustand der Systemumwelt liefern. Auf Basis dieser Informationen überprüft die *Monitor*-Komponente das Verhalten des beobachteten Architekturelementes. Dabei werden nicht nur Fehlverhalten des sicherheitskritischen Architekturelementes erkannt, sondern auch Fehlverhalten, welche auf Basis von fehlerhaften Informationen über die Systemumwelt entstehen. Für die Realisierung der *Monitor*-Komponente werden die für das System spezifizierten Gefährdungen verwendet. Die Erfüllung der zu den Gefährdungen führenden Bedingungen wird in der *Monitor*-Komponente zyklisch geprüft. Wird dabei erkannt, dass eine Gefährdung eingetreten ist, so wird eine *Fehlerbehebungs*-Komponente benachrichtigt.

Als Beispiel für die Anwendung des Monitor-Musters soll ein Bahnübergang dienen. Wie bereits beschrieben, besteht die Gefährdung in einem Zug im Bahnübergangsabschnitt und offenen Schranken. Bei der Anwendung des Musters werden demnach zwei zusätzliche redundante Sensoren verwendet. Der erste Sensor überprüft, ob sich ein Zug im Bahnübergangsabschnitt befindet, und der zweite Sensor überprüft den Status der Schranke. Die *Monitor*-Komponente überprüft die beiden Sensoren und kann somit eine vom System ausgehende Gefährdung erkennen.

Implementierung

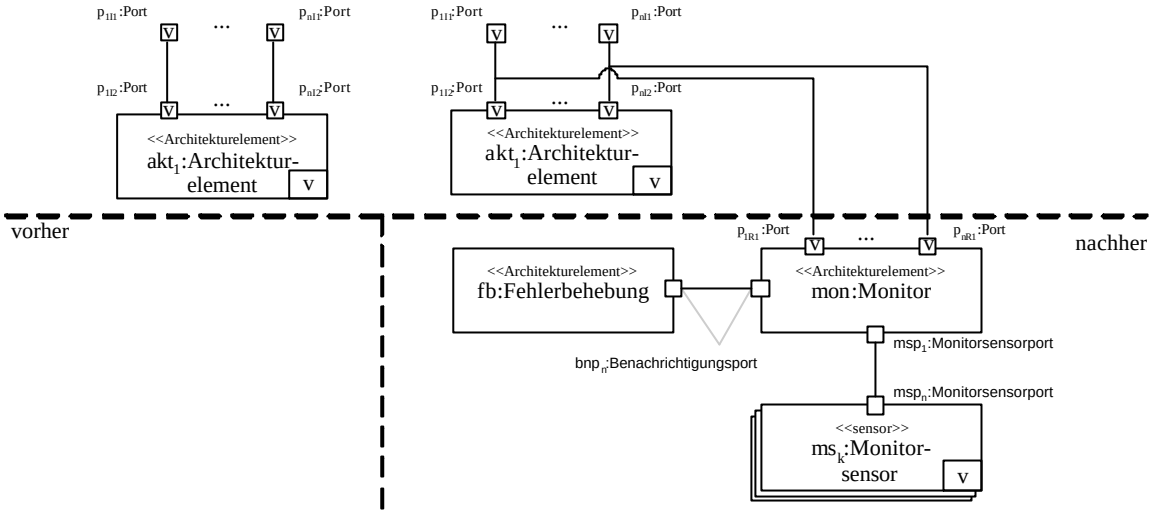
Positive Anwendungsbedingung:

$A1^{+e}$: ErfülltZuverlässigkeitsAnforderungenNicht (x_1)$\vee$ ErfülltSicherheitsAnforderungenNicht (x_1)

Negative Anwendungsbedingung:

$A2^{-e}$: $\exists$ mon$\in$ Anwendungsarchitektur: (mon.Typ=Monitor) $\wedge$Verschaltet(x_1,mon)

$A3^{-o}$: ErfülltÖkonomischeAnforderungenNicht(System)



Konsequenzen

Eigenschaft	Auswirkung	Begründung
Sicherheit	+	Die Sicherheit des Systems kann sich durch die Operatoranwendung erhöhen, da vom System ausgehende Gefährdungen erkannt und geeignet behandelt werden können
Zuverlässigkeit	o	
Wartbarkeit	o	
Verfügbarkeit	o	
Echtzeitfähigkeit	-	Das Laufzeitverhalten des beobachteten Architekturelementes wird durch die Anwendung des Musters primär nicht geändert. Durch das Hinzufügen von zusätzlichen aktiven Softwarekomponenten wird jedoch das Schedulingverhalten negativ beeinflusst.
Kosten	-	Die Kosten für die Systemerstellung erhöhen sich, da für die zusätzlichen Komponenten Entwicklungskosten anfallen.

Referenzen auf verwandte Muster und Varianten

Actuation Monitor /Douglass 99,02/, /Lala, Harper 94/

D.11 Hardwareelementaustausch

Kontext und Problembeschreibung

Qualitätseigenschaften, wie z. B. Zuverlässigkeit, Verfügbarkeit und Echtzeitfähigkeit von Softwareelementen werden von dem ausführenden Hardwareelement beeinflusst. Als Beispiel für diesen Einfluss ist die Zuverlässigkeit einer Softwarekomponente von der Zuverlässigkeit der ausführenden Hardwareplattform und die Echtzeitfähigkeit von der Ausführungsgeschwindigkeit des Prozessors abhängig. Daher führt ein Hardwareelement mit unzureichenden Qualitätseigenschaften zu einer Nichterfüllung der Qualitätsanforderungen von ein oder mehreren Softwareelementen.

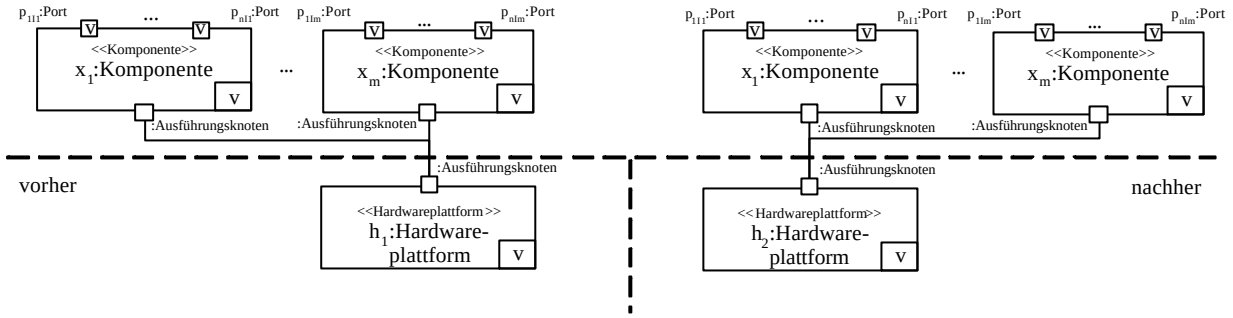
Lösung

Zur Lösung des Problems wird das ausführende Hardwareelement durch ein neues Hardwareelement ausgetauscht. Dieses neue Hardwareelement besitzt einen anderen Typ als das Vorgängerhardwareelement. Zur Verbesserung der Qualität sollten die relevanten Qualitätseigenschaften des Hardwareelementtyps besser sein als die seines Vorgängers. Alle Softwareelemente, welche auf dem alten Hardwareelement ausgeführt wurden, werden nach der Musteranwendung auf dem neuen Hardwareelement ausgeführt. Das alte Hardwareelement kann somit aus der Architektur entfernt werden.

Implementierung

Positive Anwendungsbedingung:

$A1^{+e}$: ErfülltZuverlässigkeitsAnforderungenNicht $(x_1) \vee$ ErfülltVerfügbarkeitsAnforderungenNicht (x_1) $\vee$ ErfülltEchtzeitAnforderungenNicht $(x_1) \vee$ ErfülltSicherheitsAnforderungenNicht (x_1)



Negative Anwendungsbedingung:

$A3^{-o}$: ErfülltÖkonomischeAnforderungenNicht(System)

Konsequenzen

Eigenschaft	Auswirkung	Begründung
Sicherheit	+/o	Die Zuverlässigkeit der Softwareelemente kann sich erhöhen, wenn sich die Qualität des Hardwareelementes erhöht
Zuverlässigkeit	+/o	Die Zuverlässigkeit der Softwareelemente kann sich erhöhen, wenn sich die Zuverlässigkeit des Hardwareelementes erhöht.
Wartbarkeit	+/o	Die Wartbarkeit der Softwareelemente kann sich erhöhen, wenn sich die Wartbarkeit des Hardwareelementes erhöht.
Verfügbarkeit	+/o	Die Verfügbarkeit der Softwareelemente kann sich erhöhen, wenn sich die Verfügbarkeit des Hardwareelementes erhöht.
Echtzeitfähigkeit	+/o	Die Echtzeitfähigkeit der Softwareelemente kann sich verbessern, wenn sich die Ausführungs- oder Übertragungsgeschwindigkeit des Hardwareelementes verbessert.
Kosten	-	Bessere Hardwareelemente sind zumeist mit erhöhten Kosten verbunden.

Referenzen auf verwandte Muster und Varianten

Keine bekannten Muster

D.12 Neuzuweisung des Hardwareelementes

Kontext und Problembeschreibung

Beim Muster Hardwareelementaustausch war ein ungeeignetes Hardwareelement Ursache für Qualitätsprobleme bei Softwareelementen, welche auf diesem Hardwareelement ausgeführt wurden. Bei diesem Muster liegt das Problem in der Ausführungsbeziehung eines einzelnen Softwareelementes. Dabei existieren zwei grundlegende Teilprobleme. Im ersten Teilproblem führt die Ausführung einer Softwarekomponente auf einer Hardwareplattform zu einem Schedulingproblem, da dieses Softwareelement eine zu große Ausführungszeit (WCET) besitzt. Das zweite Teilproblem sind Fehlverhalten mit gemeinsamer Ursache (common cause failure). Dabei ist ein Fehlverhalten des Hardwareelementes Ursache für die Fehlverhalten von zwei oder mehreren Softwareelementen. Diese fallen nun nicht mehr unabhängig voneinander aus, und somit ist die Wahrscheinlichkeit für einen Ausfall höher.

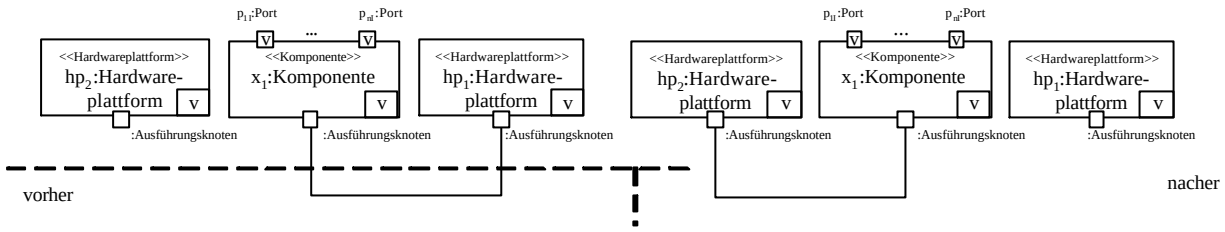
Lösung

Zur Lösung des Problems wird das Softwareelement mit der problematischen Ausführungsverschaltung auf einem anderen Hardwareelement ausgeführt. Dieses Hardwareelement sollte geeignet ausgewählt werden. Zum Beispiel sollte eine Softwarekomponente mit einer zu hohen Ausführungszeit (WCET) auf einer Hardwareplattform ausgeführt werden, welche bisher noch nicht ausgelastet ist.

Implementierung

Positive Anwendungsbedingung:

$A1^{+e}$: ErfülltZuverlässigkeitsAnforderungenNicht (x_1)$\vee$ ErfülltVerfügbarkeitsAnforderungenNicht (x_1) $\vee$ ErfülltEchtzeitAnforderungenNicht (x_1) $\vee$ ErfülltSicherheitsAnforderungenNicht (x_1)



Konsequenzen

Eigenschaft	Auswirkung	Begründung
Sicherheit	+/o	Die Sicherheit kann verbessert werden, wenn durch die Musteranwendung Fehlverhalten mit gemeinsamer Ursache reduziert werden oder zeitkritische Anforderungen besser erfüllt werden
Zuverlässigkeit	o	
Wartbarkeit	o	
Verfügbarkeit	o	
Echtzeitfähigkeit	+	Die Echtzeitfähigkeit kann sich erhöhen, wenn durch die Musteranwendung Schedulingprobleme behoben werden.
Kosten	o	

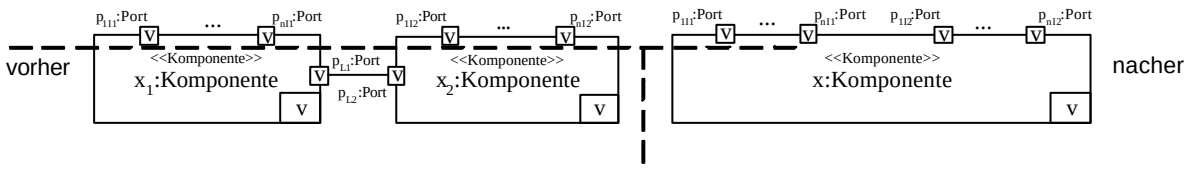
Referenzen auf verwandte Muster und Varianten
Homogene-Redundanz-Muster /Douglass 99/

D.13 Vereinigung von Komponenten

Kontext und Problembeschreibung
Werden zu viele kleine aktive Softwarekomponenten auf einer Hardwareplattform abgewickelt, so wirkt sich dies negativ auf das Schedulingverhalten und die Echtzeitfähigkeit aus. Begründet wird dies durch die Zeit, welche bei jedem Prozesswechsel für die Wiederherstellung der Register und des Stacks sowie für das Einlagern der benutzten Daten in den Cache benötigt wird.

Lösung
Zur Lösung des Problems werden jeweils zwei aktive Softwarekomponenten zu einer komplexeren zusammengefasst. Dadurch wird die Anzahl der aktiven Softwarekomponenten auf der Hardwareplattform und damit der Overhead für die Prozesswechsel und Interprozesskommunikation reduziert. Als Ergebnis verbessert sich das Schedulingverhalten und die Echtzeitfähigkeit. Zusätzlich wird die Kopplung zwischen den Softwarekomponenten reduziert.
Durch die Verschmelzung von Softwarekomponenten wird jedoch die Kapselung der einzelnen Komponenten aufgehoben. Zudem verschlechtert sich die Bindung innerhalb der zusammengefassten Softwarekomponenten, wenn die beiden ursprünglichen Softwarekomponenten unterschiedliche Anforderungen realisieren. Daher ist eine Verschmelzung nur bei Softwarekomponenten mit ähnlichen Aufgaben realisierbar

Implementierung
Positive Anwendungsbedingung:
$A1^{+e}$: ErfülltEchtzeitAnforderungenNicht (x_1)



Konsequenzen

Eigenschaft	Auswirkung	Begründung
Sicherheit	+/-	Bei der Anwendung des Musters ist sowohl eine positive als auch negative Auswirkung auf die Sicherheit möglich. Die positiven Auswirkungen beruhen auf der verringerten Kopplung und der verbesserten Echtzeitfähigkeit durch die Musteranwendung. Die

		negativen Auswirkungen werden durch eine größere Fehlerwahrscheinlichkeit bei der Entwicklung der komplexeren Softwarekomponente begründet.
Zuverlässigkeit	o	
Wartbarkeit	o	
Verfügbarkeit	o	
Echtzeitfähigkeit	+	Bei der Anwendung des Musters werden Schedulingprobleme reduziert, da die Zeiten für die Prozesswechsel und die Interprozesskommunikation reduziert werden.
Kosten	+/-	Die Anwendung des Transformationsoperators kann sich auf die Entwicklungskosten positiv auswirken, da die Kopplungseigenschaften positiv verbessert werden und dadurch mit einem geringen Kommunikationsaufwand bei der Systementwicklung zu rechnen ist. Die Anwendung kann jedoch auch einen negativen Effekt auf die Entwicklungskosten haben, da die resultierende Komponente komplexer und somit schwerer verständlich ist. Des Weiteren ist eine parallele Entwicklung wie bei den ursprünglichen Komponenten nicht mehr möglich.

Referenzen auf verwandte Muster und Varianten

Keine bekannten Muster

Literaturliste

/Abadi, Lamport 90/
Abadi M., Lamport L., *Composing specifications*, Technical report, DEC SRC, 130, Lytton Av, Palo Alto, Ca 94301, October 1990

/Abadi, Lamport 91/
Abadi M., Lamport L., *The Existence of Refinement Mappings*, in: IEEE Transactions on Software Engineering 82(2), 1991, pp. 253-284

/Abadi, Lamport 95/
Abadi M., Lamport L., *Conjoining Specifications*, in: ACM Transactions on Programming Languages and Systems, Vol. 17, No. 3, May, 1995, pp. 507-534

/Abd-Allah et al. 95/
Gacek C., Abd-Allah A., Clark B., Boehm B.. *On the Definition of Software Architecture*, ICSE 17 Software Architecture Workshop, April 1995

/Abowd et al. 95/
Abowd G. D., Allen R., Garlan D., *Formalizing Style to Understand Descriptions of Software Architecture*, in ACM Transactions on Software Engineering and Methodology, 4(4), October,1995, pp. 319-364

/Agha, Kim 99/
Agha G., Kim W., *Actors: A Unifying Model for Parallel and Distributed Computing*, in: Journal of Systems Architecture, June '98, EuroMicro Society/Elsevier, Vol. 45, 1999, pp. 1263-1277

/Agrawal et al. 02/
Agrawal A., Levendovszky T., Sprinkle J., Shi F., Karsai G., *Generative Programming via Graph Transformations in the Model-Driven Architecture*, OOPSLA, Workshop on Generative Techniques in the Context of Model Driven Architecture, Seattle, WA, November 5, 2002

/Agrawal et al. 03/
Agrawal A., Karsai G., Shi F., *Graph Transformations on Domain-Specific Models*, in International Journal on Software and Systems Modeling, (accepted), 2003

/Aksit, Bergmans 92/
Aksit M., Bergmans L., *Obstacles in Object-Oriented Software Development*, Proceedings OOPSLA '92, ACM SIGPPLAN Notices, Vol. 27, No. 10, pp. 341-358, October 1992

/Alexander 79/
Alexander C., Ishikawa S., Silverstein M., *A Pattern Language*, New York City: Oxford University Press, 1979

/Allen, Garlan 97/
Allen R., Garlan D., *A Formal Basis for Architectural Connection*, in: ACM Transactions on Software Engineering Methodology TOSEM, Vol. 6, No. 3, July, pp. 213-249, 1997

/Alur, Henzinger 89/
Alur R., Henzinger T.A., *A really temporal logic*, in: Proceeding of the 30th Annual Symposium on Foundations of Computer Science, pp. 164-169, IEEE Computer Society Press, 1989

/Alur, Henzinger 93/
Alur R., Henzinger T.A., *Real-time logics: Complexity and expressiveness*, Communications of the ACM, 26(11), 1993, pp. 390-401

/Alur, Dill 94/
Alur R., Dill D.L., *A Theory of Timed Automata*, in: Theoretical Computer Science Vol. 126, No. 2, April 1994, pp. 183-236

/Alur et al. 96/
Alur R., Feder T., Henzinger T., *The benefits of relaxing punctuality*, Journal of ACM 43(1) (1996), pp. 116-146

/Alur et al. 98/
Alur R., Henzinger T. A., Mang F. Y. C., Qadeer S., Rajamani S. K., Tasiran S., *Mocha Modularity in Model Checking,* in: Computer Aided Verication, Proc. 10th Int. Conference, volume 1427 of Lecture Notes in Computer Science. Springer Verlag, 1998, pp. 521-525

/Alur 98/
Alur, R., *Timed Automata*, in: Nato ASI Summer School on Verification of Digital and Hybrid Systems, 1998

/Alur, Henzinger 99/
Alur R., Henzinger T. A., *Reactive modules,* in: Formal Methods in System Design, 15, 1999, pp.7-48

/Andrews, Schneider 83/
Andrews G. R., Schneider F. B., *Concepts and Notations for Concurrent Programming*, in: ACM Computing Surveys, Vol. 15, No. 1, March, 1983, pp. 3-43

/Andries et al. 99/
Andries M., Engels G., Habel A., Hoffmann B., Kreowski H.- J., Kuske S., Plump D., Schürr A., Taentzer G., *Graph transformation for specification and programming*, in: *Science of Computer Programming, Vol.* 34, 1999, pp. 1-54

/Antoniol et al. 95/
Antoniol G., Lokan C., Caldiera G., Fiutem R., *A Function Point-Like Measure for Object-Oriented Software*, in: Empirical Software Engineering an International Journal, Vol. 4, 1999, pp. 263-287

/Assmann 00/
Assmann U., *Graph Rewrite Systems for Program Optimization*, in: ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 22, No. 4, New York: ACM Press 2000

/Asundi et al. 01/
Asundi J., Kazman R., Klein M., *Using Economic Considerations to Choose Among Architecture Design Alternatives,* Technical Report CMU/SEI-2001-TR035, Dezember 2001

/Avizienis 85/
Avizienis A., *The N-Version Approach to Fault-Tolerant Software*, in: IEEE Transactions on Software Engineering, vol. SE-11, no. 12, Dec. 1985, pp.1491-1501

/Avizienis et al. 01/
Avizienis A., Laprie J.-C., Randell B., *Fundamental Concepts of Dependability*, Research Report N01145, LAAS-CNRS, April 2001

/Baker, Shaw 89/
Baker T., Shaw A., *The cyclic executive model and Ada*. The Journal of Real-Time Systems, Vol. 1, September 1989, pp. 7-25

/Balzert 98/
Balzert H., *Lehrbuch der Softwaretechnik II*, (1.Aufl.) Spectrum Akademischer Verlag, Heidelberg, 1998

/Balzert 01/
Balzert H., *Lehrbuch der Softwaretechnik I*, (2.Aufl.) Spectrum Akademischer Verlag, Heidelberg, 2001

/Barcio et al. 95/
Barcio B. T., Ramaswamy S., Barber K. S., *OARS: An Object Oriented Architecture for Reactive Systems*, 1995 IEEE International Conference on Robotics and Automation, May 1995

/Bardohl et al. 99/
Bardohl R., Minas M., Schürr A., Taentzer G., *Application of graph transformation to visual languages*, pp. 105-180, in: Ehrig H., Engels G., Kreowski H.- J., Rozenberg G. (eds), Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools, Singapore: World Scientific Publisher 1999

/Bardohl et al. 03/
Bardohl R., Ermel C., Weinhold I., *GenGED - A visual definition tool for visual modeling environments,* in Proc. Application of Graph Transformations with Industrial Relevance (AGTIVE'03), Sept./Oct., 2003, Charlottesville/Virgina, USA

/Battiston et al. 96/
Battiston E., Chizzoni A., De Cindio F., *Modeling a cooperative environment with CLOWN.* in: G. Agha, F. De Cindio, A. Yonezawa (editors), Proceedings of the second international workshop on Object-

Oriented Programming and Models of Concurrency within the 16th International Conference on Application and Theory of Petri Nets, Osaka, Japan, 1996, pp. 12–24

/Bass et al. 97a/
Bass L., Clements P., Cohen S., Northtop L., Withey J. Product Line Practice Workshop Report, Technical Report, 1997

/Bass et al. 97b/
Bass L., Clements P., Chastek G., Cohen S., Northrop L., Withey J., 2nd Product Line Practice Workshop Report (CMU/SEI-98-TR-015, ESC-TR-98-015) Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997

/Bass et al. 98/
Bass L., Clements P., Kazman, R., *Software Architecture in Practice*, Addison-Wesley 1998

/Bauderon, Jacquet 01/
Bauderon M., Jacquet H., *Node Rewriting in Graphs and Hypergraphs: A Categorial Framework*, in: Theoretical Computer Science, 2001, Vol. 266, Issue 1-2, September 2001, pp. 463-487

/Bengtsson, Bosch 99/
Bengtsson P. O., Bosch J., *Architecture Level Prediction of Software Maintenance*, in: Proceedings of the Third European Conf. Software Maintenance and Reengineering, 1999, pp. 139–147

/Benlarbi, Melo 99/
Benlarbi S., Melo W. L., *Polymorpism Measure for Early Risk Prediction*, Proc. of the 21th Int. Conf. on Software Engineering, Los Angeles, CA. IEEE Press, May 1999

/Berge 89/
Berge C., *Hypergraphs: Combinatorics of Finite Sets*, North-Holland 1989

/Berry 98/
Berry D. M., *The Safety Requirements Engineering Dilemma*, Proceedings IWSSD 98, 9th International Workshop on Software Specification and Design, Isobe, IEEE CS Press, April 1998

/Binns et al. 96/
Binns P., Englehart M., Jackson M., Vestal S., *Domain-Specific Software Architectures for Guidance, Navigation and Control*, in: *International Journal of Software Engineering and Knowledge Engineering*, Volume 6, Number 2, 1996

/Biberstein et al. 1997/
Biberstein O., Buchs D., Guelfi N., COOPN/2: *A concurrent object-oriented formalism.* In: H. Bowman J. Derrick (editors), Proc. Second IFIP Conf. On Formal Methods for Open Object-Based Distributed Systems (FMOODS), London: Chapman and Hall 1997, pp. 57–72

/Bierman, Kang 98/
Bieman J. M., Kang B.-K., *Measuring Design-Level Cohesion*, in: IEEE Transactions on Software Engineering, VOL. 24, NO. 2, pp. 111-124, 1998

/Binder 00/
Binder R. V., *Testing object-oriented systems*, Addison-Wesley Longman, 2000

/Birolini 99/
Birolini A., *Reliability engineering: theory and practice* (third ed.), New York, Springer, 1999

/Bitsch 01/
Bitsch F., *Safety Patterns - The Key to Formal Specification of Safety Requirements*, in: Proceedings of 20th International Conference, SAFECOMP 2001 - Computer Safety Reliability and Security, Budapest, September 2001, LNCS 2187, Berlin, Heidelberg: Springer 2001, pp. 176-189

/Boehm 81/
Boehm B.W., *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981

/Boehm 88/
Boehm B.W., *A Spiral Model of Software Development and Enhancement*, in: IEEE Computer, May 1988, pp. 62-72

/Boehm et al. 95/
Boehm B.W., Clark B. K., Horowitz E., Madachy R., Selby R.W., Westland C., *Cost Models for Future Software Processes: COCOMO 2.0*, Annals of Software Engineering, 1995, v.1, pp. 57-94

/Boehm 95/
Boehm B.W., *Engineering Context for Software architecture*, First International Workshop on Architecture of Software System, Seattle, 1995

/Bondavalli, Simoncini 90/
Bondavalli A., Simoncini L., *Failure Classification with Respect to Detection*, in: Predictably Dependable Computing Systems, Task B, Vol. 2, May 1990

/Booch 91/
Booch G., *Object-Oriented Design with Applications*, The Benjamin/Cummings Publishing Company, Inc, 1991

/Booch 94/
Booch G., *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings, Redwood City, California, 1994

/Booch et al. 99/
Booch G., Rumbaugh J., Jacobson I., *The Unified Modeling Language User Guide*. Addison Wesley, 1999

/Bosch, Molin 99/
Bosch J., Molin P., *Software Architecture Design: Evaluation and Transformation*, IEEE Engineering of Computer Based Systems Symposium (ECBS99), December 1999

/Bosch, Svahnberg 99/
Bosch J., Svahnberg M., *Characterizing Evolution in Product-Line Architectures*, in: Proceedings of the IASTED 3rd International Conference on Software Engineering and Applications, October 1999, pp. 92-97

/Bosch 00/
Bosch J., *Design and Use of Software Architectures,* Addison Wesley, Reading, MA, 2000

/Bosch 01/
Bosch J., *Software Product Lines: Organizational Alternatives*, in: Proceedings of the 23. International Conference on Software Engineering, IEEE Computer Society Press, pp 91--101, 2001

/Bozga et al. 98/
Bozga M., Daws C., Maler O., Olivero A., Tripakis S., Yovine S., *Kronos: a modelchecking tool for real-time systems*, in: Proc. of the 10th Conference on Computer-Aided Verification, Vancouver, Canada, 28 June - 2 July 1998, Springer 1998

/Braband 98/
Braband J., *RAMS-Management nach CENELEC*, Signal & Draht 12/98, pp. 20-24, 1998

/Braband et al. 98/
Braband J., Heilmann A. und Peters H., *Bahnübergangs-Sicherungstechnik von Siemens: Sicherheitsanalyse nach CENELEC*, Signal & Draht 7/98

/Braband, Lennartz 99/
Braband J., Lennartz K., *A Systematic Process for the Definition of Safety Targets for Railway Signalling Applications*, Signal & Draht 9/99, pp. 5-10, 1999

/Braband, Lennartz 00/
Braband J., Lennartz K., *On the Estimation of Individual Risk*, Signal & Draht 11/00, 2000, pp 37-39

/Braband 01/
Braband J., *Lessons Learned from Risk Assessment Studies in Railway Signaling*, in: Proc. Kolloquium: Moderne Sicheranalysen Theorie und Praxis, Braunschweig, Oktober 2001

/Briand et al. 00/
Briand L., Wüst J., Daly J., Porter V., *Exploring the Relationship between Design Measures and Software Quality in Object-Oriented Systems*, in: Journal of Systems and Software, Ausgabe 51, 2000, pp. 245-273

/Brito et al. 02/
Brito I., Moreira A., Araújo J., *A Requirements Model for Quality Attributes*, 1st International Conference on Aspect-Oriented Software Development, University of Twente, Enschede, Holand, April 22-26, 2002

/Brookes et al. 84/
Brookes S.D., Hoare C.A.R., Roscoe A.W., *A Theory of Communicating Sequential Processes*, Journal of the ACM 31(3), pp. 560-599, 1984

/Brown, 1991/
Brown P. G., *QFD: Echoing the Voice of the Customer*, AT&T Technical Journal, March/April 1991, pp. 18-32
/Broy 97/
Broy M., *Requirements Engineering for Embedded Systems*, in: Proc. of FemSys 97, 1997
/Buchs, Guelfi 00/
Buchs D., Guelfi N., *A Formal Specification Framework for Object-Oriented Distributed Systems*, IEEE Transactions on Software Engineering, vol. 26, no. 7, July 2000, pp. 635-652
/Busatto, Hoffmann 01/
Busatto G., Hoffmann B., *Comparing Notions of Hierarchical Graph Transformation*, in: Proceedings of the Satellite Workshop to ICALP 2001, on Graph Transformation and Visual Modelling Techniques, July 2001, Heraclion, Crete, Greece, Electronic Notes in Theoretical Computer Science, Vol. 50, No. 3
/Burns, Pitblado 1993/
Burns D.J., Pitblado R.M., *A Modified HAZOP Methodology for Safety Critical System Assessment*, in: Directions in Safety-critical Systems: Proceedings of the Safetycritical Systems Symposium, Bristol, Ed. Redmill F. and Anderson T., 1993, pp. 232-245
/Buschmann et al. 96/
Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M., *Pattern-Oriented Software Architecture - A System of Patterns*, John Wiley & Sons 1996
/Chakrabarti et al. 02/
Chakrabarti A., de Alfaro L., Henzinger T. A., Jurdzinski M., Mang F. Y. C., *Interface Compatibility Checking for Software Modules*, in: Proceedings of the 14th International Conference on Computer-Aided Verification (CAV), Lecture Notes in Computer Science 2404, Springer-Verlag, 2002, pp. 428-441
/Chaki et al. 02/
Chaki S., Rajamani S. K., Rehof J., *Types as models: Model checking message-passing programs*, in: Principles of Programming Languages (POPL), January 2002
/Chidamber, Kemerer 94/
Chidamber S.R., Kemerer C.F., *A Metrics Suite for Object-Oriented Design*, in: IEEE Transactions on Software Engineering, June 1994, pp. 476-493
/Chung, Nixon 95/
Chung L., Nixon B. A., *Dealing with Non-Functional Requirements: Three Experimental Studies of a Process-Oriented Approach*, in: Proc., IEEE 17th International Conference on Software Engineering, Seattle, April 24-28, 1995, pp. 25-37
/Chung et al. 99/
Chung L., Nixon B.A., Yu E., Mylopoulos J., *Non-Functional Requirements in Software Engineering*, Kluwer Academic Publishers, ISBN 0-7923-8666-3. October 1999
/Clements et al. 95/
Clements P., Bass L., Kazman R., Abowd G., *Predicting Software Quality by Architectural Evaluation*, *Proceedings of the Fifth International Conference on Software Quality*, (Austin, TX), October 1995, pp. 485-497
/Clements et al. 85/
Clements P., Parnas D. & Weiss D., *The Modular Structure of Complex Systems*, IEEE Transactions on Software Engineering SE-11, 1 (1985): 259-266
/Clements 96/
Clements P., *A Survey of Architectural Description Languages*, Proceedings of the Eighth International Workshop on Software Specification and Design. Paderborn, Germany, March 1996.
/Coad, Yourdon 91/
Coad P., Yourdon E., *Object-Oriented Analysis*, Prentice Hall, Englewood Cliffs, New Jersey, 1991
/Coad, Yourdon 94/
Coad P., Yourdon E., *Objekt-Oriented Design*, Prentice Hall, Englewood Cliffs, New Jersey, 1994
/Cockburn 97/
Cockburn A., *Structuring Use Cases with Goals*, in: Journal of Object-Oriented Programming, Sep-Oct, Nov-Dec 1997

/Corradini et al. 97/
Corradini A., Montanari U., Rossi F., Ehrig H., Heckel R., Löwe M., *Algebraic Approaches to Graph Transformation - Part I: Basic Concepts and Double Pushout Approach*, in: Rozenberg G. (ed.), *Handbook of Graph Grammars and Computing by Graph Transformation*, Vol. 1: Foundations, Singapore: World Scientific Publisher, 1997, pp. 163-245

/Cristel, Kang 92/
Cristel M. G., Kang K. C., *Issues in Requirement Elicitation*, Technical report, CMU/SEI-92-TR-12, ESC-TR-92-012, September 1992

/Czarnecki 99/
Czarnecki C., *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*, PhD Thesis, Technical University of Ilmenau, 1999

/Dardenne et al. 91/
Dardenne A., Fickas S., van Lamsweerde S. A., *Goal-directed Concept Acquisition in Requirements Elicitation*, Proceedings 6th International Workshop on Software Specification and Design, Como, Italy, 1991, pp. 14-21

/De Alfaro, Henzinger 01a/
de Alfaro L., Henzinger T. A., *Interface Theories for Component-based Design*, in: Proceedings of the First International Workshop on Embedded Software (EMSOFT), Lecture Notes in Computer Science 2211, Springer-Verlag, 2001, pp. 148-165

/De Alfaro, Henzinger 01b/
de Alfaro L., Henzinger T. A., *Interface automata*, in: 9th Symp. Foundations of Software Engineering, ACM Press 2001

/de Alfaro et al. 02/
de Alfaro L., Henzinger T. A., Stoelinga M., *Timed Interfaces*, in: Proceedings of the Second International Workshop on Embedded Software (EMSOFT) , Lecture Notes in Computer Science, Springer-Verlag 2002

/De Lemos et al. 95/
De Lemos, Rogério, Saeed, Amer, Anderson, Tom, *Analyzing Safety Requirements for Process-Control Systems*, in: IEEE Software 12, 3, 1995, pp. 42-53

/Def Std 00-58/
Ministry of Defence, *HAZOP studies on systems containing programmable electronics*. Glasgow, U.K.: Ministry of Defence, Directorate of Standardisation, Defence Standard 00-58, parts 1 and 2

/Demeyer 03/
Demeyer S., Ducasse S., Nierstrasz O., *Object-Oriented Reengineering Patterns*, Morgan Kaufmann 2003

/DIN 55350-11/
DIN 55350, Teil 11, *Begriffe der Qualitätssicherung und Statistik*; Grundbegriffe der Qualitätssicherung;

/DIN V VDE 0801/
DIN V VDE 0801 (Vornorm) *Grundsätze für Rechner in Systemen mit Sicherheitsaufgaben* 1990 Ergänzung durch Anhang A1 (DIN V VDE 0801/A1)vom Oktober 1994

/DIN 25424/
DIN 25424, *Fehlerbaumanalyse: Teil 1 (09/81): Methode und Bildzeichen. Teil 2 (04/90): Handrechenverfahren zur Auswertung eines Fehlerbaumes.* Beuth Verlag, Berlin 1981/1990

/DIN 25448/
DIN e. V. (Hrsg.), *DIN 25448: Ausfalleffektanalyse (Fehler-Möglichkeits- und -Einfluß-Analyse)*, Beuth, Berlin, Mai 1990

/DIN ISO 9126/
ISO/IEC *9126-Information Technology, Software Product Evaluation, Quality, Characteristics and Guidelines for their Use*

/Dijkstra 68/
Dijkstra E.W., *The Structure of the `THE' Multiprogramming System.* Communications of the ACM 11, 5 (May 1968), pp. 341-346

/Dijkstra 75/
Dijkstra E. W., *Guarded Commands, Nondeterminacy, and Formal Derivation of Programs,* Communications of the ACM, Vol. 18, No. 8, pp. 453-457, 1975

/Douglass 98/
Douglas B. P., *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Addison-Wesley, Massachusetts, 1998

/Douglas 99/
Douglas B. P., *Doing Hard Time*, Addison Wesley, Reading, Massachusetts, 1999

/Douglas 02/
Douglas B. P., *Real Time Design Patterns*, Addison Wesley Reading, Massachusetts, 2002

/Drewes et al. 97/
Drewes F., Habel A., Kreowski H.- J., *Hyperedge replacement graph grammars*, in: Rozenberg G. (ed.), *Handbook of Graph Grammars and Computing by Graph Transformation*, Vol. 1: Foundations, Singapore: World Scientific Publisher 1997

/Drewes et al. 02/
Drewes F., Hoffmann B., Plump D., *Hierarchical graph transformation*, in: *Journal of Computer and System Sciences, Vol.* 64, No. 2, pp. 249-283,2002

/Dunn 03/
Dunn W. R., *Designing Safety-Critical Computer Systems*. In: IEEE Computer 36(11), 2003, pp. 40-46

/Easterbrook, Nuseibeh 96/
Easterbrook S., Nuseibeh B., *Using ViewPoints for Inconsistency Management,* in: Software Engineering Journal, 11(1): pp. 31-43, BCS/IEE Press, January 1996

/Ehrig et al. 88/
Ehrig H., Boehm P., Hummert U., Löwe M., *Distributed parallelism of graph transformation*, in: *13th Int. Workshop on Graph Theoretic Concepts in Computer Science, LNCS 314*, 1988, Berlin: Springer, pp. 1-19

/Ehrig et al. 91/
Ehrig H., Habel A., Kreowski H.-J., Parisi-Presicce F., *Parallelism and Concurrency in High Level Replacement Systems*, in: *Math. Struc. in Comp. Science*, Vol. 1, 1991, pp. 361-404

/Ehrig et al. 97/
Ehrig H., Heckel R., Korff M., Löwe M., Ribeiro L., Annika Wagner A., Corradini A., *Algebraic Approaches to Graph Transformation - Part II: Single Pushout Approach and Comparison with Double Pushout Approach*, in: Rozenberg G. (ed.), *Handbook of Graph Grammars and Computing by Graph Transformation*, Vol. 1: Foundations, Singapore: World Scientific Publisher, pp. 247-312, 1997

/Emerson 90/
Emerson E.A., *Temporal and modal logic*, in: J. van Leeuwen (ed.), Handbook Theor. Comp. Sci., Vol.B, pp. 997-1072, 1990

/EN 50126/
CENELEC, *Railway applications The specification and demonstration of dependability, reliability, availability, maintainability and safety (RAMS)*, European Committee for Electrotechnical Standardisation, Brussels, Standard EN 50126-29, November 1995

/EN 50128/
CENELEC, *Bahnanwendungen: Software für Eisenbahnsteuerungs- und Überwachungssysteme*; Deutsche Fassung EN 50128, 2001

/EN 50129/
CENELEC, *Bahnanwendungen: Sicherheitsrelevante elektronische Systeme für Signaltechnik*, Deutsche Fassung EN 50129, 1998

/Engelfriet, Rozenberg 97/
Engelfriet J., Rozenberg G., *Node replacement graph grammars*, in: Rozenberg G. (ed.), *Handbook of Graph Grammars and Computing by Graph Transformation*, Vol. 1: Foundations, Singapore: World Scientific Publisher, pp. 1-94, 1997

/Engels, Schürr 95/
Engels G., Schürr A., *Encapsulated Hierarchical Graphs, Graph Types and Meta Types*, Joint

Compugraph/*Semagraph Workshop on Graph Rewriting and Computation*, Electronic Notes in: Theoretical Computer Science, Vol. 2, Elsevier 1995

/Engels, Heckel 00/
Engels G., Heckel R., *Graph Transformation as a Conceptual and Formal Framework for System Modeling and Model Evolution*, pp. 127-150, ICALP 2000

/Ermel et al. 01/
Ermel C., Bardohl R., Padberg J., *Visual Design of Software Architecture and Evolution based on Graph Transformation*, in: Proc. Uniform Approaches to Graphical Process Specification Techniques UNIGRA'01, 2001

/Fahmy, Holt 00/
Fahmy H., Holt R. C., *Using Graph Rewriting to Specify Software Architectural Transformations*, in: Proceedings of Automated Software Engineering, Grenoble, France, September 2000

/Fenelon et al. 94/
Fenelon P., McDermid J.A., Nicholson M., Pumfrey D. J., *Towards Integrated Safety Analysis and Design*, in: ACM Applied Computing Review, August 1994

/Fenelon, McDermid 93/
Fenelon P., McDermid J.A., *An Integrated Toolset For Software Safety Analysis*, Journal of Systems and Software 21(3), pp. 279-290,1993

/Fenelon, Hebbron 94/
Fenelon P., Hebbron B. D., *Applying HAZOP to software engineering models*, in Risk Management And Critical Protective Systems: Proceedings of SARSS Altrincham, pp. 1/1--1/16, Oct. 1994

/Fernandez, Monzón 01/
Fernandez-Sanchez J. L., Monzón A., *Extending UML for Real-Time Component Based Architectures*. International conference on Software and Systems Engineering & their Applications (ICSSEA 2001), Paris, December 2001

/Fowler 96/
Fowler M., *Analysis Patterns : Reusable Object Models*, Addison-Wesley, 1996

/Fowler, Scott 97/
Fowler, M., Scott K., *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. AddisonWesley, Reading, Massachusetts, 2nd edition, 1999

/Fowler 99/
Fowler M., *Refactoring: Improving the Design of Existing Code.* Addison-Wesley 1999

/France et al. 03/
France R., Ghosh S., Song E., Kim D.-K., *A Metamodeling Approach to Pattern-based Model Refactoring*, in: IEEE Software, Special Issue on Model Driven Development, Vol.20, No.5, September/October 2003

/Gamma et al. 95/
Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns, Elements of Object- Oriented Software*, Reading, MA: Addison-Wesley, 1995

/Garlan, Shaw 93/
Garlan D., Shaw M., *An Introduction to Software Architecture*, Advances in Software Engineering and Knowledge Engineering, Vol 1. River Edge, NJ: World Scientific Publishing Company, 1993

/Garlan 95/
Garlan D., *Architectural Mismatch: Why It's Hard to Build Systems Out of Existing Parts*, Proceedings, 17th International Conference on Software Engineering. Seattle, WA, April 23-30, 1995. New York: Association for Computing Machinery, 1995. pp. 170-185

/Garlan, Perry 95/
Garlan D, Perry, D., *Introduction to the Special Issue on Software Architecture,* in IEEE Transactions on Software Engineering, Vol 21, No. 4, 1995

/Garlan et al. 95/
Garlan D., Allen R., Ockerbloom J., *Architectural Mismatch or Why it's hard to build systems out of existing parts*. in: Proceedings of the 17th International Conference on Software Engineering, Seattle, Washington, ACM SIGSOFT, April 1995, pp. 179-185

/Garlan et al. 97/
Garlan D., Monroe R., Wile D., *ACME: An Architectural Description Interchange Language*, in: Proceedings of CASCON 97, November 1997

/Gericke, Liggesmeyer 02/
Gericke J., Liggesmeyer P., *Eine Erweiterung der Unified Modeling Language zur Verfolgung von Software-Anforderungen in sicherheitskritischen Systemen*, Informatik Forschung und Entwicklung 17 (2), 2002 pp. 60-67

/Glass 98/
Glass R.L., *Reuse: What's wrong with this picture?*, IEEE Software, Ausgabe 3-4, 1998, Seite 57

/Gomaa 92/
Gomaa H., *An Object-Oriented Domain Analysis and Modeling Method for Software Reuse.* In Proceedings of the Hawaii International Conference on System Sciences, Hawaii, January, 1992

/Gomaa 00/
Gomaa H., *Designing Concurrent, Distributed, and Real-Time Applications with UML*, Addison-WesleyPublishing Company, Reading Massachusetts, 2000

/Gomez et al. 97/
Gomez F.C., Escrig D. de, Ruiz V. V., *A Sound and Complete Proof System for Probalistic Processes*, in: ARTS, 1997

/Goodenough, Sha 88/
Goodenough J. B., Sha L., *The priority ceiling protocol: A method for minimizing the blocking of high priority Ada tasks*, in: Proceedings of the International Workshop on Real-Time Ada Issues, Moretonhampstead, Devon, UK, 1988, pp. 20-31

/Grunske, Neumann 02/
Grunske L., Neumann R., *Quality Improvement by Integrating Non-Functional Properties in Architecture Specification*, Proceedings of the 2nd Workshop on Evaluating and Architecting System dependabilitY (EASY 02) at ASPLOS-X, San Jose/California, October 3-6, 2002, pp. 23-33

/Grunske 03a/
Grunske L., *Transformational Patterns for the Improvement of Safety Properties in Architectural Specifications*, Proceedings of The Second Nordic Conference on Pattern Languages of Programs (VikingPLoP 03), Bergen/Norge, 2003

/Grunske 03b/
Grunske L., *Annotation of Component Specifications with Modular Analysis Models for Safety Properties*, Proceedings of the 1st International Workshop on Component Engineering Methodology, Erfurt (WCEM 03), September 22, 2003, pp. 31-41

/Grunske 03c/
Grunske L., *Automated Software Architecture Evolution with Hypergraph Transformation*, in Proceedings of the 7th International IASTED on Conference Software Engineering and Application (SEA 03), Marina del Ray, Nov. 3-5, 2003, pp. 613-621

/Grunske 03d/
Grunske L., *Application of Behavior-Preserving Transformations to Improve Non-Functional Properties of an Architecture Specification*, in: Proceedings of the 4th International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'03), Lübeck, October 16-18, 2003, pp. 439-446

/Grunske 03e/
Grunske L., *A Visual Architecture Description Language for Embedded Systems with Hierarchical Typed Hypergraphs*, in Proceedings 3rd Workshop on Domain-Specific Modeling at the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 03), Anaheim, 2003, pp.1-8

/Grunske 04/
Grunske L., *Improving Quality Characteristics by Architecture Evolution*, submitted chapter, in H: Yang (ed.), Software Evolution with UML and XML to appear *2004*

/Guttag 77/
Guttag J., *Abstract data types and the development of data structures*, in: Communications of the Association for Computing Machinery, Vol. 20, June, 1977, pp. 297-404

/Habel 92/
Habel A., *Hyperedge Replacement: Grammars and Languages*, in: Lecture Notes in Computer Science, No. 643, Berlin: Springer 1992

/Habel et al. 01/
Habel A., Müller J., Plump D., *Double-Pushout Graph Transformation Revisited*, in: *Mathematical Structures in Computer Science* 11(5), pp 637-688, 2001

/Harel 87/
Harel D., *Statecharts: a Visual Formalism for Complex Systems*, Science of Computer Programming 8, pp. 231-274, 1987

/Haxthausen, Peleska 00/
Haxthausen A.E., Peleska J., *Formal Development and Verification of a Distributed Railway Control System*, IEEE Transactions on Software Engineering Vol. 26, No. 8, pp. 687-701, 2000

/Hayes-Roth 94/
Hayes-Roth F., *Architecture-Based Acquisition and Development of Software: Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) Program*

/Heckel et al. 99/
Heckel R., Engels G., Ehrig H., Taentzer G., *Classification and Comparison of Module Concepts for Graph Transformation Systems*, in: Ehrig H., Engels G., Kreowski H.- J., Rozenberg G. (eds.), *Handbook of Graph Grammars and Computing by Graph Transformation*, Vol. 2: Applications, Languages and Tools, Singapore: World Scientific Publisher 1999

/Heimdahl, Leveson 96/
Heimdahl M. P. E., Leveson N. G., *Completeness and Consistency in Hierarchical State-Based Requirements*, Transaction on Software Engineering 22(6): pp. 363-377, 1996

/Heitmeyer et al. 96/
Heitmeyer C. L., Jeffords R.D., Labaw B.G., *Automated consistency checking of requirements specifications*, ACM Transactions on Software Engineering and Methodology Vol. 3, pp. 231-261, July 1996

/Henzinger et al. 94/
Henzinger T., Nicollin X., Sifakis J., Yovine S., *Symbolic Modelchecking for Real-time Systems*, Information and Computation 111, pp. 193-244, 1994

/Henderson 97/
Henderson P., *Formal models of process components*, in: Proceedings of the FSE'97, FoCBS Workshop, Zurich, pp. 131-140, 1997

/Herrtwich, Hommel 94/
Herrtwich R.G., Hommel G., *Nebenläufige Programme*, Springer-Verlag, Berlin, Heidelberg, New York, 1994

/Hirsch et al. 99/
Hirsch D., Inverardi P., Montanari U., *Modeling Software Architectures and Styles with Graph Grammars and Constraint Solving*, in: Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1), San Antonio, Texas, U.S.A., February 22-24, 1999, pp. 127-142

/Hirsch, Montanari 99/
Hirsch D., Montanari U., *Consistent Transformations for Software Architecture Styles of Distributed Systems*, in: Proceedings of the Workshop on (formal methods applied to) Distributed Systems, Gheorghe Stefanescu, Ed., Iasi, Rumania, September 2-3, Electronic Notes in Theoretical Computer Science, Vol. 28, 1999, pp. 23-40

/Hirsch, Montanari 00/
Hirsch D., Montanari U., *Higher-Order Hyperedge Replacement Systems and their Transformations: Specifying Software Architecture Reconfigurations*, in: Proceedings of the Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems (GRATRA 2000), Satellite Event of the European Joint Conference on Theory and Practice of Software (ETAPS 2000), H. Ehrig, G. Taentzer, Eds., Berlin, Germany, March 25-27, Technical Report of Computer Science Department/TU Berlin, No. 2000-02, 2000, pp. 215-223

/Hopcroft, Ullman 97/
Hopcroft J.E., Ullman J.D., *Introduction to Automata Theory*, Languages and Computation Addison-Wesley, Reading, Massachusetts/USA, 1979
/Hoare 85/
Hoare C. A. R., *Communicating Sequential Processes*, International Series in Computer Science, Prentice-Hall, Inc., 1985
/Hoffmann, Minas 01/
Hoffmann B., Minas M., *Transformation of shaped nested graphs and diagrams*, in: Brand M. van den, Verma R. (eds.), Electronic Notes in Theoretical Computer Science, Vol. 59, Elsevier Science Publishers, 2001
/Hofmeister et al. 99/
Hofmeister C., Nord R., Soni D., *Applied Software Architecture*, Reading, MA: Addison Wesley Longman 1999
/Holt 98/
Holt R. C., *Structural Manipulations of Software Architecture using Tarski Relational Algebra*, WCRE '98: Working Conference on Reverse Engineering, Honolulu, October 1998
/Holt 02/
Holt R. C., *Introduction to the Grok Programming Language*, weblink http://plg.uwaterloo.ca/~holt/papers/grok-intro.doc download version May 2002
/Hopcroft, Ullman 79/
Hopcroft J. E., Ullman J. D., *Introduction to Automata Theory, Languages and Computation,* Addison-Wesley Publishing Company Inc., California, 1979
/Hruschka, Rupp 01/
Hruschka, P., Rupp, C., *„Echt Zeit" für Use-Cases*, in: OBJEKTspektrum, No. 4, April 2001
/Huang, Kintala 93/
Huang Y., Kintala C., *Software Implemented Fault Tolerance: Technologies and Experience*, in: Proceedings of the 23rd Fault-Tolerant Computing Symposium, June 1993, pp. 2-9
/Hunter, Nuseibeh 97/
Hunter A., Nuseibeh B., *Analysing Inconsistent Specifications*, in: Proc. of 3rd Int. Symposium on Requirements Engineering (RE97), Annapolis, USA: IEEE CS Press, January 1997
/Hunter, Nuseibeh 98/
Hunter A., Nuseibeh B., *Managing Inconsistent Specifications: Reasoning, Analysis and Action* , Transactions on Software Engineering and Methodology, ACM Press, October 1998
/Hüsener 94/
Hüsener T., *Entwurf komplexer Echtzeitsysteme*, BI Wiss.Verlag, Mannheim, Leipzig, Wien, Zürich, 1995
/IEEE 610.12/
IEEE-STD 610.12-1990 - *Standard Glossary of Software Engineering Terminology*, 1990
/Igarashi, Kobayashi 01/
Igarashi A., Kobayashi N., *A generic type system for the Pi-calculus*, in: POPL 01: Principles of Programming Languages, ACM, 2001, pp. 128-141
/ITU Z.120/
ITU-T. Recommendation Z.120. *ITU - Telecommunication Standardization Sector*, Geneva, Switzerland, May 1996
/Jacobsen et al. 92/
Jacobson I., Christerson M., Jonsson M., van Overgaard P., *Object Oriented Software Engineering, A Use Case Driven Approach*, Addison-Wesley, Reading, Massachusets 1992
/Jacobson et al. 99/
Jacobson I., Booch G., Rumbaugh J., *The Unified Software Development Process*, Addison-Wesley, 1999.
/Jarke 98/
Jarke M., *Requirement Tracing*, Communications of the ACM, Vol. 41, Issue 12, December 1998

/Jarvinen et al. 90/
Jarvinen H.-M., Kurki-Suonio R., Sakkinnen M., Systa K., *Object-Oriented Specification of Reactive Systems*, Proceedings of International Conference on Software Engineering, IEEE, 1990

/Jazayeri et al. 00/
Jazayeri M., Ran A., van der Linden F., *Software Architecture for Product Families – Principles and Practice*, Addison-Wesley, Reading, 2000

/Jonsson, Padilla 01/
Jonsson B., Padilla G., *An Execution Semantics for MSC2000*, Tenth SDL Forum, Copenhagen, June 2001, LNCS 2078, p. 365 ff.

/Kahn 74/
Kahn G., *The Semantics of a Simple Language for Parallel Programming*, in: Proc. of IFIP Congress 74, pages 471--475. North Holland Publishing Company, 1974.

/Kahn, MacQueen 77/
Kahn G., MacQueen D. B., *Coroutines and Networks of Parallel Processes*, in: B. Gilchrist (editor): Information Processing 77, North-Holland Publishing Co. 1977

/Kaiser et al. 03/
Kaiser B., Liggesmeyer P., Mäckel O., *A New Component Concept for Fault Trees*, in: Proceedings of the 8th Australian Workshop on Safety Critical Systems and Software (SCS'03), Adelaide, 2003

/Kang et al. 90/
Kang K., Cohen S., Hess J., Nowak W., Peterson S., *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1990

/Kaplan et al. 91/
Kaplan S., Loyall J., Goering S., *Specifying Concurrent Languages and Systems with Δ-Grammars*, in: Ehrig H., Kreowski H.-J., Rozenberg G. (eds.): Proc. 4th Int. Workshop on Graph Grammars and Their Application to Computer Science, LNCS 532, Springer Verlag 1991, pp. 475-489

/Karlsson 96/
Karlsson J., *Software Requirements Prioritizing*, Proceedings of the 2nd International Conference on Requirements Engineering (ICRE 96), IEEE 1996

/Karlsson 97/
Karlsson J., *A Cost-Value Approach for Prioritizing Requirements*, IEEE Software, 14(5), September 1997, pp. 67-74

/Karlsson et al. 98/
Karlsson J., Wohlin C., Regnell B., *An Evaluation of Methods for Prioritizing Software Requirements*, Information and Software Technology, (39) 14-15, 1998, pp. 939-947

/Karsai et al. 03/
Karsai G., Agrawal A., Shi F., Sprinkle J., *On the use of Graph Transformations in the Formal Specification of Computer-Based Systems*, IEEE TC-ECBS and IFIP10.1 Joint Workshop on Formal Specifications of Computer-Based Systems, Huntsville, Alabama, April 9, 2003, p. 19-27

/Katoen, Lambert 98/
Katoen J.P, Lambert L., *Pomsets for message sequence charts*, SAM98: 1st conference on SDL and MSC, 1998, pp. 281-290

/Kazeman et al. 96/
Kazman R., Abowd G., Bass L., Clements P., *Scenario-based analysis of software architecture*, IEEE Software, pages 47-53, November 1996

/Kazeman, Burth 98/
Kazman R., Burth M., *Assessing Architectural Complexity*, Proceedings of the 2 nd Euromicro Conference on Software Maintenance and Reengineering (CSMR 98), IEEE Computer Society Press, 1998

/Kazeman et al. 99/
Kazman R., Klein M., Clements P., *Evaluating software architecture for real-time systems*, Annals of Software Engineering, 7, 1999

/Kazeman et al. 01/
Kazman R., *Software Architecture*, in: Handbook of Software Engineering and Knowledge Engineering, S-K Chang (ed.), World Scientific Publishing, 2001

/Kececioglu 91/
Kececioglu D., *Reliability Engineering Handbook*, Vol. I, Prentice--Hall, Inc., Englewood Cliffs, NJ. 1991

/Klein et al. 94/
Klein M.H., Lehoczky J.P., Rajkumar R., *Rate Monotonic Analysis for Real Time Industrial Computing*, IEEE Computer, January 1994.

/Klein et al. 93/
Klein M. H., Ralya T., Pollak B., Obenza R., Gonzalez Harbour M., *A Practitioners Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Boston, MA: Kluwer Academic Publishers, 1993

/Knight, Leveson 86/
Knight J. C. Leveson N. G., *An Experimental Evaluation of the Assumption of Independence in Multiversion Programming*, IEEE Transactions on Software Engineering, Volume 12, Number 1, January 1986, pp. 96-109

/King 89/
King R., *My cat is object-oriented*, in: Won Kim Frederick H. Lochovsky, editors, Object-Oriented Concepts, Databases, Applications, pages 23--30. Addison Wesley, Reading, MA, 1989

/Koopman 96/
Koopman P., *Embedded System Design Issues -- The Rest of the Story*, Proceedings of the 1996 International Conference on Computer Design, Austin, October 7-9 1996

/Kruchten 95/
Kruchten, P. B. *The 4+1 View Model of Architecture*. IEEE Software 12, 6 November 1995 pp. 42-50

/Lala, Harper 94/
Lala J. H., Harper R. E., *Architectural Principles for Safety-Critical Real-Time Applications*, Proc. of the IEEE, vol. 82, no. 1, Jan. 1994, pp. 25-40

/Lamsweerde, Letier 98/
Lamsweerde A van, Letier E., *Integrating Obstacles in Goal-Driven Requirements Engineering*, Proc. ICSE-98: 20th International Conference on Software Engineering, Kyoto, April 1998

/Lamsweerde et al. 98/
Lamsweerde A van, Darimont R., Letier E., *Managing Conflicts in Goal-Driven Requirements Engineering*, IEEE Transactions on Software Engineering, *Special Issue on Managing Inconsistency in Software Development*, November 1998

/Lamsweerde, Willemet 98/
Lamsweerde A. van, Willemet L., *Inferring Declarative Requirements Specifications from Operational Scenarios*, IEEE Transactions on Software Engineering, *Special Issue on Scenario Management*, December 1998

/Lamsweerde, Letier 00/
Lamsweerde A. van, Letier E., *Handling Obstacles in Goal-Oriented Requirements Engineering*, IEEE Transactions on Software Engineering, *Special Issue on Exception Handling*, Vol. 26, No. 10, October 2000, pp. 978-1005

/Lamsweerde 00/
Lamsweerde A. van, *Requirements Engineering in the Year 00: A Research Perspective* Invited Paper for ICSE'2000 - 22nd International Conference on Software Engineering, Limerick, ACM Press, 2000

/Lamsweerde 01a/
Lamsweerde A. van, *Goal-Oriented Requirements Engineering: A Guided Tour*, Proc. of the 5th International Symposium on Requirements Engineering (RE 01), pp. 249-261, IEEE CS Press 2001

/Lamsweerde 01b/
Lamsweerde A. van, *Building Formal Requirements Models for Reliable Software*, in: Reliable Software Technologies, Ada-Europe`2001, Springer-Verlag Lecture Notes in Computer Science, LNCS 2043, May 2001

/Lamsweerde 01c/
Lamsweerde A. van, *Formal Specification: a Roadmap*, in: The Future of Software Engineering, A. Finkelstein (ed.), ACM Press 2001

/Laprie 92/
Laprie J.C. (ed.), *Dependability: Basic Concepts and Associated Terminology*. Vol. 5, Dependable Computing and Fault-Tolerant Systems Series, Vienna: Springer 1992

/Latronico, Koopman/
Latronico E., Koopman P., *Representing Embedded System Sequence Diagrams as a Formal Language*, UML 2001, Toronto Ontario, 3-5 Oct. 2001, LNCS 2185, pp. 302-316

/Le Metayer 96/
Le Metayer D., *Software architecture styles as graph grammars*, in: Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering, October 16-18, 1996, Vol. 216 of ACM Software Engineering Notes, New York: ACM Press, pp. 15-23

/Le Metayer 98/
Le Metayer D., *Describing software architecture styles using graph grammars*, IEEE Transactions on Software Engineering, Vol. 24, No. 7, July 1998, pp. 521-553

/Lee 01/
Lee E. A., *Computing for Embedded Systems*, IEEE Instrumentation and Measurement Technology Conference, Budapest, Hungary, May 21-23, 2001

/Lee, Xiong 02/
Lee E. L., Xiong Y., *Behavioral Types for Component-Based Design*, in: Memorandum UCB/ERL M02/29, University of California, Berkeley, CA 94720, USA, September 27, 2002

/Leveson 95/
Leveson N. G., *Safeware: System Safety and Computers*. Addison-Wesley, 1995

/Liggesmeyer 90/
Liggesmeyer P., *Modultest und Modulverifikation*, BI Wiss. Verlag, Mannheim; Wien; Zürich, 1990

/Liggesmeyer 00/
Liggesmeyer P., *Qualitätssicherung softwareintensiver technischer Systeme*, Spektrum-Akademischer-Verlag, Heidelberg, 2000

/Liggesmeyer 02/
Liggesmeyer P., *Software-Qualität: Testen, Analysieren und Verifizieren von Software*, Spektrum-Akademischer-Verlag, Heidelberg, Berlin, 2002

/Lindsay et al. 00/
Lindsay P. A., McDermid J. A., Tombs D. J., *Deriving Quantified Safety Requirements in Complex Systems*, F. Koorneef M. van Meulen (eds.): SAFECOMP 2000, LNCS 1943, pp. 117-130, Berlin, Heidelberg: Springer 2000

/Liu, Layland 73/
Liu C. L., Layland J. W., *Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment*, in: Journal of the ACM, Vol. 20, No. 1, Jan., pp. 46-61,1973

/Liskov, Zilles 74/
Liskov B., Zilles S., *Programming with abstract data types*, ACM Sigplan Notices, 9 (4), pp 50 - 59, 1974

/Löwe 93/
Löwe M., *Algebraic approach to single pushout graph transformations*, in: Journal of Theoretical Computer Science, Elsevier, Vol. 109, 1993, pp. 181-224

/Lowe 95/
Lowe G., *Probabilistic and Prioritized Models of Timed CSP*, in: Theoretical Computer Science 138, (Special Issue on the Mathematical Foundations of Programming Semantics Conference 1992), 1995, pp. 315-352

/Loyall, Kaplan 92/
Loyall J., Kaplan S., *Visual Concurrent Programming with Delta-Grammars*, in Journal of Visual Languages and Computing, Vol 3, 1992, pp. 107-133

/Luckham et al. 95/
Luckham D. C., Kenney J. J., Augustin L. M., Vera J., Bryan D., Mann W., *Specification and Analysis of*

System Architecture Using Rapide, in: IEEE Transactions on Software Engineering, Vol. 21, No. 4, April, pp. 336-355, 1995

/Luckham et al. 93/
Luckham D. C., Vera J., Bryan D., Augustin L., Belz F., *Partial orderings of event sets and their application to prototyping concurrent, timed systems*, in: Journal of Systems and Software, Vol. 21, No. 3, June, pp. 253-265, 1993

/Lundberg et al. 99/
Lundberg L., Bosch J., Häggander D., Bengtsson P.-O., *Quality Attributes in Software Architecture Design*, Proceedings of the IASTED 3rd International Conference on Software Engineering and Applications, pp. 353-362, October 1999

/Lutz, Woodhouse 96/
Lutz R. R., Woodhouse R., *Experience Report: Contributions of SFMEA to Requirements Analysis*, in: Proc. of ICRE 96, 1996

/Lutz, Woodhouse 97/
Lutz R. R., Woodhouse R., *Requirements Analysis Using Forward and Backward Search*, Annals of Software Engineering, Special Volume on Requirements Engineering, 3, 1997, pp. 459-475

/Leinhos 96/
Leinhos D., *Analyse und Entwurf von Ortungssystemen für Schienenverkehr mit strukturierten Methoden*, Fortschritt-Berichte VDI, TU Braunschweig, München, 1996

/Lyu 96/
Lyu M. R., *Handbook of Software Reliability Engineering*, McGraw-Hill, 1996

/Nicholson et. al. 94/
Nicholson M., McDermid J. A., Burns A., *Analysis and Design Synthesis for Hard Real-Time Safety Critical Systems*; YCS-94-237, November 1994

/Niere, Zündorf 00/
Niere J., Zündorf A. J., *Using Fujaba for the development of production control systems*, in: Proc. Int.Workshop Agtive 99, Nagl M., Schürr A., Münch M. (eds.), Vol. 1779 of Lecture Notes in Computer Science, Springer-Verlag 2000, pp. 181-191

/Nierstrasz 93/
Nierstrasz O., *Regular Types for Active Objects,* in: Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'**93**). ACM Press, 1993

/Nitzberg, Lo 91/
Nitzberg B., Lo V., *Distributed shared memory: A survey of issues and algorithms*, IEEE Computer, vol. 24, pp. 52--60, Aug. 1991

/Nordland 99/
Nordland O., *A discussion of Risk Tolerance Principles*, in: Safety Systems, vol. 8, no. 3, pp. 1-4, 1999

/Nuseibeh, Easterbrook 00/
Nuseibeh B., Easterbrook S., *Requirements Engineering: A Roadmap*, Proceedings of International Conference on Software Engineering (ICSE-2000), 4-11 June 2000, Limerick, Ireland

/Magee, Krammer 96/
Magee J., Krammer J., *Dynamic Structure in software architectures*, in: Proceedings of the ACM SIFSOFT'96 Fourth Symposium on the foundations of software engineering (FSE4), San Francisco, CA, pp. 3-14, 1996

/Magee et al. 95/
Magee J., Dulay N., Eisenbach S., Kramer J., *Specifying Distributed Software Architecture*, in: Software Engineering, ESEC'95, LNCS 989, Springer, 1995, pp. 137-153

/Mahmood, McCluskey 88/
Mahmood A., McCluskey E. J., *Concurrent error detection using watchdog processors – a survey*, IEEE Transactions on Computers, vol. 37, February 1988, pp. 160-174

/Maier, Moldt 97/
Maier C., Moldt D., *Object Coloured Petri Nets - a Formal Technique for Object Oriented Modelling*, in: Farwer B., Moldt D., Stehr M.-O. (eds.), Petri Nets in System Engineering, Modelling, Verification and Validation, University of Hamburg, 1997, pp. 11-19

/Manna, Pnueli 92/
Manna Z., Pneuli A., *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag: Heidelberg, 1992

/Mauri 01/
Mauri G., *Integrating Safety Analysis Techniques, Supporting Identification of Common Cause Failures,* Dissertation, University of York, YCST-2001-02, York, 2001

/Mauw 96/
Mauw S., *The formalization of Message Sequence Charts*, Computer Networks and ISDN Systems, 28(12): pp. 1643-1657, 1996

/McDermid, Pumfrey 95/
McDermid J.A., Pumfrey D.J., *A Development of Hazard Analysis to aid Software Design,* in Proceedings of the Ninth Annual Conference on Computer Assurance COMPASS '94, Gaithersburg, IEEE 1995, pp. 17-25

/McDermid et al. 95/
McDermid J. A., Nicholson M., Pumfrey D. J., Fenelon P., *Experience with the Application of HAZOP to Computer-Based Systems*, in: Proceedings of the Tenth Annual Conference on Computer Assurance COMPASS '95, Gaithersburg, pp. 37-48,1995

/McDermid, Pumfrey 98/
McDermid J.A., Pumfrey D.J., *Safety analysis of hardware/software interactions in complex systems*, in: Proceedings of 16th International System Safety Conference, Seattle, 1998

/McIlroy 68/
McIlroy M. D., *Mass produced software components*, in: Peter Naur and Brian Randell (eds), Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, NATO Scientific Affairs Division, October 1968

/Medvidovic et al. 96/
Medvidovic N., Oreizy P., Robbins J. E., Taylor R. N., *Using object-oriented typing to support architectural design in the C2 style*, in: Proceedings of the Fourth ACM Symposium on the Foundations of Software Engineering, SIGSOFT'96, ACM Press, October 1996

/Medvidovic et al. 97/
Medvidovic N., Oreizy P., Taylor R. N., *Reuse of Off-the-Shelf Components in C2-Style Architectures*. Proceedings of the 1997 Intemational Conference on Software Engineering (ICSE'97), Boston, MA, May 1997

/Medvidovic, Rosenblum 99/
Medvidovic N., Rosenblum D. S., *Assessing the Suitability of a Standard Design Method for Modeling Software Architectures*, in: Proceedings of the TC2 First IFIP Working Conference on Software Architecture (WICSA1), Kluwer Academic Publishers, pp. 161-182 , 1999

/Medvidovic et al. 99/
Medvidovic N., Rosenblum D. S., Taylor R. N., *A language and environment for Architecture-based software development and Evolution*, in: Proceedings of the 21th International Conference on Software Engineering (ICSE 99), Los Angeles, California, May 1999

/Medvidovic, Taylor 00/
Medvidovic N., Taylor R., *A Classification and Composition Framework for Software Architecture Description Languages*, IEEE Transactions on Software Engineering, Vol. 26, No.1, January 2000

/Mens 99/
Mens T., *A formal foundation for object-oriented software evolution*, PhD Dissertation, Department of Computer Science, Vrije Universiteit Brussel, September 1999

/Mens 00/
Mens T., *Conditional graph rewriting as a domain-independent formalism for software evolution*, in: Proc. Int'l Conf. Agtive 1999: Applications of Graph Transformations with Industrial Relevance, Vol. 1779 of Lecture Notes in Computer Science, Springer-Verlag 2000, pp. 127-143

/Mens, D'Hondt 00/
Mens T., D'Hondt T., *Automating support for software evolution in UML*, in: Automated Software Engineering Journal, Vol. 7, No. 1, February 2000, pp. 39-59

/Mens et al. 02/
Mens T., Demeyer S., Janssens D., *Formalising behaviour preserving program transformations*, in: Corradini A., Ehrig H., Kreowski H.-J., Rozenberg G. (eds.), *Proc. Int'l Conf. Graph Transformation*, Vol. 2505 of Lecture Notes in Computer Science, Springer-Verlag 2002, pp. 286-301

/Mens, van Eetvelde 03/
Mens T., van Eetvelde N., *Formalising refactoring with graph transformations*, Fundamenta Informaticae, 2003, Special Issue on Graph Rewriting. In progress

/Meyer 97/
Meyer B., *Object-Oriented Software Construction*, Second Edition, Prentice Hall, New York, 1997

/MIL HDBK 338B/
MIL HDBK 338B, *Electronic Reliability Design Handbook* - Revision B - 1 October 1998

/Mili et. al. 99/
Mili A., Yacoub S., Addy E., Mili H., *Toward an engineering discipline of Software reuse*, IEEE Software, Ausgabe 7-8, 1999, Seite 22-33

/Milner 89/
Milner R., *Communication and Concurrency*, Prentice Hall, New York, 1989

/Mitra et al. 99/
Mitra S., Saxena N.R., McCluskey E. J., *Design Diversity for Redundant Systems*, 29th International Symposium on Fault-Tolerant Computing (FTCS-29) Fast Abstracts, pp. 33-34, Madison, WI, June 15-18, 1999

/Moriconi et al. 95/
Moriconi M., Qian X., Riemenschneider R. A., *Correct Architecture Refinement*, in: IEEE Transactions on Software Engineering, Vol. 21, No. 4, April 1995, pp. 356-372

/Mylopoulos et al. 92/
Mylopoulos J., Chung L., Nixon B., *Representing and Using Non-Functional Requirements: A Process-Oriented Approach*, in: IEEE Transactions on Software Engineering 18(6), 1992, pp. 483-497

/Okstad, Hokstad 01/
Okstad E., Hokstad P., *Risk Assessment and use of Risk Acceptance Criteria for the regulation of dangerous substances*, in: Proc. of ESREL2001 Torino, Italy: September 2001

/OMG 03/
OMG, *The Model-Driven Architecture*, http://www.omg.org/mda/ Download: 17.11.03

/Padberg 03/
Padberg J., *Basic Ideas for Transformations of Specification Architectures*, Accepted for Workshop on Software Evolution Through Transformations, Satellite of the 1st Int. Conference on Graph Transformation (ICGT'02), 2002, in: Electronic Notes in Theoretical Computer Science, Vol. 72, No. 4, 2003

/Pagel, Six 94/
Pagel B.-U., Six H.-W., *Software Engineering, Band 1: Die Phasen der Softwareentwicklung*, Addison Wesley, Bonn 1994

/Papadopoulos, McDermid 99/
Papadopoulos Y., McDermid J. A., *A new method for Safety Analysis and the Mechanical Synthesis of Fault Trees in Complex Systems*, in Proceedings of ICSSEA `99, 12 th International Conference on Software and Systems Engineering and their Applications, 4(13):1-9, Paris, December 1999. 228-239

/Papadopoulos 00/
Papadopoulos Y., *Safety-Directed System Monitoring Using Safety Cases*, Dissertation, University of York,YCST-2000-08, York, 2001

/Papadopoulos et al. 01/
Papadopoulos Y., McDermid J. A., Sasse R., Heiner G., *Analysis and Synthesis of the Behaviour of Complex Programmable Electronic Systems in Conditions of Failure*, Reliability Engineering and System Safety, 71(3), Elsevier Science, 2001 pp. 229-247

/Pardo et al. 00/
Pardo J., Valero V., Cuartero F., *A Dynamic State Graph for a Timed Process Algebra*, in: Proceedings of SNPD'00, Ed. ACIS, Reims (France), May 2000

/Parnas 72/
Parnas D., *On the Criteria for Decomposing Systems into Modules*, Communications of the ACM 15, 12 December 1972, pp. 1053-1058

/Parnas 76/
Parnas D., *On the Design and Development of Program Families*, IEEE Transactions on Software Engineering SE-2, 1, 1976, pp. 1-9

/Paulk et al. 93/
Paulk M. C., Curtis B., Chrissis M. B., Weber C. V., *Capability Maturity Model, Version 1.1*, IEEE Software, Vol. 10, No. 4, July 1993, pp. 18-27

/Perry Wolf 92/
Perry D.E., Wolf A.L., *Foundations for the Study of Software Architecture*, Software Engineering Notes, ACM SIGSOFT 17, 4: 40-52, October 1992

/Pierce, Sangiorgi 00/
Pierce B., Sangiorgi D., *Behavioral Equivalence in the Polymorphic Pi-Calculus*, in: POPL 97, Journal of the ACM (JACM), Vol. 47, Issue 3, May 2000

/Pont 01/
Pont M.J., *Patterns for time-triggered embedded systems: Building reliable applications with the 8051 family of microcontrollers*, ACM Press / Addison-Wesley 2001

/Pont, Ong 02/
Pont M.J., Ong H.L.R., *Using watchdog timers to improve the reliability of TTCS embedded systems*, in Hruby, P. Soressen, K. E. Proceedings of the First Nordic Conference on Pattern Languages of Programs, September, 2002, pp. 159-200

/Pratt 86/
Pratt V., *Modeling Concurrency with Partial Orders*, in: International Journal of Parallel Programming, Vol. 15, No. 1, 1986, pp. 33-71

/Priami 95/
Priami C., *Stochastic pi-calculus with General Distributions*, in: The Computer Journal, Vol. 38, No. 6, 1995, pp. 578-589

/Prowell, Poore 03/
Prowell S. J., Poore J. H., *Foundations of Sequence-Based Software Specification*, IEEE Transactions on Software Engineering, May 2003, Vol. 29, Issue 5, 2001, pp. 417-429

/Pumfrey 99/
Pumfrey D.J., *The Principled Design of Computer System Safety Analyses*, Dissertation, University of York 1999

/Plump, Habel 96/
Plump D., Habel A., *Graph Unification and Matching*, in: *Proc. Graph Grammars and Their Application to Computer Science*, Lecture Notes in Computer Science 1073, Springer, pp. 75-89, 1996

/Rajamani, Rehof 01/
Rajamani S. K., Rehof J., *A Behavioral Module System for the Pi-Calculus*, in: Static Analysis Symposium (SAS01), Paris, France, 16-18 July 2001

/Ramesh, Jarke 01/
Ramesh B., Jarke M., *Towards Reference Models for Requirements Traceability*, IEEE Transactions on Software Engineering, Jan. 2001, Vol. 27, Issue 1, 2001

/Randell, Xu 95/
Randell B., Xu J., *The Evolution of the Recovery Block Concept*, in: Software Fault Tolerance, Michael R. Lyu, editor, Wiley, 1995, pp. 1-21

/Randel 75/
Randel B., *System structure for software fault tolerance*, IEEE Transactions on Software Engineering, No. 2, 1975, pp. 220-232.

/Raskin, Schobbens 97/
Raskin J.-F., Schobbens P.-Y., *State-clock Logic: A Decidable Real-Time Logic*, Proc. HART '97: Hybrid and Real-Time Systems, LNCS 1201, 1997, pp. 33-47

/Reed, Roscoe 88/
Reed G.M., Roscoe A.W., *A timed model for Communicating Sequential Processes*, in: Theoretical Computer Science, 58, 1988, pp. 249-261

/Renpenning 01/
Renpenning F., *Gefährdungsanalyse am Beispiel des FunktFahrBetriebs*, in: Proc. Kolloquium: Moderne Sicheranalysen Theorie und Praxis, Braunschweig, Oktober 2001

/Riemenschneider 99/
Riemenschneider R. A., *Checking the correctness of architectural transformation steps via proof-carrying architectures*, in: Proceedings of the First Working IFIP Conference on Software Architecture, Kluwer Academic Press 1999

/Riemenschneider et al. 00/
Riemenschneider R.A., Dutertre B., Stavridou V., Salasin J., van Lamsweerde A., *From System Requirements to System Architecture*, Proceedings of the Fourth International Software Architecture Workshop, IEEE-ACM (June 2000)

/Robertson, Robertson 00/
Robertson J., Robertson S., *Volere Requirements Specification Template*, in: Principals of the Atlantic Systems Guild, London, Aachen & New York 2000

/Robertson, Robertson 99/
Robertson S., Robertson J., *Mastering the requirements process*, Addison-Wesley, Harlow, England, 1999

/Robinson et al. 99/
Robinson W. N., Pawlowski S. D., Volkov V., *Requirements Interaction Management*, Department of Computer Information Systems, Georgia State University, Atlanta, GA 30302, 1999

/Robinson, Volkov 98/
Robinson W. N., Volkov S., *Supporting the Negotiation Life Cycle*, ACM, Communications of the ACM, May 1998, pp. 95-102

/Robinson, Volkov 97/
Robinson W. N., Volkov S., *A meta-model for restructuring stakeholder requirements*, Proceedings of the 19th international conference on Software engineering, May 17-23, Boston, Massachusetts 1997, pp. 140-149

/Roscoe 95/
Roscoe A. W., *Modelling and verifying key exchange protocols using CSP and FDR*. In: Proc. 8th IEEE Computer Security Foundations Workshop (CSFW), IEEE Computer Society Press,1995, pp 98-107

/Rudolph 99/
Rudolph E., Grabowski J., Graubmann P., *Towards a harmonization of UMLsequence diagrams and MSC*, in: R. Dssouli, G. von Bochmann, Y. Lahav, (eds.), SDL'99: Proceedings of the Ninth SDL Forum, North-Holland 1999

/Rumbaugh 91/
Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W., *Object-Oriented Modeling and Design*, Prentice Hall, 1991

/Rumbaugh et al. 99/
Rumbaugh J., Jacobson I., Booch G.. *The Unified Modeling Language Refrence Manual*, Object Technology Series, Addison Wesley Longman, Reading, Mass., 1999

/Ryan, Schneider 01/
Ryan P. Y. A., Schneider S.A., *Process algebra and non-interference*, in: Journal of Computer Security, Vol. 9, No.1/2, Special Issue on CSFW-12, 2001, pp. 75-103

/SAE ARP 4754/
Society of Automotive Engineers SAE *ARP 4754 - Certification Considerations for Highly-Integrated or Complex Aircraft Systems*," Aerospace Recommended Practice, Warrendale, 1994

/SAE ARP 4761/
Society of Automotive Engineers SAE ARP 4761, *Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment*, Aerospace Recommended Practice, Warrendale, 1996

/Saridakis 02/
Saridakis T., *A System of Patterns for Fault Tolerance*, in: Proceedings of the EuroPlop 2002
/Saridakis 03/
Saridakis T., *Design Patterns for Fault Containment*, Proceedings of the EuroPlop 2003
/Schneider 97/
Schneider S., *Timewise Refinement for Communicating Processes*, in: Science of Computer Programming, Vol. 28, No. 1, 1997, pp. 43-90
/Schürr 97/
Schürr A., *Programmed graph replacement systems*, in: Rozenberg G. (ed.), *Handbook of Graph Grammars and Computing by Graph Transformation*, Vol. 1: Foundations, Singapore: World Scientific Publisher, pp. 247-312, 1997, pp. 247-312
/Schürr et al. 99/
Schürr A., Winter A., Zündorf A., *The PROGRES Approach: Language and Environment*, in: Ehrig H., Engels G., Kreowski H.- J., Rozenberg G. (eds.), *Handbook of Graph Grammars and Computing by Graph Transformation*, Vol. 2: Applications, Languages and Tools, Singapore: World Scientific Publisher, pp. 487-550, 1999
/Schürr et al. 95/
Schürr A., Winter A. J., Zündorf A., *Graph grammar engineering with PROGRES*, in: Proc. European Conf. Software Engineering, Schäfer W., Botella P. (eds.), Vol. 989 of Lecture Notes in Computer Science, Springer-Verlag 1995, pp. 219-234
/Scott 82/
Scott D. S., *Domains for denotational semantics*. In: Proccedings of the International Colloquium on Automata, Languages and Programs, pages 577-- 613, Lecture Notes in Computer Science vol. 140, Springer, 1982
/Selic et al. 94/
Selic B., Gullekson G., Ward P. T., *Real-Time Object-Oriented Modeling*. Wiley, New York, 1994
/Sha et al. 90/
Sha L., Rajkumar R., Lehoczky J., *Priority Inheritance Protocols: An Approach to Real-time Synchronization*. in: IEEE Transactions on Computers, vol. 39, no. 3, pp. 1175-1198, 1990
/Sha, Goodenough 90/
Sha L., Goodenough J. B., *Real-time scheduling theory and Ada*, in: IEEE Computer, Vol. 23 April, 1990, pp. 53-62
/Shaw 94/
Shaw M., *Making Choices: A Comparison of Styles for Software Architecture*. IEEE Software 12, 6, November, 1995, pp. 27-41
/Shaw 98/
Shaw M., *Moving from Qualities to Architectures: Architectural Styles*, in: L. Bass, P. Clements, & R. Kazman (eds.), Software Architecture in Practice, Addison-Wesley, 1998
/Shaw 01/
Shaw A. C., *Real-time systems and software*, John Wiley & Sons, 2001
/Shaw, Clements 97/
Shaw M., Clements P., *A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems*, Proc. COMPSAC97, 1st Int'l Computer Software and Applications Conference, August, 1997
/Shaw et al. 96/
Shaw M., DeLine R., Zelesnik G., *Abstractions and Implementations for Architectural Connections*, in: Third International Conference on Configurable Distributed Systems, Annapolis, Maryland, May 1996
/Shaw, Garlan 93/
Garlan D., Shaw M., *An introduction to software architecture*, in: Advances in Software Engineering and Knowledge Engineering, volume 1. World Scientific Publishing Co., 1993
/Shaw, Garlan 96/
Shaw M., Garlan D., *Software Architecture. Perspectives on an Emerging Discipline*, Prentice Hall, 1996

/Six 01/
Six J., *Ableitung von Zahlenwerten für Risikoanalysen aus der Umfallstatistik* , in: Proc. Kolloquium: Moderne Sicheranalysen Theorie und Praxis, Braunschweig, Oktober 2001

/Sneed 95/
Sneed H., *Estimating the Costs of Object-Oriented Software*, in: Proceedings of Software Cost Estimation Seminar, 1995

/Sommerville, Sawyer 97/
Sommerville I., Sawyer P., *Requirements Engineering -- A Good Practice Guide*, John Wiley & Sons, New York, 1997

/Soni et al. 95/
Soni D., Nord R. L., Hofmeister C., *Software Architecture in Industrial Applications*, Proc. of the 17th International Conference on Software Engineering, Seattle, 1995

/Sprunt et al. 89/
Sprunt B., Sha L., Lehoczky J. P., *Aperiodic task scheduling for hardreal-time systems*, in: The Journal of Real-Time Systems, Vol. 1, 1989, pp. 27-60

/Stone et al 98/
Stone J., Greenwald M., Partridge C., Hughes J., *Performance of Checksums and CRCs over Real Data*, pp. 529-543, IEEE/ACM Trans. on Networking, Vol. 6, No. 5, October 1998

/Szyperski 98/
Szyperski C., *Component Software: Beyond ObjectOriented Programming*, Addison-Wesley, 1998

/Taentzer 99/
Taentzer G., *Adding Visual Rules to Object-Oriented Modeling Techniques*, in: Proc. of Technology of Object-Oriented Languages and Systems (TOOLS'99), IEEE, Nancy, France, 1999

/Taentzer et al. 99/
Taentzer G., Ermel C., Rudolf M., *The AGG Approach: Language and Tool Environment*, in: the Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools, World Scientific 1999

/Taentzer 01/
Taentzer G., *Visual Modeling of Distributed Object Systems by Graph Transformation*, in: M. Bauderon M. Corradini A. (eds.), Proc. GETGRATS Closing Workshop, Electronic Notes in Theoretical Computer Science (ENTCS), Vol. 55, 2001

/Tapken 99/
Tapken J., *Implementing hierarchical graph structures*, in: Finance J.- P. (eds.), Proc. Formal Aspects of Software Engineering (FASE'99), Vol. 1577, Lecture Notes in Computer Science, Springer 1996

/Tracz 95/
Tracz W., *DSSA (Domain-Specific Software Architecture) Pedagogical Example*, in: ACM SIGSOFT Software Engineering Notes, vol. 20, no. 4, July 1995

/V-Modell 97/
V-Modell 97, *Lifecycle Process Model -Developing Standard for IT Systems of the Federal Republic of Germany*. General Directive No. 250. 97

/Vestal et al. 97/
Vestal S., Guerby L., Dewar R., McConnell D., Lewis B., *Reimplementing a Multiprocess Distributed Paradigm for Real-Time Systems in Ada 95*, in: Proceedings of the 8th international workshop on Real-Time Ada, ACM SIGAda Ada Letters, October 1997 Vol. 17, Issue 5, ACM Press, New York, 1997, pp. 93-99

/Vincent et al. 88/
Vincent J., Waters A, Sinclair J., *Software Quality Assurance*, Practice and Implementation, Englewood Cliffs, Prentice-Hall, New York, 1988

/Wermelinger, Fiadeiro 99/
Wermelinger M., Fiadeiro J. L., *Algebraic Software Architecture Reconfiguration*; in: Software Engineering -ESEC/FSE'99, LNCS 1687, Springer-Verlag 1999, pp. 393-409

/Wermelinger et al. 01/
Wermelinger M., Lopes A., Fiadeiro J. L., *A Graph Based Architectural (Re)configuration Language*, in:

Proc. of the Joint 8th European Software Engineering Conference and 9th Symposium on the Foundations of Software Engineering, ACM Press 2001, pp. 21-32

/Wermelinger, Fiadeiro 02/
Wermelinger M., Fiadeiro J. L., *A Graph Transformation Approach to Software Architecture Reconfiguration*, Science of Computer Programming, Vol. 44, No. 2, August 2002, pp. 133-155

/Wiegers 97/
Wiegers K., *Listening to the Customer's Voice*, Software Development, Vol. 5, No. 3, March 1997

/Wiegers 99/
Wiegers K., *First Things First: Prioritizing Requirements*, Software Development, Vol. 7, No. 9, September 1999

/Wolberg, Kiefer 00/
Wolberg J., Kiefer J., *Life Cycle Costs - Die Kosten von Betrieb, Wartung und Verfügbarkeit*, Signal und Draht, Jg 92, 6/2000, pp. 19-25

/Yellin, Strom 97/
Yellin D., Strom R., *Protocol specifications and component adapters*, in: ACM Trans. Programming Languages and Systems, Vol. 19, 1997, pp. 292-333

/Zimmer 99/
Zimmer W., *Frameworks und Entwurfsmuster*, Dissertation Uni Karlsruhe Shaker Verlag, 1999