

Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Szenariobasierte Programmierung und verteilte Ausführung in Java

Masterarbeit

im Studiengang Informatik

von

Florian Wolfgang Hagen König

Prüfer: Prof. Dr. Joel Greenyer
Zweitprüfer: Prof. Dr. Kurt Schneider
Betreuer: Prof. Dr. Joel Greenyer

Hannover, 08.01.2017

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 08.01.2017

Florian Wolfgang Hagen König

Inhaltsverzeichnis

1. Einleitung	1
2. Grundlagen	7
2.1. Verteilte reaktive Systeme	7
2.2. Szenariobasierte Entwicklung	8
2.3. ScenarioTools	9
2.3.1. Modellierung der Systemkomponenten	9
2.3.2. Modellierung des Systemverhaltens in SML	9
2.3.3. Szenario	12
2.3.4. Violations und Nachrichtenverwaltung im Szenario	14
2.3.5. Bestandteile eines Szenarios	15
2.3.6. Ausführung von Szenarien	19
2.4. Deployment	24
2.5. Verteilte Ausführung	26
2.6. Behavioral Programming	26
2.6.1. B-Thread-Beispiel	28
2.7. Code-Generierung und Xtend	29
2.8. Objektsystemtransformation mit Henshin	30
3. Anforderungsanalyse	31
3.1. Problemanalyse	31
3.2. Lösungsansatz	33
3.3. Vision	34
3.4. Anforderungen	35
4. Szenariobasierte Programmierung in Java	41
4.1. Java Spezifikation	43
4.1.1. Szenarien einbinden	44
4.1.2. Transformationsregeln für das Objektsystem	45
4.2. Nachrichten in Java	46
4.3. Java Szenario	47
4.3.1. Ausführung von Szenarien beobachten	51

Inhaltsverzeichnis

4.4.	Java Objektsystem	52
4.5.	Java Runconfiguration	53
4.5.1.	Start einer SpecificationRunconfig	55
4.6.	Runtime-Adapter	55
4.6.1.	Runtime-Adapter Interface und Adapterbau	55
4.6.2.	Runtime-Adapter für lokale Simulation	56
4.6.3.	Runtime-Adapter für verteilte Ausführung	56
4.6.4.	Registrierung eines Runtime-Adapters	57
5.	SML-to-SBP Mapping	59
5.1.	Mapping für ein Szenario	59
5.1.1.	Mapping für eine ModalMessage	61
5.1.2.	Mapping für eine Condition	62
5.1.3.	Mapping für eine Variable	62
5.1.4.	Mapping für einen Loop	63
5.1.5.	Mapping für eine Alternative	63
5.1.6.	Mapping für ein Parallel Fragment	64
5.2.	Mapping für eine Collaboration	65
5.3.	Mapping für eine Spezifikation	66
5.4.	Mapping für eine Runconfiguration	67
5.5.	Codegenerator für SML	67
6.	Methodik	69
6.1.	Einbinden von UI und anderer Software	69
6.2.	Einbinden einer Steuerung der Umgebung	70
6.3.	Überwachung des Nachrichtenverlaufes	71
6.4.	Bedeutung von Assumptions in SBP	72
6.5.	Debugging mit SBP	74
6.6.	Objektmodelle ohne EMF	76
6.7.	Objektmodelle aus EMF importieren	76
6.8.	Objekterzeugung und -zerstörung	77
6.9.	Testing	78
6.10.	Andere Protokolle für die verteilte Ausführung verwenden	81
7.	Implementierung	83
7.1.	Rollen und Bindings	83
7.2.	Nachrichten mit Rollen und Parametern	84
7.3.	System-Szenarien	85
7.4.	Parallel-Szenarien	87

7.5.	Einstellungen in SBP	88
7.5.1.	Einstellungen für das Logging	88
7.5.2.	Einstellungen für den Executor	89
7.5.3.	Einstellungen für Output-Streams	89
7.5.4.	Methoden für die Output-Streams	90
7.6.	Aktivierung von Szenarien	90
7.7.	Lifecycle von Szenarien	91
7.7.1.	Interpretation von Nachrichten in verteilten Komponenten	92
7.8.	Event-Selection in SBP	93
7.9.	Prozessprioritäten von Szenarien	96
7.10.	Verteilte Ausführung über das Netzwerk	97
7.10.1.	Architektur der verteilten Ausführung	98
7.10.2.	Senden einer Nachricht	98
7.10.3.	Empfangen einer Nachricht	99
8.	Evaluation und fortführende Ansätze	101
8.1.	Java Bibliothek	101
8.2.	Ausführung	102
8.3.	Entwicklungsmethodik	102
8.4.	Qualitätsaspekte	103
8.4.1.	Erweiterbarkeit	104
8.4.2.	Plattformunabhängigkeit	105
8.4.3.	Performance	105
8.4.4.	Skalierbarkeit	108
9.	Verwandte Arbeiten	115
9.1.	Scenario-based Programming mit PlayGo	115
9.2.	Codegenerierung von LSCs nach AspectJ	115
9.3.	Codegenerierung aus UML Sequenzdiagrammen	116
9.4.	Akka	116
10.	Fazit	117
10.1.	Zusammenfassung	117
10.2.	Ausblick	118
A.	Anhang	119
A.1.	Repository zum Projekt	119
A.2.	Download der Bibliothek	119
A.3.	Inhalt der DVD	119
A.4.	Verwendung des Codegenerators	120

Inhaltsverzeichnis

A.5. Car-To-X Spezifikation in SML	120
A.6. Car-To-X Klassendiagramm	125
A.7. Transformationsregeln-Henshin	125
Literaturverzeichnis	129

1. Einleitung

In der modernen Softwareentwicklung geht es immer mehr um verteilte reaktive Systeme, die oftmals eine dynamische Struktur aufweisen. Zu solchen Systemen zählen unter anderem autonome Fahrzeugsysteme, mobile Serviceroboter, Smart Factories und Car-To-X-Systeme. Diese Systeme kommunizieren über einen Nachrichtenaustausch miteinander, indem sie Ereignisse aus ihrer Umgebung über Sensoren empfangen und darauf mit dem Senden von weiteren Ereignissen reagieren. Diese Ereignisse modellieren wir als Nachrichten.

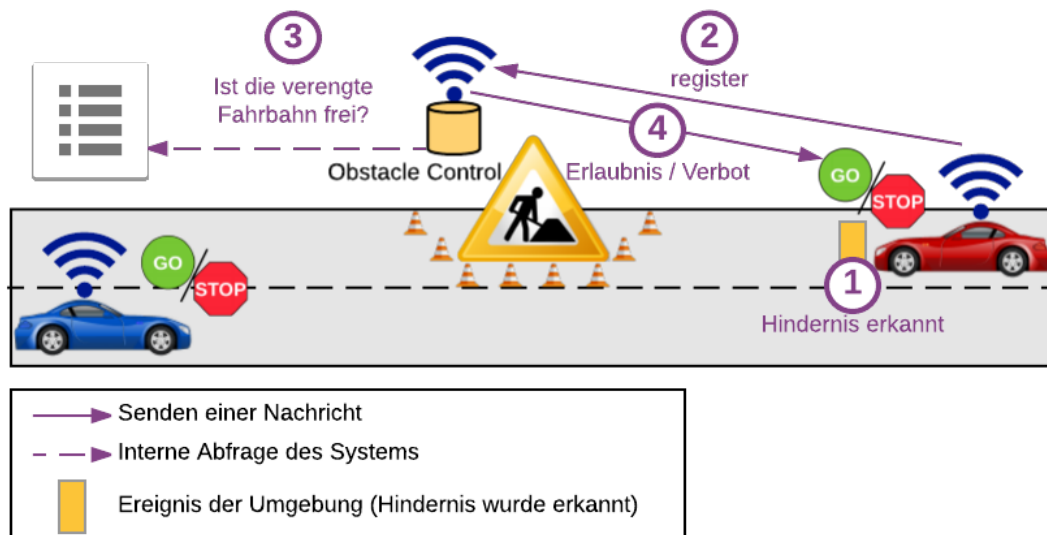


Abbildung 1.1.: Car-To-X Beispiel: Zwei Fahrzeuge nähern sich einer Baustelle, die eine Spur blockiert. Die Fahrzeuge registrieren sich bei einer Kontrollereinheit der Baustelle (*Obstacle Control*). Die Kontrollereinheit entscheidet daraufhin, welches Fahrzeug fahren darf.

Abbildung 1.1 zeigt eine Beispielsituation in einem Car-To-X System. Die beiden Fahrzeuge wollen die Baustelle, die eine Spur blockiert, passieren. Das Fahrerassistenzsystem soll den Fahrern zeigen, ob sie weiterfahren oder anhalten müssen, ohne auf Ampeln oder andere Verkehrszeichen angewiesen zu sein. Die Fahrzeuge registrieren sich dafür bei einer Kontrollereinheit (*Obstacle Control*), die die Baustelle verwaltet. Diese entscheidet dann, welches Fahrzeug fahren

1. Einleitung

darf. Die Kommunikation findet in Form eines Nachrichtenaustausches statt. In diesem Beispiel lernt die Kontrolleinheit die beiden Fahrzeuge durch die Nachricht *register* kennen und entscheidet je nachdem, ob die verengte Fahrbahn (*Narrow Passage*) gerade frei ist oder nicht, ob ein Fahrzeug fahren darf (*Go*) oder anhalten muss (*Stop*).

Die Struktur dieser Systeme muss sich wegen der Veränderlichkeit ihrer Umgebung und ihrer Komponenten dynamisch anpassen können. Auf das obige Beispiel bezogen heißt das, dass die Kontrolleinheit immer wieder mit verschiedenen Fahrzeugen kommuniziert und das Car-To-X System die Anordnung seiner Komponenten je nach Situation ändert. Diese Eigenschaften sorgen für sehr komplexe Anforderungen an solche Systeme, die die gesamte Entwicklung zu einer Herausforderung für den Ingenieur machen. Durch diese Komplexität können sich Widersprüche und Fehler in die Anforderungen einschleichen, die nur schwer zu entdecken sind. Oftmals werden diese Widersprüche oder Fehler erst spät bei der Entwicklung sichtbar. Das Beheben dieser Fehler kann im späteren Verlauf der Entwicklung sehr teuer und zeitaufwendig werden. Um dieser Problematik entgegenzuwirken, erfordert es präzise Methoden zur Modellierung, um die reaktiven Eigenschaften und dynamischen Strukturen erfassen und umsetzen zu können. So können Fehler und Widersprüche innerhalb der Anforderungen gefunden werden, bevor es zu einer Implementierung kommt.

Um diese Problematik zu lösen, gibt es den Ansatz der *szenariobasierten Modellierung*, der sich für die formale Modellierung und Analyse solcher Systeme in der Entwurfsphase eignet. Mit Hilfe bestimmter *domänenspezifischer Sprachen* (DSL) ist es so möglich, das Verhalten von Systemen szenariobasiert zu modellieren, indem Abläufe und speziell die Inter-Komponenten-Kommunikation mittels Szenarien beschrieben werden. Beispiele für solche DSLs sind die grafischen Sprachen der *Live Sequence Charts* [1] bzw. *Modal Sequence Diagrams* [6]. Als textuelle DSL bietet das Werkzeug SCENARIOTOOLS¹ die *Scenario Modeling Language* (SML)². Die mit diesen Werkzeugen hergestellten szenariobasierten Spezifikationen beschreiben die Anforderungen an das System formal in Form von vielen Szenarien, die zusammen das Systemverhalten beschreiben. Diese Szenarien unterscheiden zwischen System- und Umweltkomponenten, die untereinander Nachrichten austauschen. Dabei beschreiben sie Situationen, in denen gewisse Nachrichten erlaubt, gefordert oder verboten sind. Dies ermöglicht die feingranulare Aufteilung der Gesamtkomplexität eines Systems auf viele kleinere Teilfunktionalitäten. Szenarien sagen zwar einzeln nur wenig aus, dafür sind sie übersichtlich und kombiniert bilden sie das Gesamtsystemverhalten. Die

¹<http://scenariotools.org/>

²<http://scenariotools.org/scenario-modeling-language/>

Aufteilung des Systemverhaltens und die Beschreibung des Verhaltens zwischen Komponenten ermöglicht eine Abstufung und Unterteilung in Kategorien und erleichtern somit stark das Spezifizieren von komplexen Systemen.

Die so modellierten Szenarien können anschließend mittels *Play-Out* Algorithmen, basierend auf der Arbeit von Harel und Marelly [8], ausgeführt und simuliert werden. Unter Verwendung dieser Algorithmen kann SCENARIOTOOLS den Zustandsraum der Spezifikation entweder teilweise oder vollständig explorieren. Mithilfe dieser Simulation können Fehler und Wechselwirkungen zwischen Anforderungen aufgedeckt werden, bevor es zu der eigentlichen Implementierung des Systems kommt.

Für das Modellieren von reaktiven Systemen stehen also verschiedene Werkzeuge zur Verfügung. Jedoch reicht es nicht, ein System modellieren zu können, sondern es soll auch ein *Deployment* möglich sein, damit es auf dem Zielsystem ausführbar ist. Deployment bedeutet die Installation des modellierten Systemverhaltens auf dem Zielsystem. Abbildung 1.2 veranschaulicht, wie ein Deployment im Car-To-X Beispiel aussieht.

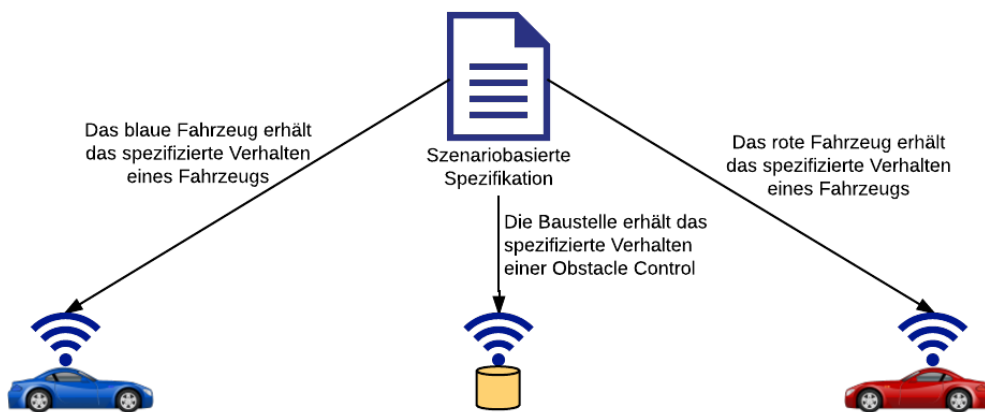


Abbildung 1.2.: Deployment der formalisierten Anforderungen auf dem Zielsystem.

Für SML Spezifikationen gibt es bisher keine Möglichkeit, die Spezifikation auf ein Zielsystem zu bringen, ohne die gesamte Werkzeugumgebung SCENARIOTOOLS mit dem Interpreter der DSL mit aufzuspielen. SCENARIOTOOLS bringt sehr viele Abhängigkeiten mit sich, die beim Deployment das Zielsystem belasten können.

Verteilte reaktive Systeme bestehen aus mehreren Komponenten. Also gibt es nicht das eine Zielsystem, sondern eine Reihe von Teilsystemen, die in der Spe-

1. Einleitung

zifikation unterschiedliche Rollen spielen. Ein verteiltes Deployment gestaltet sich für den Interpreter der formalen DSL also schwierig, da auf dem Zielsystem nicht einfach nur der Interpreter der lokalen Simulation laufen muss, sondern dieser zudem um die Verteiltheit der Komponenten erweitert werden müsste. Es reicht also nicht, die Spezifikation in ihrer domänenspezifischen Form auf das Zielsystem zu bringen, sondern sie muss implementiert werden.

Bei der Implementierung der Spezifikation geschieht allerdings ein methodischer Bruch. Zuerst werden die Anforderungen szenariobasiert modelliert, danach allerdings per klassischer Programmierung implementiert. In dieser Arbeit wird eine neuartige Form der Programmierung vorgeschlagen, die den Ansatz der szenariobasierten Modellierung nachbildet und somit diesen Bruch vermeidet. Abbildung 3.2 zeigt einen neuen Entwicklungsprozess, in dem es eine szenariobasierte Programmiermethodik gibt, die sich für eine Codegenerierung eignet.

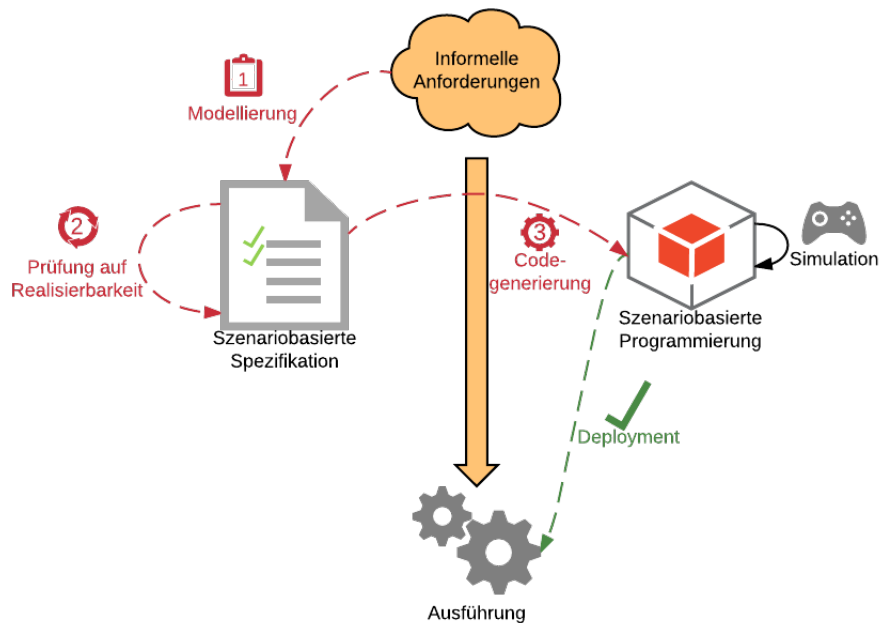


Abbildung 1.3.: Der neue Entwicklungsprozess von den informellen Anforderungen über die Formalisierung zur Ausführung. Die Pfeile zeigen den Ablauf der Entwicklung eines Softwaresystems.

Das *Behavioral Programming* (BP) [9] bietet Möglichkeiten, ein System verhaltensgesteuert zu implementieren. BP ermöglicht die Implementierung einer verhaltensgesteuerten Anwendung, die für jede Komponente des Systems einen

Thread vorsieht. Diese Threads synchronisieren sich an bestimmten Punkten über Nachrichten. BP kommt somit schon etwas näher an die Anforderungen eines reaktiven Systems heran. Jedoch beschreibt es jede Komponente als Ganzes und setzt somit den Fokus auf die einzelne Komponente. Die Inter-Komponenten-Kommunikation wird dadurch nur implizit beschrieben. BP stellt zwar einen verhaltensorientierten Ansatz dar, aber keinen szenariobasierten.

Das Ziel dieser Arbeit ist es, eine Methodik zur szenariobasierten Programmierung von verteilten reaktiven Systemen zu entwickeln. Dafür werden die Vorlage von BP und der Ansatz der szenariobasierten Spezifikation verschmolzen und eine Bibliothek implementiert, die es ermöglicht, die Programmierung von verteilten reaktiven Systemen in einer General Purpose Language - in diesem Fall Java - szenariobasiert zu gestalten. Durch diese Methodik ist es möglich, eine szenariobasierte Spezifikation auf verschiedenen Plattformen auszuführen. Ein weiteres Ziel ist es, die Ausführung verteilt durchzuführen. Dafür wird eine Netzwerkebene entwickelt, die dies möglich macht. Um eine Automatisierung des Entwicklungsprozesses zu ermöglichen, wurde nun ein Codegenerator implementiert, der von bereits existierenden SML-Spezifikationen eine entsprechende Java-Implementierung für die verteilte Ausführung realisiert.

Struktur der Arbeit

Diese Arbeit ist wie folgt strukturiert.

In Kapitel 1 wird die Motivation, das Ziel und die Struktur der Arbeit beschrieben.

Kapitel 2 erläutert grundlegende Konzepte und Begrifflichkeiten, die für das Verständnis dieser Arbeit notwendig sind.

In Kapitel 3 werden die Problemstellung, der Lösungsansatz und die Anforderungen genauer analysiert und beschrieben.

Kapitel 4 beschreibt die implementierte Bibliothek zur szenariobasierten Programmierung.

In Kapitel 5 wird die Übersetzung der Scenario Modeling Language in ausführbaren Java-Code beschrieben, der für das Deployment auf dem Zielsystem genutzt werden kann.

Kapitel 6 beschreibt die Methodik zur szenariobasierten Programmierung in Java. Hier werden verschiedene Patterns und empfohlene Methoden für den Umgang mit SBP vorgestellt.

In Kapitel 7 wird die Implementierung der SBP Library vorgestellt. Hier werden die verschiedenen Teile der Bibliothek, sowie Designentscheidungen vorgestellt.

1. Einleitung

In Kapitel 8 wird die szenariobasierte Programmierung in Java mit dem Interpreter von SCENARIOTOOLS verglichen. Zudem wird auf Skalierbarkeit und Limitierungen der SBP Bibliothek eingegangen.

Kapitel 9 beschreibt verwandte Arbeiten. Hier werden Arbeiten und Artikel vorgestellt, die sich mit den gleichen oder mit ähnlichen Themen befassen wie diese Arbeit.

In Kapitel 10 werden die Erfolge und Entwicklungen der Arbeit noch einmal zusammengefasst und es wird ein Fazit gezogen.

2. Grundlagen

In diesem Kapitel werden Grundlagen erläutert, die für das Verständnis dieser Arbeit nötig sind.

2.1. Verteilte reaktive Systeme

Verteilte reaktive Systeme sind heutzutage weit verbreitet. Dazu zählen unter anderem autonome Fahrzeugsysteme, mobile Serviceroboter, Smart Factories und Car-to-X Systeme. Diese Systeme bestehen aus mehreren Software- und Hardwarekomponenten, die auf der Ebene eines Nachrichtenaustausches kommunizieren und aufeinander reagieren. Diese softwareintensiven Systeme zeichnen sich dadurch aus, dass sie ständig aktiv sind und auf Stimuli ihrer Umwelt warten, diese verarbeiten und gegebenenfalls Daten weiterleiten oder Entscheidungen treffen. Stimuli können verschiedenste Formen haben. Im Falle eines Car-To-X Systems können es zum Beispiel Ereignisse des GPS oder der Annäherungssensorik des Fahrzeugs sein. Diese Ereignisse modellieren wir als Nachrichten. Abbildung 2.1 zeigt ein Beispiel eines Fahrzeugsystems, das eine Nachricht aus der Umwelt erhält, wenn es sich einer Baustelle nähert. Das Fahrzeugsystem kann dann basierend auf seinen Anforderungen entscheiden, was nach dem Erhalt dieser Nachricht zu tun ist. Im Beispiel 1.1 aus der Einleitung würde das Fahrzeug eine Nachricht an die Baustelle senden, um sich zu registrieren.

Eine besondere Eigenheit dieser Systeme ist ihre Verteiltheit. Verteilte reaktive Systeme bestehen oft aus vielen Komponenten, die in ständiger Interaktion zueinander stehen. Diese Komponenten müssen nicht fest miteinander verdrahtet sein, sondern können gegebenenfalls weit voneinander entfernt sein. Im Beispiel des Fahrzeugsystems sind die Baustelle und das Fahrzeug auf der Ebene der Hardware eigenständige Systeme. Auf der Ebene der Software allerdings verbinden sie sich zu einem Gesamtsystem, das aus verteilten Komponenten besteht. Diese Komponenten können zudem austauschbar sein. Fährt das Fahrzeug an der Baustelle vorbei und nimmt seinen normalen Kurs wieder auf, benötigt es den Kommunikationsweg zur Baustelle nicht mehr. Demnach kann das Fahrzeug die Baustelle wieder vergessen und die Baustelle kann sich wieder

2. Grundlagen

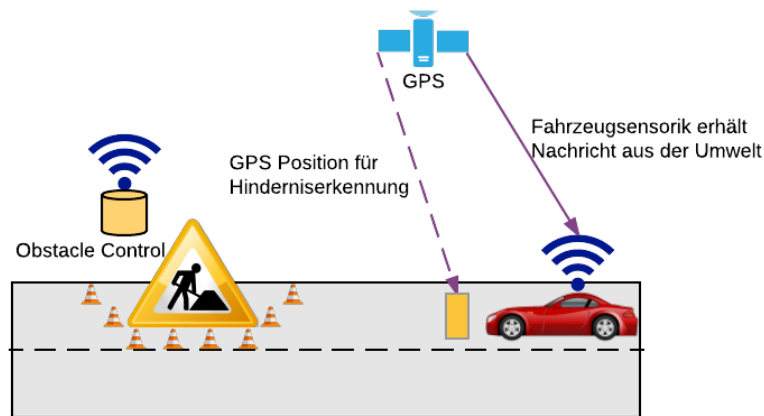


Abbildung 2.1.: Beispiel für Ereignis der Umgebung eines Fahrzeugs durch GPS. Für eine Baustelle ist der Gefahrenbereich mittels GPS-Koordinaten festgelegt. Wenn das Fahrzeug in den Bereich eintritt, wird es von seiner Sensorik (Umwelt) gewarnt.

um weitere Fahrzeuge kümmern. Dies bezeichnen wir als *dynamische Struktur* des verteilten reaktiven Systems.

2.2. Szenariobasierte Entwicklung

Die szenariobasierte Entwicklung ist die moderne Art der Entwicklung von softwareintensiven Systemen. Aufgrund der wachsenden Systemkomplexität heutiger Softwaresysteme ist es sinnvoll und nötig, das System schon bei der Entwicklung gründlich zu strukturieren und zu testen. Dies ist aufgrund der großen Menge an Anforderungen an das System leider oft schwierig. Die szenariobasierte Entwicklung bietet Verfahren, diese Anforderungen strukturiert und formal zu modellieren. Diese Verfahren stützen sich auf sogenannte Szenarien, die einen kleinen Teilaspekt des Systems beschreiben. Diese Teilaspekte sind meist eine einzelne Anforderung oder sogar nur ein kleiner Teil einer komplexeren Anforderung. Ein großer Vorteil von Szenarien ist ihre Strukturierbarkeit. Eine Anforderung kann so erst mal komplett in ein Szenario übertragen und später in kleine, wiederverwendbare oder mehrfach auftretende Teilszenarien aufgeteilt werden. Diese Szenarien folgen einer formalen Syntax und Semantik, wodurch es dem Ingenieur bereits bei der Modellierung möglich ist, das System zu testen. Es ist also möglich, die Funktionalitäten des modellierten Systems mit den

Anforderungen zu überprüfen.

2.3. ScenarioTools

SCENARIOTOOLS ist eine Werkzeug-Umgebung basierend auf der Entwicklungsplattform ECLIPSE. Sie bietet Plugins zur formalen Modellierung von Anforderungen an verteilte reaktive Systeme. Verwendet werden Modelle des *Eclipse Modeling Frameworks*, mit denen Klassenmodelle für ein gefordertes System erstellt werden können. Diese Modelle haben allerdings alleine noch kein Verhalten. SCENARIOTOOLS bietet Technologien, mit denen diesen Modellen ein erwünschtes Verhalten zugeordnet werden kann. Dafür bietet SCENARIOTOOLS eine domänenspezifische Sprache - die *Scenario Modeling Language* - mit der zu den Modellen sogenannte *Szenarien* definiert werden können. Diese Szenarien modellieren das Systemverhalten und werden in einer *SML-Spezifikation* gesammelt. Im Folgenden werden die Grundlagen der szenariobasierten Entwicklung mit SCENARIOTOOLS genauer beschrieben.

2.3.1. Modellierung der Systemkomponenten

Die Komponenten des Systems werden mittels eines Klassendiagramms modelliert. Bei der Entwicklung mit SCENARIOTOOLS wird hier das Eclipse Modeling Framework (EMF) verwendet. Das Klassendiagramm basierend auf dem Car-To-X Beispiel aus der Einleitung findet sich im Anhang A.6. Es ist modelliert in EMF Ecore, der grafischen Modellierungssprache des Eclipse Modeling Frameworks.

Abbildung 2.2 zeigt eine Ausprägung des Klassendiagramms bezüglich des Car-To-X Beispiels. Zur besseren Übersicht stellt die Abbildung nur Ausprägungen der Klassen Car, Obstacle, ObstacleControl und Lane dar. Zudem werden Beziehungen der Objekte in der Ausgangssituation dargestellt. Diese Abbildung wird im Folgenden als *Objektsystem* und die Ausprägungen als *Objekte* bezeichnet.

Dieses Objektsystem wird für die spätere Ausführung der SML-Spezifikation benötigt. Für ein System kann es - je nach Situation oder Anforderung - verschiedene Objektsysteme geben.

2.3.2. Modellierung des Systemverhaltens in SML

Nach der Aufstellung des Klassenmodells folgt die Modellierung des Systemverhaltens. SCENARIOTOOLS bietet dafür die Scenario Modeling Language (SML).

2. Grundlagen

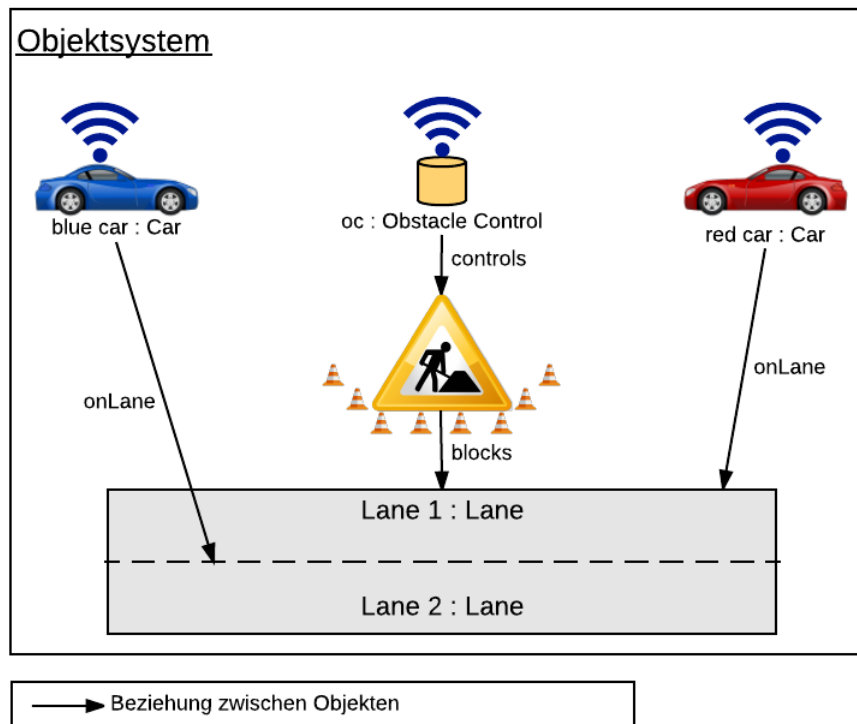


Abbildung 2.2.: Vereinfachte Darstellung der Objekte aus dem Car-To-X Beispiel als Objektsystem.

Mit dieser Sprache lassen sich Szenarien definieren, die den Nachrichtenaustausch zwischen den System- und Umweltkomponenten auf der Ebene von *Rollen* beschreiben. Diese Rollen sind auf die Modelle des Systems getypt und beschreiben damit einen Teil des Systems. Wir unterscheiden also Rollen, die Komponenten des Systems symbolisieren, und Objekte, die reale Ausprägungen der Systemkomponenten darstellen. Abbildung 2.3 beschreibt beispielhaft das Verhältnis zwischen Objekten und Rollen. Die Objekte des Objektsystems auf der rechten Seite spielen die Rollen der SML Spezifikation auf der linken Seite. Dabei ist speziell zu sehen, dass sowohl das blaue, als auch das rote Fahrzeug dieselbe Rolle spielen können, je nachdem in welcher realen Situation sie sich gerade befinden.

Der Codeausschnitt 1 zeigt einen Teil der Spezifikation des Car-To-X Beispiels. Die vollständige Spezifikation findet sich im Anhang A.5. In der ersten Zeile wird ein Klassenmodell importiert. Dies ist das Klassenmodell des Car-To-

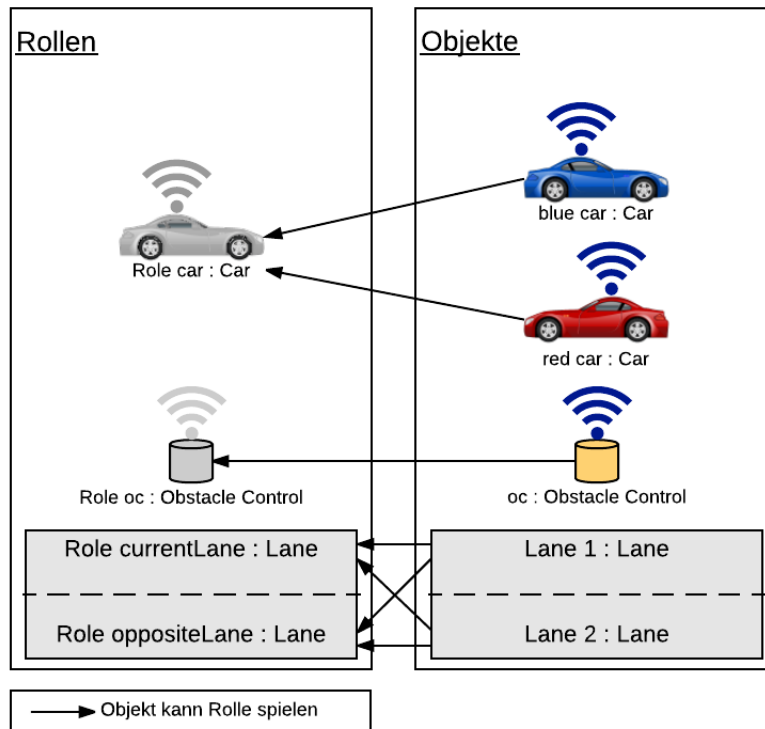


Abbildung 2.3.: Darstellung des Verhältnisses zwischen Objekten und Rollen. Auf der linken Seite sind Rollen der SML-Spezifikation dargestellt. Auf der rechten Seite sind die Objekte des Objektsystems ohne Beziehungen dargestellt. Die Pfeile zeigen an, welche Rolle ein Objekt spielen kann.

X Systems, dessen Verhalten modelliert werden soll. Danach folgt die Spezifikation selbst, die den Name CarToX trägt. Sie verlangt die Angabe eines Pakets aus dem Klassenmodell als Domäne. Die Klassen in dieser Domäne können dann als kontrollierbar (**controllable**) oder unkontrollierbar (**uncontrollable**) definiert werden. Ist eine Klasse kontrollierbar, so ist sie ein Teil des Systems und kann spezifiziert werden. Klassen, die unkontrollierbar sind, sind Teil der Umwelt, über die lediglich Annahmen getroffen werden können. Mit einer Spezifikation wird das Verhalten des Systems modelliert. Die Umwelt wird als gegeben angenommen und unterliegt gewissen Regeln, die als Annahmen für ihr Verhalten angegeben und geprüft werden können. Nur definierte Klassen können an Szenarien der Spezifikation teilnehmen. Nach der Definition der Klassen folgen

2. Grundlagen

die *Collaborations*. Dies sind Zusammenschlüsse von Rollen, die ein bestimmtes Interaktionsverhalten aufweisen. Innerhalb einer Collaboration können Rollen als **static** oder als **dynamic** definiert werden. Statische Rollen können lediglich an ein Objekt gebunden werden, wohingegen dynamische Rollen von verschiedenen Objekten gespielt werden können. Eine Rolle ist immer über eine Klasse des importierten Klassenmodells getypt und kann nur an Objekte dieser Klassen gebunden werden. Sind alle Rollen einer Collaboration definiert, kann die Modellierung der Szenarien beginnen. Im Folgenden werden die Bestandteile einer Spezifikation und deren Bedeutung genauer erklärt.

```
1 import "../model/cartox.ecore"
2
3 system specification CarToX {
4
5     // Refer to a package in the imported.ecore model.
6     domain cartox
7
8     // Define classes of objects that are controllable or uncontrollable.
9     define Car as controllable
10    define ObstacleControl as controllable
11    define Environment as uncontrollable
12    define Driver as uncontrollable
13    define Dashboard as uncontrollable
14    define Obstacle as uncontrollable
15    define LaneArea as uncontrollable
16    define StreetSection as uncontrollable
17
18    // Collaborations describe how objects interact in a certain
19    // context to collectively accomplish some desired functionality.
20    collaboration CarDriving {
21
22        dynamic role Environment env
23        dynamic role Car car
24
25        specification scenario CarMovesToNextArea {
26            message env -> car.movedToNextArea
27        }
28    ...
29 }
```

Code 1: Definition einer Car-To-X Spezifikation in SML. Die vollständige Spezifikation befindet sich im Appendix dieser Arbeit.

2.3.3. Szenario

Die Anforderungen an das System können formalisiert, gruppiert und unterteilt werden. Dies passiert in der Form von *Szenarien*. Ein Szenario beschreibt den Nachrichtenaustausch zwischen Rollen und damit das Systemverhalten. Der Codeausschnitt 2 zeigt ein Szenario aus der Car-To-X Spezifikation. In diesem Szenario wird definiert, was nach dem Erkennen des Hindernisses geschehen soll. Die erste Nachricht `setApproachingObstacle` heißt *initialisierend*, da sie diejenige ist, die für eine *Aktivierung* des Szenarios sorgt. Ist ein Szenario aktiviert, gibt es eine Kopie des Szenarios, welche zur Laufzeit Nachrichten fordern und

erwarten kann. Nur aktivierte Szenarien können den Nachrichtenablauf steuern. Da die Nachricht von der Rolle `env` an die Rolle `car` gesendet wurde, sind diese beiden Rollen bereits durch den Sender und Empfänger der Nachricht gebunden. Anschließend werden die Rollen `obstacle` und `obstacleControl` durch Einträge im Objektsystem gebunden. Das Schlüsselwort `bind` startet eine *Binding Expression*. Ein solcher Ausdruck definiert, woran eine Rolle für das Szenario gebunden werden soll. Die Rolle `obstacle` soll demnach an den Teilausdruck `car.approachingObstacle` gebunden werden. Dies ist eine Referenz der Klasse `Car`, bezogen auf das an die Rolle `car` gebundene Objekt. Ab diesem Zeitpunkt gilt das Szenario als *aktiviert*.

Ein aktiviertes Szenario definiert also den weiteren Ablauf des Nachrichtenaustausches. In dem Beispielszenario wird durch das Schlüsselwort `requested` als nächstes die Nachricht `register` gefordert. Wäre diese Nachricht nicht `requested`, würde sie erwartet werden. In beiden Fällen nennen wir diese Nachricht *enabled*. Durch das Schlüsselwort `strict` sind alle anderen Nachrichten, die in diesem Szenario definiert sind, streng verboten. Diese beiden Schlüsselwörter heißen *Modalitäten* und werden in Kapitel 2.3.5 weiter erklärt. Tritt diese Nachricht auf, ändert das Szenario seinen *Zustand*, indem es zur nächsten Nachricht weiter schreitet. Ein inaktives Szenario steht vor seiner initialisierenden Nachricht. Nach dem Auftreten dieser Nachricht, schreitet das Szenario weiter und steht vor der zweiten Nachricht. Ein Szenario ist solange aktiv, wie es noch vor einer Nachricht oder einer Bedingung steht. Sobald die letzte Nachricht des Szenarios aufgetreten ist, terminiert das Szenario.

```

1  specification scenario ControlStationAllowsOrDisallowsCarToEnterNarrowPassage
2  with dynamic bindings [
3    bind obstacle to car.approachingObstacle
4    bind obstacleControl to obstacle.controlledBy
5  ] {
6    message env -> car.setApproachingObstacle(*)
7    message strict requested car -> obstacleControl.register
8    alternative {
9      message strict requested obstacleControl -> car.enteringAllowed
10   } or {
11     message strict requested obstacleControl -> car.enteringDisallowed
12   }
13 }

```

Code 2: Beispiel eines SML-Szenarios, welches definiert, was nach dem Erkennen eines Hindernisses geschehen soll. Das Fahrzeug registriert sich zunächst bei der Obstacle Control, die zum Hindernis gehört. Anschließend wird entweder das Durchfahren erlaubt oder verboten.

Es gibt verschiedene Arten von Szenarien. Die zum Verständnis dieser Arbeit relevanten Typen von Szenarien sind die Folgenden:

- Das *Specification* Szenario beschreibt das Verhalten des zu modellieren-

2. Grundlagen

den Systems. Hier wird genau bestimmt, wie das System auf bestimmte Nachrichten zu reagieren hat. Diese Szenarien sind die wichtigsten in einer Spezifikation, da sie alleine das System modellieren. Bei einer Simulation des Systems übernehmen sie die Steuerung des Systems.

- Das *Assumption* Szenario beschreibt das Verhalten der Umwelt des Systems. Sie dienen im Grunde der Simulation der Umwelt bei der Ausführung. Diese Szenarien können keine Nachrichten vom System fordern, sondern beschreiben nur, wie die Umwelt auf Nachrichten reagiert. Bei einer Simulation des Systems können sie allerdings auch Fehler in den Annahmen für die Umwelt aufzeigen.

2.3.4. Violations und Nachrichtenverwaltung im Szenario

Ein Szenario kann nicht nur durch die letzte Nachricht terminiert, sondern kann schon vorher verletzt werden. Das kann passieren, wenn eine Nachricht auftritt, die im aktuellen Zustand des Szenarios verboten ist oder wenn eine Bedingung eine *Violation* explizit fordert. Es gibt drei Arten der Violation:

- *Interrupt-Violation*: Tritt eine Interrupt-Violation in einem Szenario auf, wird dieses direkt abgebrochen, ohne weitere Auswirkungen auf andere Szenarien und damit den Rest des System zu haben. Sie wird oft verwendet, um Szenarien mit Fallunterscheidungen herzustellen. So können gezielt Bedingungen gefordert werden, die in bestimmten Situationen ein Szenario als ungültig erklären.
- *Safety-Violation*: Eine Safety-Violation ist eine Art der Verletzung, die nicht auftreten **darf**. Tritt sie dennoch auf, bricht nicht nur das Szenario ab, sondern die gesamte Ausführung des Systems ist betroffen. Denn sie bedeutet, dass ein schwerer Fehler aufgetreten ist, der gegen das gewünschte Systemverhalten spricht. Es ist nun zu unterscheiden, ob der Fehler vom System oder von der Umwelt verursacht wurde. Tritt diese Violation in einem Specification-Szenario auf, ist es ein Fehler des Systems. Tritt sie in einem Assumption Szenario auf, ist es ein Fehler der Annahmen für die Umwelt.
- *Liveness-Violation*: Dies ist eine spezielle Art der Verletzung und kann nicht explizit gefordert werden. Sie tritt auf, wenn eine geforderte Nachricht **niemals** auftreten kann, da sie woanders für immer verboten bleibt.

Ob Nachrichten eine Violation auslösen, hängt vom Zustand des Szenarios ab. Jede Nachricht kann für jeden Zustand eines Szenarios einen der folgenden Zustände annehmen:

- *Ignored*: Die Nachricht taucht im Szenario nicht auf oder wird durch eine Bedingung des Szenarios als *ignored* deklariert. Diese Nachricht darf immer auftreten, ohne dass das Szenario darauf reagiert. Sie führt zu keiner Zustandsveränderung des Szenarios.
- *Requested*: Die Nachricht wird vom Szenario direkt mit dem Schlüsselwort **requested** gefordert (siehe Absatz 2.3.5). Sofern es eine Nachricht des Systems ist, muss sie ausgeführt werden, bevor die Umwelt wieder eine Nachricht senden kann (siehe Superstep 2.3.6).
- *Monitored*: Die Nachricht wird vom Szenario durch das Weglassen des Schlüsselwortes **requested** erwartet. Diese Nachricht soll zwar auftreten, muss aber durch ein anderes Szenario gefordert werden. Sie kann auch zu einem späteren Zeitpunkt ausgeführt werden, nachdem die Umwelt wieder Nachrichten gesendet hat (siehe Superstep 2.3.6).
- *Safety-Forbidden*: Eine Nachricht ist Safety-Forbidden, wenn sie im Szenario selbst vorkommt, aber nicht enabled ist. Dafür muss die Nachricht und damit der Zustand des Szenarios strict sein. Tritt diese Nachricht dennoch auf, wird das Szenario durch eine Safety-Violation unterbrochen.
- *Interrupt-Forbidden*: Eine Nachricht ist Interrupt-Forbidden, wenn sie im Szenario selbst vorkommt, aber nicht enabled ist. Dafür darf die Nachricht und damit der Zustand des Szenarios nicht strict sein. Tritt diese Nachricht dennoch auf, wird das Szenario durch eine Interrupt-Violation unterbrochen.

2.3.5. Bestandteile eines Szenarios

Ein Szenario besteht aus verschiedenen *Fragmenten*. Das wichtigste Fragment ist die *SML Nachricht*, die ein Ereignis im System modelliert. Dazu kommen weitere Fragmente, die im Folgenden beschrieben werden.

SML Nachricht

Die *SML Nachricht* ist das Kernelement eines Szenarios. Sie beschreibt eine Nachricht mit *Sender*, *Empfänger*, Name und Parametern. Der Sender einer Nachricht ist das Objekt, das das Ereignis ausgelöst hat. Der Empfänger ist

2. Grundlagen

demnach das Objekt, das das Ereignis empfangen hat. In der SML Nachricht sind diese Objekte allerdings durch Rollen repräsentiert, die an Objekte gebunden werden. Der Name der Nachricht ist eine Eigenschaft der Empfängers. Dies kann eine Operation oder der Setter eines Attributes bzw. einer Referenz sein. Parameter können beliebige Werte enthalten, die vom Typ Boolean, Integer, String oder Object sind. Der Parameter `*` bezeichnet einen Parameter, dessen Wert für das Szenario irrelevant ist. Die SML Nachricht hat zudem zwei Modalitäten.

- Sie kann *requested* sein. Dies bedeutet, dass die Nachricht vom Szenario angefordert und ausgeführt werden soll. Fehlt dieses Schlüsselwort, wird die Nachricht lediglich erwartet. Infolgedessen bleibt das Szenario an dieser Stelle stehen, bis die Nachricht irgendwann auftritt. Der Befehl zur Ausführung der Nachricht muss dann an einem anderen Punkt stattfinden. Sollte der Sender dieser Nachricht eine Komponente des Systems sein, ist die Nachricht eine Systemnachricht und muss noch im aktuellen Superstep auftreten. Passiert dies nicht, wird die Ausführung des Systems blockiert und es tritt eine Liveness-Violation auf.
- Sie kann *strict* sein. Dies bestimmt, wie schlimm eine Verletzung des Szenarios an dieser Stelle ist. Ist die Nachricht *strict*, so darf das Szenario an dieser Stelle **nicht** verletzt werden. Ist die Nachricht nicht *strict*, darf das Szenario durch eine Verletzung unterbrochen werden.

```
1 message strict requested car -> obstacleControl.register
```

Code 3: Beispiel einer SML-Nachricht. Die Rolle `car` sendet der Rolle `obstacleControl` die Nachricht `register`. Die SML-Nachricht ist *strict* und *requested*.

Loop

Ein *Loop* beschreibt eine sich wiederholende Abfolge von Ereignissen. Diese Ereignisse wiederholen sich, solange eine gewisse Bedingung erfüllt bleibt, oder das Szenario verletzt wird. Das Codebeispiel 4 zeigt einen solchen Loop in SML. Hier wird die Nachricht `movedToNextArea` unendlich oft ausgeführt, da die Bedingung des Loops immer wahr ist. Dieser Loop kann nur durch eine Verletzung des Szenarios unterbrochen werden. Ein Loop wird meist dafür verwendet, um einen gewissen Nachrichtenverlauf mehrfach zu durchlaufen, bis zum Beispiel eine Änderung des Objektsystems auftritt und die Bedingung des Loops nicht mehr erfüllt ist. Hiermit kann ein Ausdruck wie „Tue A solange, bis B erfüllt ist“ modelliert werden.


```

1 while [ true ] {
2   message requested env -> car.movedToNextArea
3 }

```

Code 4: Beispiel eines Loops in SML. Die Nachricht `movedToNextArea` wird durch die Bedingung `true` unendlich oft gefordert.

Alternative

Eine *Alternative* beschreibt eine Verzweigung des Szenarios in verschiedene mögliche Abfolgen von Ereignissen. Das Codebeispiel 5 zeigt eine solche Verzweigung. In diesem Beispiel verzweigt sich das Szenario in zwei Pfade. Im ersten ist die Nachricht `enteringAllowed` aktiv, im zweiten ist es die Nachricht `enteringDisallowed`. An dieser Stelle sind beide Nachrichten enabled. Welche Nachricht nun auftritt, hängt von weiteren Szenarien ab, die gleichzeitig aktiv sind. Wird der zu wählende Pfad nicht durch ein anderes Szenario vorgegeben, wird stets die erste Möglichkeit gewählt. Eine Alternative kann zudem für jeden Pfad eine Bedingung festlegen, die gültig sein muss, damit der Pfad gewählt werden kann.

```

1 alternative {
2   message strict requested obstacleControl -> car.enteringAllowed
3 } or {
4   message strict requested obstacleControl -> car.enteringDisallowed
5 }

```

Code 5: Beispiel einer Alternative in SML. Hier kann entweder die Nachricht `enteringAllowed` oder die Nachricht `enteringDisallowed` auftreten.

Parallel

Das Fragment *Parallel* verhält sich ähnlich wie eine Alternative. Auch hier verzweigt sich das Szenario. Anders ist allerdings, dass nicht nur ein Pfad gewählt werden kann, sondern alle Pfade gleichzeitig. Die verzweigten Pfade werden nun parallel abgearbeitet, bis alle Pfade ihr Ende erreichen. Codebeispiel 6 zeigt solch ein Parallel Fragment. In diesem Beispiel gibt es zwei Nachrichten, die durch das Parallel Fragment in beliebiger Reihenfolge auftreten dürfen. Es müssen jedoch beide auftreten, damit das Fragment abgearbeitet ist.

```

1 parallel {
2   message requested car -> car.doA
3 } and {
4   message requested car -> car.doB
5 }

```

Code 6: Beispiel eines Parallel Fragmentes in SML. Hier können die Beispielnachrichten `doA` und `doB` in beliebiger Reihenfolge passieren.

2. Grundlagen

Condition

Eine *Condition* ist eine Bedingung, die bei Erfüllung oder Nichterfüllung zu einer Verhaltensänderung eines Szenarios führt. Conditions können Teil einer Alternative oder eines Parallel Fragmentes sein. Hier entscheiden sie, welche Pfade während der Ausführung valide sind. Conditions können allerdings auch alleinstehend sein. Dann können sie die folgenden Formen annehmen:

- *Interrupt-Condition*: Diese Condition unterbricht ein Szenario durch eine Interrupt-Violation, wenn der Ausdruck der Bedingung *wahr* ist.
- *Violation-Condition*: Diese Condition unterbricht ein Szenario durch eine Safety-Violation, wenn der Ausdruck der Bedingung *wahr* ist.
- *Wait Until*: Diese Condition hält das Szenario solange auf, bis der Ausdruck der Bedingung *wahr* ist. Das Szenario bleibt vor der Bedingung stehen und sein Zustand ist **nicht** strict. Tritt also eine verbotene Nachricht auf, während das Szenario an dieser Stelle wartet, bricht das Szenario mit einer Interrupt-Violation ab.
- *Strict Wait Until*: Diese Condition hält das Szenario solange auf, bis der Ausdruck der Bedingung *wahr* ist. Das Szenario bleibt vor der Bedingung stehen und sein Zustand ist strict. Tritt also eine verbotene Nachricht auf, während das Szenario an dieser Stelle wartet, bricht das Szenario mit einer Safety-Violation ab.

In Codebeispiel 7 wird die Syntax der alleinstehenden Conditions vorgestellt.

```
1 // Ruft eine Interrupt-Violation auf, wenn die Bedingung erfüllt ist.
2 interrupt if [ laneArea.obstacle == null ]
3 // Ruft eine Safety-Violation auf, wenn die Bedingung erfüllt ist.
4 violation if [ laneArea.obstacle == null ]
5 // Das Szenario bleibt stehen, bis Bedingung erfüllt ist. Der Zustand ist nicht
   strict.
6 wait until [ laneArea.obstacle == null ]
7 // Das Szenario bleibt stehen, bis die Bedingung erfüllt ist. Der Zustand ist strict.
8 strict wait until [ laneArea.obstacle == null ]
```

Code 7: Beispiel verschiedener Conditions in SML Syntax.

Variable

In SML ist es möglich, *Variablen* zu deklarieren und zuzuweisen. Diese Variablen sind immer getypt. Mögliche Typen für Variablen sind Boolean, Integer, String und Object. Auch können Typen aus importierten Modellen, z.B. des

Systemmodells, verwendet werden. Das Codebeispiel 8 zeigt die Deklaration einer Variablen, sowie zwei Möglichkeiten einer Zuweisung. Die erste Möglichkeit der Zuweisung ist durch ein Gleichheitszeichen, wie in gängigen Programmiersprachen. Die zweite Möglichkeit hingegen ist es, der Variable mit Hilfe einer Nachricht einen Wert zuzuweisen. Hier kann durch das Schlüsselwort `bind to` der Wert eines Parameters an eine Variable gebunden werden.

```

1 // Variable value ist vom Typ Integer und wird mit 0 initialisiert.
2 var EInt value = 0
3 // Der Wert der Variable value wird auf 1 gesetzt.
4 value = 1
5 // Der Wert der Variable value wird um 1 erhöht.
6 value = value + 1
7
8 // Variable id ist vom Typ Integer.
9 var EInt id
10 // Der Wert der Variable id wird auf den Wert des Parameters der Nachricht gesetzt.
11 message obstacleControl -> car.setID(bind to id)

```

Code 8: Beispiel für die Deklaration und Zuweisung von Variablen in SML.

2.3.6. Ausführung von Szenarien

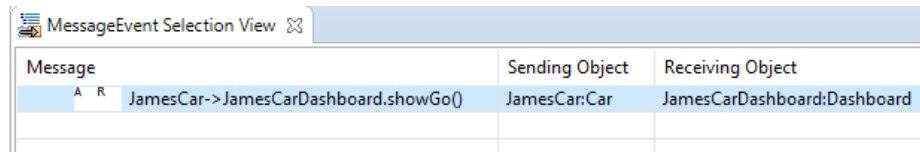
Nach der Modellierung des Systemverhaltens, muss es auf Korrektheit getestet werden. Dafür steht eine Methodik zu Verfügung, die sich *Play-Out* nennt. Playout wird von der ScenarioTools *Runtime* verwendet, um eine SML Spezifikation auszuführen. Die Grundlagen des Play-Out und der ScenarioTools Runtime werden im Folgenden erläutert.

ScenarioTools Runtime und Play-Out

Der *Play-Out* Algorithmus [8] ist als Methode zur Ausführung von Modal Sequence Diagrams [6] entstanden. In dieser Arbeit geht es jedoch um das Play-Out von ScenarioTools [2]. Play-Out ist eine Methode, mit der sich SML Spezifikationen in ScenarioTools ausführen lassen. Für das Ausführen einer Spezifikation sind drei Dinge nötig:

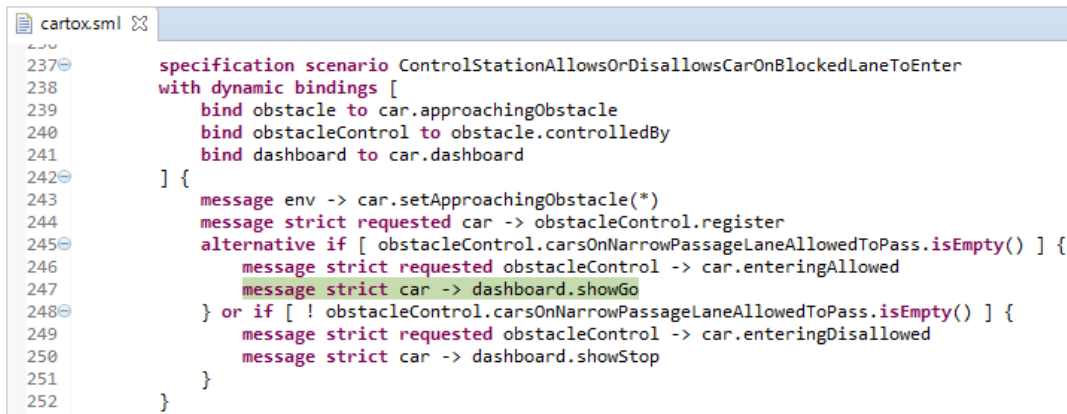
1. Die SML-Spezifikation: Eine Datei, in der sämtliche Collaborations mit den benötigten Szenarien enthalten sind.
2. Ein Objektsystem: Eine dynamische Instanz des Klassenmodells. Hier sind Objekte angelegt, die an der Ausführung teilnehmen sollen.
3. Eine Runkonfiguration: Eine Datei, die angibt, welches Objektsystem verwendet wird und welche Objekte welche Rollen spielen sollen.

2. Grundlagen



Message	Sending Object	Receiving Object
A R JamesCar->JamesCarDashboard.showGo()	JamesCar:Car	JamesCarDashboard:Dashboard

Abbildung 2.4.: Der MessageEvent Selection View im Play-Out Modus von ScenarioTools.



```
237 specification scenario ControlStationAllowsOrDisallowsCarOnBlockedLaneToEnter
238 with dynamic bindings [
239     bind obstacle to car.approachingObstacle
240     bind obstacleControl to obstacle.controlledBy
241     bind dashboard to car.dashboard
242 ] {
243     message env -> car.setApproachingObstacle(*)
244     message strict requested car -> obstacleControl.register
245     alternative if [ obstacleControl.carsOnNarrowPassageLaneAllowedToPass.isEmpty() ] {
246         message strict requested obstacleControl -> car.enteringAllowed
247         message strict car -> dashboard.showGo
248     } or if [ ! obstacleControl.carsOnNarrowPassageLaneAllowedToPass.isEmpty() ] {
249         message strict requested obstacleControl -> car.enteringDisallowed
250         message strict car -> dashboard.showStop
251     }
252 }
```

Abbildung 2.5.: Die SML Spezifikation mit Highlighting von aktiven Nachrichten im Play-Out Modus von ScenarioTools.

Mit diesen drei Dateien kann die Runtime gestartet werden. Im Play-Out Modus bietet ScenarioTools verschiedene Tools, um die Spezifikation zu analysieren. Startet die Ausführung, beginnt eine Simulation des Systems basierend auf dem Objektsystem. Im Ausgangszustand sind noch keine Szenarien aktiv und das Objektsystem befindet sich in seiner Ursprungsform. Nun kann der Benutzer die Steuerung der Simulation übernehmen, indem er Nachrichten auslöst. Im *MessageEvent Selection View* werden nun Nachrichten angezeigt, die im aktuellen Zustand enabled sind. Abbildung 2.4 zeigt den MessageEvent Selection View mit einer aktiven Nachricht.

Zum Analysieren der Szenarien werden die aktiven Nachrichten (enabled) in jedem Szenario der Spezifikation farblich markiert, wie in Abbildung 2.5 zu erkennen ist.

Zudem wird eine Auflistung aller aktiven Szenarien und aller Objekte angezeigt, durch die der Benutzer direkt deren Werte abfragen kann. Außerdem kann direkt auf die aktiven Nachrichten eines Szenarios gesprungen werden. Abbildung 2.6 zeigt diese Debuganzeige.

Wird nun ein Element dieser Debuganzeige ausgewählt, werden entsprechend

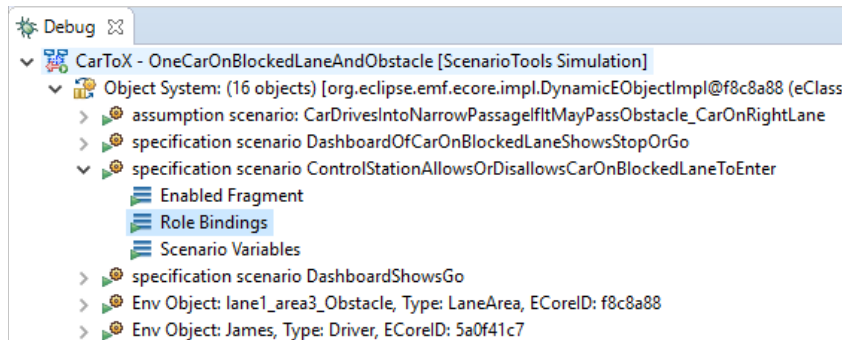


Abbildung 2.6.: Der Debug-View im Play-Out Modus von ScenarioTools. Er zeigt aktive Szenarien und an der Simulation teilnehmende Objekte.

Name	Value
env	env
obstacle	obstacle
obstacleControl	obstacleControl
dashboard	JamesCarDashboard
car	JamesCar

Abbildung 2.7.: Der Variable View im Play-Out Modus von ScenarioTools. Er zeigt die teilnehmenden Rollen im Szenario und die Objekte an, an die die Rollen gebunden sind.

die Werte des Elements in der Variable Anzeige aufgelistet. In Abbildung 2.7 werden die teilnehmenden Rollen eines Szenarios und die Objekte, an die sie gebunden sind, in dieser Anzeige aufgelistet.

Während der Benutzer verschiedene Nachrichten auswählt, wird ein *State Graph* erzeugt, der den bisherigen Pfad des Nachrichtenverlaufs darstellt. Durch ihn kann der Benutzer außerdem in einen früheren Zustand der Simulation zurückkehren, um einen anderen Pfad einzuschlagen. Abbildung 2.8 zeigt einen solchen State Graph.

Unifizierung von Nachrichten und Ereignissen

Nachrichten sind ein Modell für Ereignisse innerhalb des Systems. Während der Laufzeit werden diese Nachrichten interpretiert und mit auftretenden Ereignissen verglichen. Diesen Vergleich nennen wir Unifizierung. Um zu ermitteln, ob eine Nachricht mit einem Ereignis unifizierbar ist, überprüfen wir deren Eigen-

2. Grundlagen

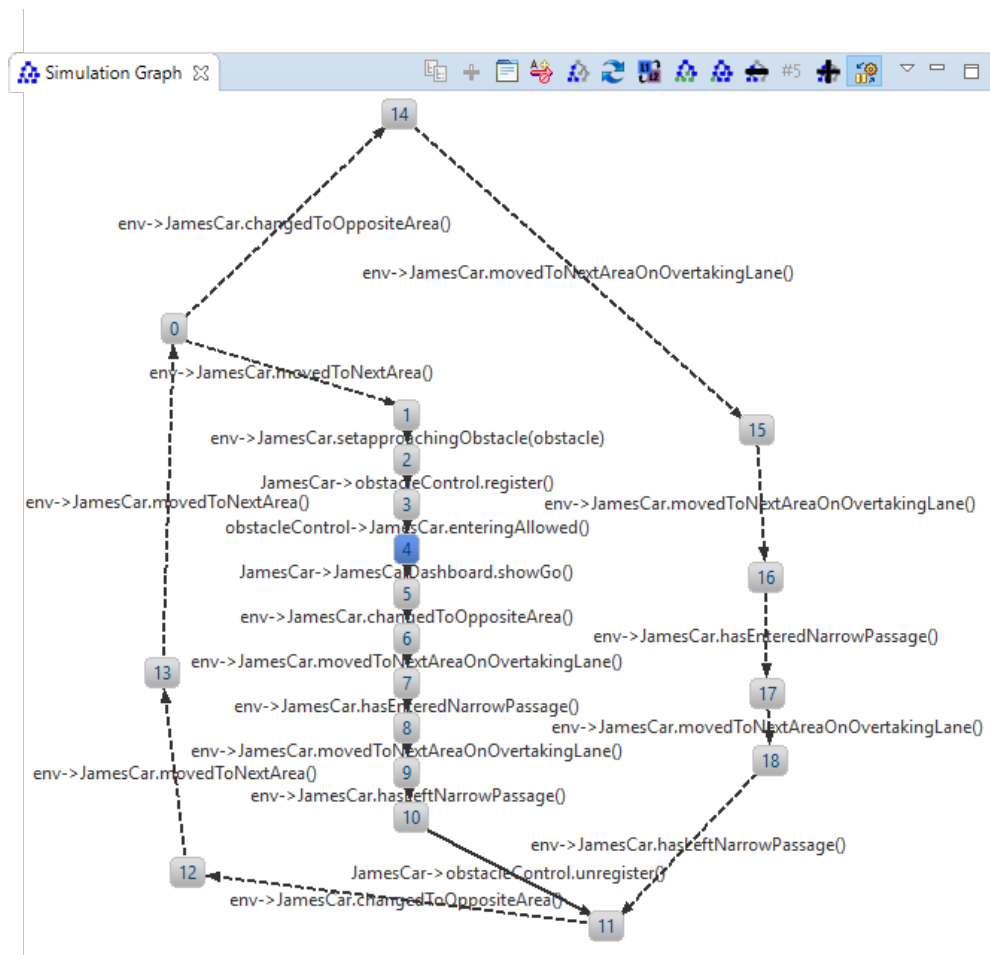


Abbildung 2.8.: Der State Graph View im Play-Out Modus von ScenarioTools. Er zeigt die erforschten Pfade des Zustandsgraphen der Simulation an.

schaften. Dazu zählen der Sender, der Empfänger, der Name und die Parameter der Nachricht bzw. des Ereignisses. Stimmen diese Eigenschaften überein, sind sie unifizierbar. Wir können außerdem Nachrichten miteinander unifizieren, um während der Ausführung Szenarien mit gleichen Nachrichten zu identifizieren. Wir definieren, dass in einem System Ereignisse auftreten, die wir mit Nachrichten modellieren. Diese Ereignisse können mit Nachrichten unifiziert werden. Vereinfacht sagen wir auch, dass eine Nachricht auftreten kann. Damit ist gemeint, dass ein als Nachricht modelliertes Ereignis auftritt.

Auswahl der ausführbaren Nachrichten

Die Auswahl der ausführbaren Nachrichten in der ScenarioTools Runtime ist ein mehrstufiges Verfahren. Abbildung 2.9 zeigt ein Aktivitätsdiagramm dieses *Event Selection Verfahrens*. Zuerst werden aus allen aktiven Specification Szenarien die Nachrichten, die requested sind, in einer Liste gesammelt. Nun können zwei Fälle auftreten:

1. **Nachrichten vorhanden:** Wenn die Liste Nachrichten enthält, also mindestens eine Nachricht requested ist, ist das System an der Reihe, eine Nachricht zu senden. Dann werden die verbotenen Nachrichten aus den Szenarien von dieser Liste gelöscht. Sollte die Liste dann immer noch Nachrichten enthalten, sind das die Nachrichten, die ausführbar sind. Ist die Liste jedoch leer, ist das System zwar an der Reihe, ist aber blockiert. Dadurch ist die Spezifikation verletzt. Es tritt eine Liveness-Violation auf.
2. **Keine Nachrichten vorhanden:** Wenn die Liste keine Nachrichten enthält, also in keinem Szenario eine Nachricht requested ist, ist die Umwelt an der Reihe eine Nachricht zu senden. Als nächstes werden dann Umweltnachrichten aus aktiven Szenarien gesammelt und mit den Umweltnachrichten vereint, die weitere Szenarien initialisieren. Danach werden aus der neuen Liste die Nachrichten gelöscht, die durch aktive Szenarien verboten sind. Was übrig bleibt, sind die ausführbaren Nachrichten.

Superstep

Unter dem Begriff *Superstep* ist eine Reihe von Ereignissen zu verstehen, die ohne Unterbrechung hintereinander ausgeführt werden. Dies tritt bei der Ausführung von Systemnachrichten auf. Abbildung 2.10 zeigt beispielhaft solch einen Superstep. Die Systemnachrichten zwischen den Umweltnachrichten werden zusammen als Superstep bezeichnet. Alle Systemnachrichten, die nacheinander requested werden, müssen innerhalb eines Supersteps ausgeführt werden.

2. Grundlagen

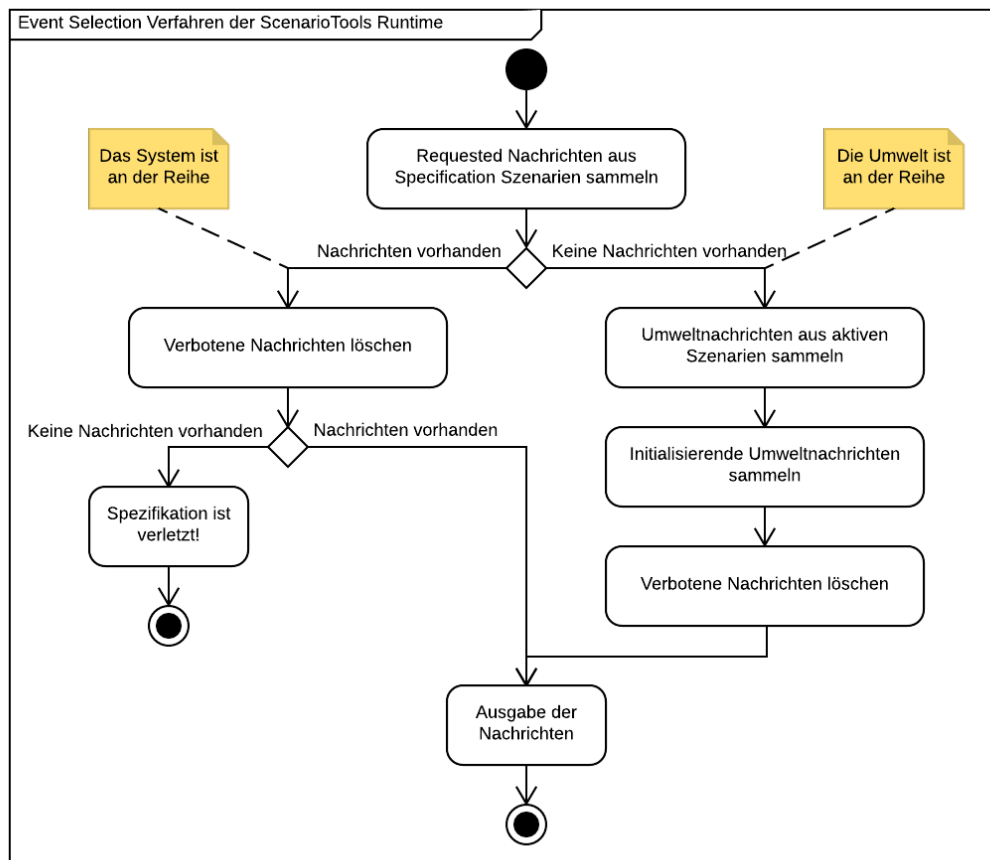


Abbildung 2.9.: Aktivitätsdiagramm zur Veranschaulichung des Event Selection Verfahrens in der ScenarioTools Runtime.

Sollte das nicht möglich sein, da eine Nachricht dauerhaft blockiert ist, kann der Superstep nicht beendet werden und es tritt eine Liveness-Violation auf.

2.4. Deployment

Das *Deployment* ist das Portieren einer formalen Spezifikation in einer ausführbaren Form auf die Zielmaschine. Ein Deployment kann auf verschiedene Weise geschehen:

- **Portieren des Interpreters:** Für eine Modellierungssprache mit Interpreter kann ein formal spezifiziertes Softwaresystem auf die Zielmaschine

gebracht werden, indem der Interpreter portiert wird. Die Zielmaschine interpretiert ihr eigenes Verhalten dann durch die formale Spezifikation. Je nachdem, wie umfangreich der Interpreter ist und wie viele Abhängigkeiten er mitbringt, kann er die Zielmaschine allerdings stark beeinträchtigen. Außerdem kann es vorkommen, dass ein Interpreter auf der Zielmaschine seine Arbeitsweise ändern muss, was unter Umständen zu einem großen Umbau des Interpreters führen kann.

- **Übersetzung in Code:** Alternativ können die formalen Modelle des Systems in ausführbaren, plattformspezifischen Code übersetzt werden. Beispielsweise können EMF Ecore Modelle in Javacode übersetzt werden, der auf der Zielmaschine ohne Interpreter ausgeführt werden kann. Diese Übersetzung der formalen Modelle in eine General Purpose Language (GPL) - wie Java - benötigt einen Übersetzer, der einmalig aus den Modellen Code generiert. Danach ist der Code beliebig auf der Zielmaschine einsetzbar. Dadurch gibt es zwar eine höhere Last bei der Entwicklung, da ein Übersetzer entwickelt werden muss, jedoch ist die Zielmaschine von der Last des Interpreters befreit und läuft im besten Fall deutlich

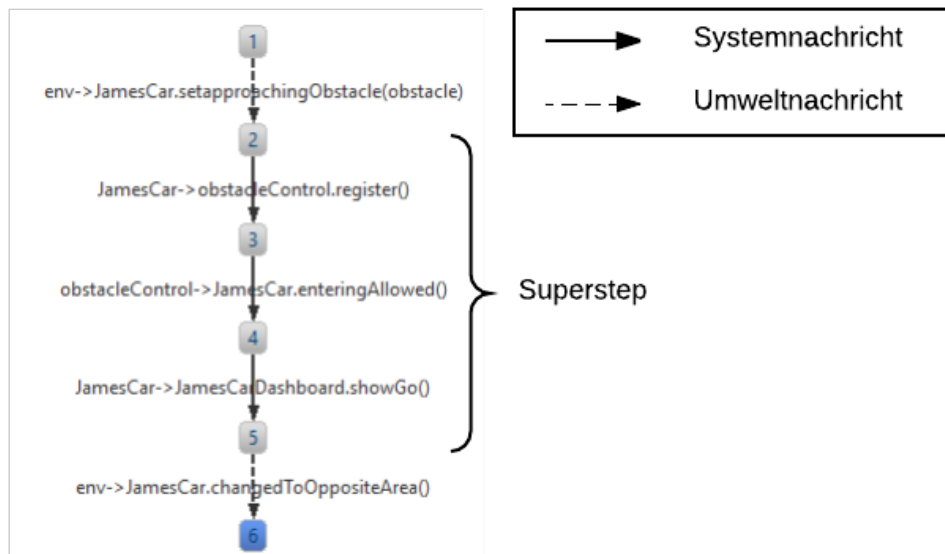


Abbildung 2.10.: Beispiel eines Supersteps. Die Nachrichten 2-4 sind zusammen ein Superstep der Ausführung des Systems.

2. Grundlagen

schneller.

2.5. Verteilte Ausführung

Anders als bei simplen Desktopanwendungen, die meist auf einem Computer laufen, sind reaktive Systeme deutlich komplexer. Die Zielmaschine ist meist ein Mehrkomponentensystem. Das bedeutet, dass die modellierte Software nicht nur auf eine Komponente deployed werden muss, sondern auf mehrere Komponenten verteilt wird. Im Falle eines Car-To-X Systems können die Komponenten in beliebiger Häufigkeit und in verschiedensten Variationen auftreten. Dabei müssen die verteilten Komponenten so arbeiten, als wären sie eine Einheit, wie in einer vorherigen Simulation. Denn die Spezifikation geht davon aus, dass die Komponenten als Einheit funktionieren. Bekommen die verteilten Komponenten nun alle die selbe Spezifikation, können sie damit wenig anfangen, da sie nicht wissen *wer* sie sind. Die Komponenten müssen die Spezifikation also aus unterschiedlichen Blickwinkeln betrachten - ihren Rollen entsprechend. Dadurch müssen sie die Spezifikation unter Umständen unterschiedlich interpretieren, was das Portieren des Interpreters auf die Zielsystemkomponenten oft erschwert. Die verteilte Ausführung einer Spezifikation bedeutet also ein angepasstes Deployment der Spezifikation auf verschiedenen Komponenten des Systems.

2.6. Behavioral Programming

Behavioral Programming (BP) ist eine Methodik zur Programmierung von verhaltensgesteuerten Anwendungen in einer General Purpose Language. Dabei wird eine *Behavioral Application* (B-Application) entwickelt, die aus mehreren unabhängigen Komponenten besteht. Diese Komponenten werden als *B-Threads* implementiert. Diese B-Threads implementieren das Verhalten der Komponenten des Systems, indem jeder B-Thread eine Komponente darstellt. Sie laufen parallel zu anderen B-Threads in der Anwendung und generieren eine Reihe von *Events*. Diese Events symbolisieren Nachrichten innerhalb des Systems. Über sie findet eine Synchronisation der B-Threads statt. Erreicht ein B-Thread einen Synchronisationspunkt, so wartet er solange an dieser Stelle, bis alle anderen B-Threads an dem selben Punkt in ihrem Programmfluss angelangt sind. Ist ein Synchronisationspunkt erreicht, erklärt der B-Thread drei Listen von Events:

1. **Requested Events:** Die Liste der Events, die der B-Thread ausführen will. Nur durch diese Liste können Events gefordert werden.

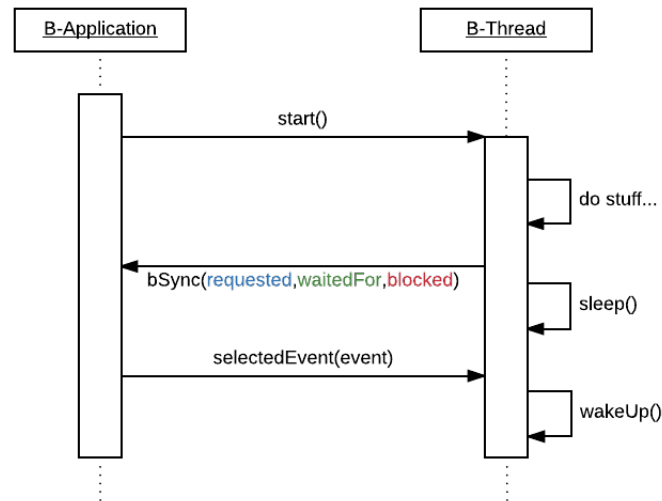


Abbildung 2.11.: Workflow eines B-Threads. Die B-Application startet einen B-Thread. Daraufhin beginnt der normale Programmfluss des B-Threads, bis ein Synchronisationspunkt erreicht ist. Der B-Thread ruft den Befehl *bsync* auf, womit er drei Listen von Events deklariert. Sobald die B-Application ein passendes Event auswählt, wacht der B-Thread wieder auf.

2. **Waited For Events:** Die Liste der Events, die vom B-Thread erwartet wird. Durch diese Liste kann kein Event gefordert, sondern lediglich beobachtet werden.
3. **Blocked Events:** Die Liste der Events, die an diesem Synchronisationspunkt nicht ausgeführt werden dürfen. Diese Events sind für die Ausführung an diesem Punkt verboten und können nicht für die Ausführung ausgewählt werden.

Sobald alle B-Threads diese drei Listen an die B-Application weitergegeben haben, warten sie darauf, dass diese ein Event für die Ausführung auswählt. Sobald dies geschehen ist, erwachen die B-Threads wieder, die dieses Event gefordert, bzw. auf dieses Event gewartet haben. Sie nehmen ihren Programmfluss wieder auf, bis sie den nächsten Synchronisationspunkt erreichen. Abbildung 2.11 zeigt den Workflow eines B-Threads.

2. Grundlagen

2.6.1. B-Thread-Beispiel

In ihrem Artikel [9] beschreiben Harel, Marron und Weiss das Beispiel einer Flusskontrolle für die Wärmeregulierung von Wasser. Das vorgestellte System besteht aus zwei Wasserleitungen (warmes und kaltes Wasser). Das durchgeleitete Wasser vermischt sich. Dabei können die Leitungen allerdings nur einzeln angesteuert werden. Sie stellen die drei B-Threads aus Codebeispiel 9 vor, die dennoch für eine gleichmäßige Zufuhr von warmem und kaltem Wasser sorgen. Der erste B-Thread `AddHotThreeTimes` steuert die Warmwasserzufuhr und leitet drei mal warmes Wasser in das System. Der zweite B-Thread `AddColdThreeTimes` tut dasselbe für die Kaltwasserzufuhr. Die Methode `bsync` sorgt für eine Synchronisation und die Ausführung von Ereignissen. Der erste B-Thread fordert (requested) das Ereignis `addHot`, erwartet und blockiert nichts. Der zweite B-Thread tut dasselbe mit dem Ereignis `addCold`. Damit nun eine gleichmäßige Wasserzufuhr ermöglicht wird, stellen die Autoren einen dritten B-Thread `Interleave` vor. Dieser läuft endlos und ruft abwechselnd zwei Synchronisationen auf. Als Erstes erwartet er `addHot` und blockiert `addCold`. Danach wechseln die beiden Ereignisse. Fordern tut dieser B-Thread nie etwas, er dient lediglich der Verwaltung der anderen B-Threads. Durch diese abwechselnde Blockade wechseln sich auch die ersten beiden B-Threads bei ihrer Ausführung ab und es entsteht eine geregelte Warm-Kalt-Zufuhr.

```
1 // B-Thread für warmes Wasser
2 class AddHotThreeTimes extends BThread {
3     public void runBThread() {
4         for (int i = 1; i <= 3; i++) {
5             bp.bSync( addHot, none, none );
6         }
7     }
8 }
9
10 // B-Thread für kaltes Wasser
11 class AddColdThreeTimes extends BThread {
12     public void runBThread() {
13         for (int i = 1; i <= 3; i++) {
14             bp.bSync( addCold, none, none );
15         }
16     }
17 }
18
19 // B-Thread für die Regulierung
20 class Interleave extends BThread
21     public void runBThread() {
22         while (true) {
23             bp.bSync( none, addHot, addCold );
24             bp.bSync( none, addCold, addHot );
25         }
26     }
27 }
```

Code 9: Warm-Kalt-Beispiel für die Funktionsweise von B-Threads.
Entnommen aus Harel et al. [9].

2.7. Code-Generierung und Xtend

Code-Generierung ist ein Prozess, bei dem ein Modell aus einer Quellsprache in Code einer Zielsprache übersetzt wird. Ein Beispiel dafür ist die Code-Generierung von EMF Ecore. Mit EMF Ecore lassen sich formale Modelle von Klassendiagrammen erzeugen, von denen aus anschließend Javacode generiert werden kann. Speziell im Bereich der domänenspezifischen Sprachen sind Code-Generatoren häufig anzutreffen. Modelle von domänenspezifischen Sprachen sind leicht validierbar und können deshalb gut für das Modellieren und Testen von Softwaresystemen verwendet werden. Anschließend kann aus diesen plattformunabhängigen Modellen plattformabhängiger Code generiert werden, der anschließend auf dem Zielsystem installiert wird.

Eine in dieser Arbeit verwendete Software zur Code-Generierung ist das Eclipse-Plugin *Xtend*. Es bietet eine Sprache um Modelle einzulesen und daraus mit Hilfe von hierarchischen Patterns Code zu generieren. Das Codebeispiel 10 zeigt einen Ausschnitt des Codegenerators, der in dieser Arbeit implementiert wurde. Hier wird eine Alternative von SML in Javacode für eine Alternative in zwei Schritten generiert.

```

1  def String generateAlternativeHead(Alternative alternative)'''
2  // Begin Alternative
3  List<Message> requestedEvents = new ArrayList<Message>();
4  List<Message> waitedForEvents = new ArrayList<Message>();
5  <FOR alternativeCase : alternative.cases>
6  <IF alternativeCase.caseCondition == null>
7  if (true) {
8  <ELSE>
9  if (<alternativeCase.caseCondition.generate>) {
10 <ENDIF>
11 <alternativeCase.caseInteraction.generateUntilFirstMessageAlternative>
12 }
13 <ENDFOR>
14 // Insert into BP
15 doStep(requestedEvents, waitedForEvents);'''
16
17 def String generateAlternativeBody(Alternative alternative)'''
18 // Determine which path has been chosen
19 <FOR alternativeCase : alternative.cases SEPARATOR "else">
20 <val m = InteractionUtil.getInitializingMessages(alternativeCase.caseInteraction)
21   .get(0)>
22   if (getLastEvent().equals(new Message(<m.sender.name>, <m.receiver.name>, "<m.
23     modelElement.modifiedName>"<m.parameters.generateParameters>))) {
24     <alternativeCase.caseInteraction.generateAfterFirstMessage>
25   }
26 <ENDIFOR>
27 // End Alternative'''

```

Code 10: Codeausschnitt der Implementierung des Codegenerators in Xtend.

2. Grundlagen

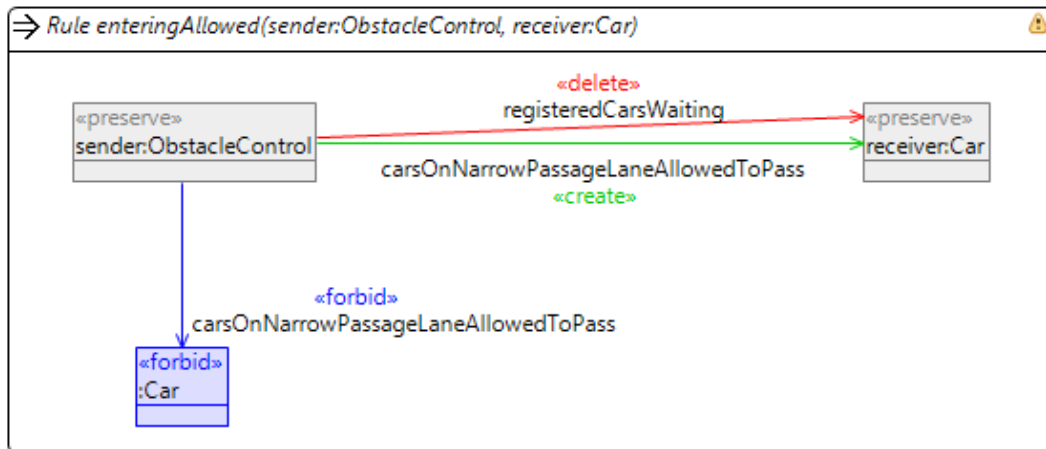


Abbildung 2.12.: Henshin-Regel für die Nachricht `enteringAllowed`. Die Kästen symbolisieren Objekte. Die roten Pfeile zeigen Relationen, die gelöscht werden. Grüne Pfeile zeigen Relationen die hinzugefügt werden. Blaue Elemente zeigen ein Verbot.

2.8. Objektsystemtransformation mit Henshin

Henshin ist ein Werkzeug für Modelltransformationen. Es ist als Eclipse-Plugin implementiert und basiert auf dem Eclipse Modeling Framework. Mit der Hilfe von *Henshin* ist es möglich, grafische Regeln für die Transformation von Model-Instanzen anzugeben. Diese Regeln können angewendet werden, um eine Model-Instanz auf eine bestimmte Weise zu transformieren. In *SCENARIOTOOLS* verwenden wir diese Methode für die Transformation des Objektsystems. Wir können Regeln so definieren, dass sie auf Nachrichten abgebildet werden können. Wenn eine Nachricht auftritt, wird die passende Regel auf das Objektsystem angewendet. Abbildung 2.12 zeigt eine Henshin-Regel, die für die Car-To-X Spezifikation verwendet wird. Sie wird angewendet, wenn die Nachricht `enteringAllowed` von einer Obstacle Control zu einem Fahrzeug gesendet wird. Dadurch wird das Fahrzeug aus der Liste `registeredCarsWaiting` gelöscht und in die Liste `carsOnNarrowPassageLaneAllowedToPass` ergänzt. Die Regel darf nur dann nicht angewendet werden, wenn sich bereits ein Fahrzeug in der zweiten Liste befindet. Das wird durch das blaue Element bestimmt. Gibt es ein Objekt an der Stelle einer blauen Relation, so kann die Regel nicht angewendet werden. Die Car-To-X Spezifikation enthält mehrere solcher Regeln, die in Anhang A.7 angegeben sind.

3. Anforderungsanalyse

Im Folgenden wird die Problemstellung, der Lösungsansatz und die Vision dieser Arbeit analysiert und eine Reihe von Anforderungen erhoben.

3.1. Problemanalyse

Das in dieser Arbeit adressierte Kernproblem ist der methodische Bruch zwischen der Modellierung und der Implementierung von verteilten reaktiven Systemen. Diese Systeme werden immer präsenter und fordern aufgrund der Komplexität ihrer Anforderungen immer genauere Verfahren zur Modellierung und zur Implementierung. Die bisher in der Einleitung und in den Grundlagen vorgestellten Werkzeuge nehmen sich zwar durch ihren szenariobasierten Ansatz der Problematik an, scheitern jedoch am letzten Schritt der Entwicklung. Dieser Schritt ist die verteilte Ausführung auf dem Zielsystem, da hierfür immer noch eine Implementierung in klassischer Form nötig ist. Das liegt einerseits an der Menge ihrer Abhängigkeiten, wie grafischer Editoren oder komplizierter Laufzeitinterpreter, und andererseits daran, dass oft einfache Programmierparadigmen fehlen. Dadurch sind die Möglichkeiten, eigenen Code in diese Werkzeuge einzubetten, stark eingeschränkt. Es fehlt die Verschmelzung des szenariobasierten Ansatzes von SCENARIOTOOLS mit der Flexibilität von Javacode. Außerdem kann es speziell für Programmierer oder Softwareentwickler, die nicht aus der Domäne der modellbasierten oder modellgetriebenen Entwicklung kommen, schwierig sein, diese Werkzeuge und die damit verbundenen Sprachen zu erlernen. Dazu kommen weitere Probleme, die mit der Plattform der Werkzeuge zusammenhängen. Die meisten Werkzeuge werden nur auf einer bestimmten Plattform angeboten und der Anwender muss zwangsläufig auf dieser Plattform entwickeln.

Aktuelle Lösungsmöglichkeiten für die Ausführung von szenariobasierten Spezifikationen sind bekannte Layout- und Syntheseverfahren, die bereits normaler Bestandteil der szenariobasierten Entwicklung sind. Ein großes Problem dieser Verfahren und damit der szenariobasierten Modellierung ist es, eine verteilte Ausführung der Spezifikation auf dem Zielsystem zu realisieren. SCENARIOTOOLS bietet mit der Scenario Modeling Language die Möglichkeit, das Ver-

3. Anforderungsanalyse

halten von Systemen zu modellieren und dann lokal zu simulieren. Die Ausführung auf mehreren Komponenten bzw. Computern hingegen, zum Beispiel im Netzwerk oder über Internetprotokolle, war bisher nicht möglich. Dazu gibt es zwar bereits theoretische Ansätze in SCENARIOTOOLS, um mit SML hergestellte Spezifikationen verteilt auszuführen [4], die jedoch noch nicht umgesetzt sind. Bisherige Versuche der Umsetzung (siehe [3]) waren zwar vorhanden, aber nicht zufriedenstellend.

Die bisherigen Lösungsmöglichkeiten lösen besonders die folgenden Probleme der verteilten Ausführung noch nicht vollständig: Zum Einen besteht das Problem der Synchronisation der Objektsysteme jeder Systemkomponente. Die lokalen Objektsysteme der Teilsysteme müssen global für das Gesamtsystem synchron gehalten werden, damit der Systemzustand global konsistent bleibt. Außerdem wird die Unterscheidung von System und Umwelt schwieriger, da verteilte Systemkomponenten auch untereinander wie mit Umweltkomponenten interagieren müssen. Dieser Sachverhalt lässt sich speziell in ScenarioTools noch nicht modellieren. Zudem besteht das Problem der Skalierbarkeit des Systems. Besonders bei sehr großen Systemen, wie Car-to-X Systemen, ist es nicht möglich, dass alle Komponenten alle anderen Komponenten und deren Ereignisse kennen. In diesem Fall müsste jedes Fahrzeug alle anderen Fahrzeuge auf der Welt kennen, was aus Sicht der Skalierbarkeit nicht praktikabel ist.

Ein weiteres erhebliches Problem ist die Portierung der SCENARIOTOOLS Werkzeugumgebung oder des Interpreters der Sprache SML, wenn es um die Ausführung auf dem Zielsystem geht. Durch die große Last an Abhängigkeiten, die ein solch komplexes Analysewerkzeug mit sich bringt, ist es schwer, dieses auf jedweder beliebigen Plattform zu installieren. Stattdessen muss hier eine manuelle Übersetzung der formalen Spezifikation zu ausführbarem Code passieren. Es fehlt an dieser Stelle eine Art Zwischenebene, die eine direkte Übersetzung von SML in ausführbarem Code ermöglicht.

Es muss eine Lösung her, die leichtgewichtig und plattformunabhängig ist. Außerdem ist die Erweiterbarkeit der Lösung wichtig, damit sie auf beliebigen Systemen für die verteilte Ausführung praktikabel ist. Die Lösung muss es außerdem ermöglichen, eigenen, plattformspezifischen Code in die Spezifikation einfließen zu lassen. Es muss eine Möglichkeit geschaffen werden, unter Verwendung bestimmter Protokolle eine verteilte Ausführung auf dem Zielsystem durchzuführen.

3.2. Lösungsansatz

Eine Lösung dieser Problematik bietet eine leichtgewichtige Java Bibliothek, welche plattformunabhängig das Spezifizieren von dynamischen reaktiven Systemen in einer bekannten *General Purpose Language* (GPL) ermöglicht. Damit sollen sich die Konzepte der szenariobasierten Modellierung einem domänenfremden Entwickler erschließen, ohne dass er spezielle Werkzeuge erlernen muss.

In dieser Arbeit wird eine Methodik für Java entwickelt, welche Programmierparadigmen bereitstellt, um die Implementierung von szenariobasierten Spezifikationen ohne die Notwendigkeit spezieller domänenspezifischer Sprachen oder Werkzeuge zu ermöglichen. Es werden Patterns zur Spezifikation von Szenarien bereitgestellt und eine Methodik entwickelt, wie dieses *szenariobasierte Programmieren* (SBP) in Java benutzt werden kann. Im Kern dieser Methodik wird die BP Bibliothek verwendet, welches das Herstellen von verhaltensgesteuerten Anwendungen nach dem Konzept des Behavioral Programming ermöglicht. Des Weiteren wird ein Protokoll entwickelt, das eine verteilte Ausführung der Spezifikation ermöglicht.

Mit dieser Methodik soll es möglich sein, Spezifikationen, bestehend aus Szenarien, in Java zu definieren. Szenarien beschreiben das Inter-Komponenten-Verhalten des Systems und der Umwelt auf der Ebene eines Nachrichtenaustausches nach dem Vorbild von SML. Damit soll eine Simulation der Spezifikation mit einem Objektsystem als Eingabe möglich sein. Die Auswahl und Ausführung von Nachrichten wird von der BP Bibliothek übernommen. Zudem soll es eine Möglichkeit geben, die Umwelt zu steuern und so Nachrichten in das System einzuspeisen. Das ist für die Einbindung von Sensorik oder für eine simulierte Ausführung notwendig. Es soll eine Möglichkeit geben, die Simulation verteilt durchzuführen. Außerdem soll ein Deployment des so implementierten Systems möglich sein. Unter Verwendung eines Codegenerators soll aus bestehenden SML Spezifikationen Code für die SBP Bibliothek generiert werden, der das Deployment und die verteilte Ausführung der Spezifikation ermöglicht.

Abbildung 3.1 zeigt, wo dieser Lösungsansatz in den vorher beschriebenen Entwicklungsverlauf eingreift. Zu sehen ist, dass es nun einen alternativen Pfad von den informellen Anforderungen zum Deployment über SBP gibt. Zudem gibt es die Möglichkeit, von der formalen Spezifikation Code für SBP zu generieren.

3. Anforderungsanalyse

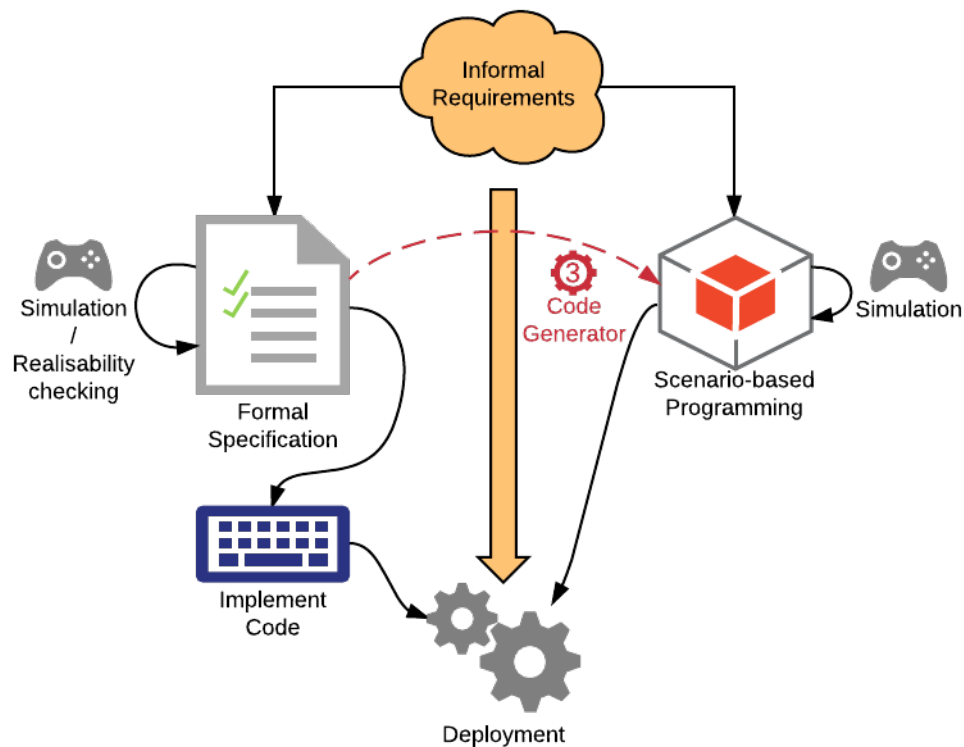


Abbildung 3.1.: Der Weg von den informellen Anforderungen über die Formalisierung zum Deployment. Die schwarzen Pfeile zeigen den aktuellen Ablauf der Entwicklung eines Softwaresystems. Der rote Pfeil zeigt einen alternativen Weg über SBP.

3.3. Vision

Das Ziel der Arbeit ist es, den Weg von der formalen Spezifikation in SML zum Deployment so zu verkürzen, dass keine zusätzliche Implementierung der Spezifikation mehr nötig ist. Abbildung 3.2 zeigt den neuen Weg von den informellen Anforderungen zum Deployment, der über eine formale Spezifikation und einer generierten Implementierung in SBP führt. Die manuelle Implementierung der Spezifikation fällt nun weg. Nach der Analyse der formalen Spezifikation wird durch einen Codegenerator SBP Code erzeugt, der direkt für ein Deployment verwendet werden kann.

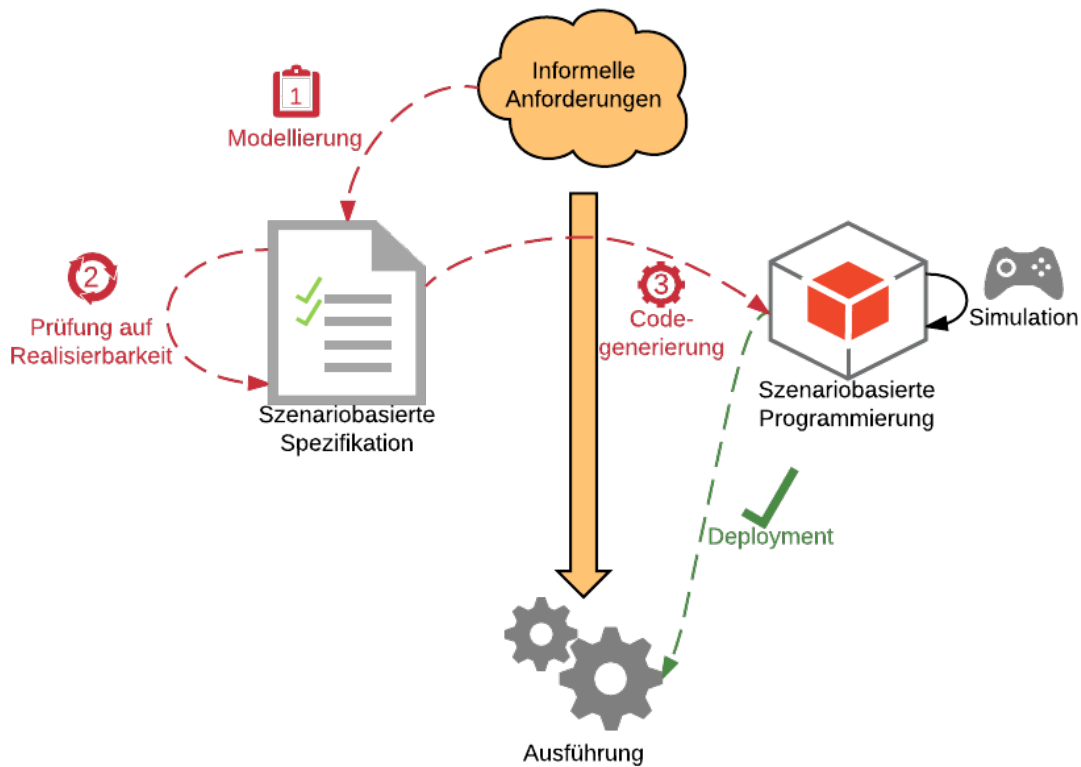


Abbildung 3.2.: Der neue Weg von den informellen Anforderungen über die Formalisierung zum Deployment über SBP. Die roten Pfeile zeigen den neuen Ablauf der Entwicklung eines Softwaresystems mit SBP. Der Grüne Pfeil impliziert ein einfaches Deployment der Spezifikation ohne weitere Implementierung der Spezifikation.

Das erspart viel Aufwand und Zeit, da eine zusätzliche Implementierung der Anforderungen wegfällt. So können SML Spezifikationen einfach auf Zielsystemen simuliert und getestet werden. Es gibt zudem die Möglichkeit, ein System direkt in Java szenariobasiert zu entwickeln, ohne zusätzliche Werkzeuge zu verwenden.

3.4. Anforderungen

Nach dem Vorbild der Scenario Modeling Language soll eine Java Bibliothek entwickelt werden, die die Konzepte der szenariobasierten Spezifikationen in leichtgewichtigen Javacode umsetzt. Das bedeutet, es soll eine Spezifikation

3. Anforderungsanalyse

geben, die durch Szenarien das Verhalten eines Systems beschreibt. In diesen Szenarien wird das Inter-Komponenten-Verhalten auf der Ebene eines Nachrichtenaustausches implementiert. Ebenso werden die Konzepte von Rollen, multi-modalen Nachrichten und Violations umgesetzt. Im Folgenden werden die Anforderungen an diese Arbeit aufgestellt. Wir unterteilen die Anforderungen in vier Themengebiete. Das erste beschreibt Anforderungen an die SBP Bibliothek. Die zweite umfasst Anforderungen der Ausführung des Codes. Das vierte Thema befasst sich mit der Methodik für die Herangehensweise an SBP. Das letzte Thema fasst Qualitätsaspekte an die Bibliothek zusammen. Abbildung 3.3 zeigt die Anforderungen noch einmal als Zieldiagramm.

1. Anforderung

Es soll eine Java Bibliothek zur Einbindung in Java Projekte entwickelt werden, die szenariobasiertes Programmieren in Java ermöglicht.

1.1 Anforderung

Als Basis der Bibliothek soll die Java Bibliothek von BP verwendet werden.

1.2 Anforderung

Die Bibliothek orientiert sich an den Konzepten der Scenario Modeling Language. Dazu gehören im Speziellen die Konzepte von Szenarien, Rollen und Spezifikationen.

1.3 Anforderung

Während der Ausführung der Spezifikation muss plattformspezifischer Code verwendet werden können. Dafür muss es eine Methode geben, diesen Code in die Spezifikation oder in die Ausführung einfließen zu lassen.

1.4 Anforderung

Für eine verteilte Ausführung auf dem Zielsystem soll die Umwelt von der Sensorik des Systems gesteuert werden. Außerdem sollen auch Aktoren durch Nachrichten oder Szenarien angesprochen werden können. Dafür müssen passende Schnittstellen entworfen werden.

2. Anforderung

Es soll eine Methode geben, die Java-Spezifikation auszuführen. Durch die Spezifikation und die Szenarien soll eine lauffähige, szenariobasierte Anwendung entstehen.

2.1 Anforderung

Eine mit der Bibliothek entwickelte Spezifikation soll lokal als Simu-

lation ausgeführt werden können. Dafür müssen alle Komponenten von einer Maschine kontrolliert werden können.

2.2 Anforderung

Die Spezifikation soll verteilt auf mehreren Komponenten ausgeführt werden können. Dafür müssen die Komponenten synchronisiert der Spezifikation folgen.

2.3 Anforderung

Für die verteilte Ausführung muss es eine Kommunikation zwischen den Komponenten geben, damit der Nachrichtenaustausch und die Synchronisation möglich sind.

2.4 Anforderung

Die Simulation muss sich vom Benutzer steuern lassen. Das bedeutet, es muss eine Möglichkeit geben, von außen Nachrichten der Umgebung in die Ausführung einzuspeisen.

3. Anforderung

Es soll eine Methodik für die Herangehensweise an die szenariobasierte Programmierung erarbeitet werden. Dafür sollen Verfahren und Patterns vorgelegt werden.

3.1 Anforderung

Es soll eine Möglichkeit geben, diese Methodik für das Deployment bestehender SML-Spezifikationen zu verwenden. Dafür muss ein Schema zur Übersetzung (Mapping) von SML nach SBP entwickelt werden.

3.2 Anforderung

Für die Entwicklung von Software ist das Testen unabdingbar. Daher soll es auch in SBP eine Methode zum Testen der SBP Anwendung geben.

3.3 Anforderung

In Programmcode können sich immer Fehler einschleichen. Es ist wichtig, eine sinnvolle Methode des Debuggens zu haben, um diese Fehler finden zu können.

4. Anforderung

Jede Software unterliegt gewissen Qualitätsanforderungen, so auch die SBP Bibliothek.

4.1 Anforderung

Lesbarkeit von Code ist wichtig für das Verständnis. Daher soll der

3. Anforderungsanalyse

Code eine ähnliche Struktur aufweisen, wie SML. Das betrifft ebenso wichtige Schlüsselwörter der Sprache.

4.2 **Anforderung**

Damit SBP für eine Vielzahl von Projekten verwendet werden kann, ist Erweiterbarkeit sehr wichtig. Es muss die Möglichkeit gegeben sein, die Bibliothek zu erweitern, um neue Protokolle und Funktionalitäten zu ergänzen.

4.3 **Anforderung**

SBP soll nicht an eine Toolplattform gebunden sein, wie SCENARIO-TOOLS. Eine SBP Anwendung soll ohne viel Aufwand auf verschiedenen Plattformen lauffähig gemacht werden können.

4.4 **Anforderung**

SBP soll auch für komplexe Anwendungen verwendbar sein. Dafür ist die Performance der Ausführung zu beachten. Bei ausführbarem Code ist die Laufzeit als besser anzunehmen, als bei der Interpretation von Modellen, wie bei SCENARIOTOOLS.

4.5 **Anforderung**

Verteilte reaktive Systeme können aus vielen Komponenten bestehen. Dafür muss die Anwendung auch für eine große Anzahl von Komponenten funktionsfähig sein. Dies gilt es zu überprüfen.

3.4. Anforderungen

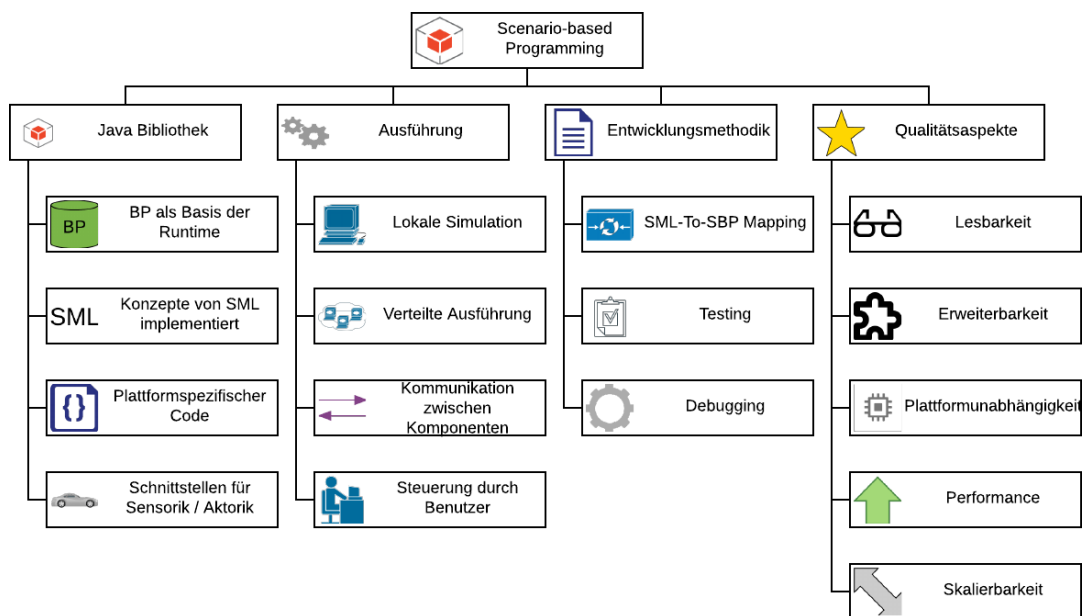


Abbildung 3.3.: Zieldiagramm für diese Arbeit.

3. Anforderungsanalyse

4. Szenariobasierte Programmierung in Java

In diesem Kapitel wird eine neue Programmiermethodik - die *Szenariobasierte Programmierung* (SBP) - vorgestellt und ein Überblick über die in dieser Arbeit entstandene Java Bibliothek gegeben. Szenariobasiertes Programmieren ist eine neue Programmiermethodik, die die Ansätze der szenariobasierten Modellierung in die General Purpose Language Java integriert. SBP verspricht den methodischen Bruch zwischen der Vorgehensweise der Modellierung und der Implementierung zu vermeiden. Es beinhaltet ein Verfahren und verschiedene Patterns zur szenariobasierten Gestaltung des Systemverhaltens in Java. Wir entwickeln Szenarien in Java als Teil einer szenariobasierten Anwendung, für die es eine spezifische Ausführungsmethodik gibt.

Das szenariobasierte Programmieren ist stark an die Methodik zur Modellierung des Systemverhaltens in ScenarioTools angelehnt. Ein Großteil der hierarchischen Eigenschaften von Spezifikationen, Collaborations und Szenarien aus SML sind übernommen worden. Aus der Sicht der Szenarien steht SBP der Modellierungssprache SML in nichts nach, da die in SML enthaltenen Sprachfeatures auch in SBP durch Patterns nachgebildet werden können. Mit SBP soll es einem Ingenieur möglich sein, SML Spezifikationen in Java zu übersetzen oder das System direkt szenariobasiert in Java zu entwickeln. Das verkürzt den Weg von der Entwicklung des Systemverhaltens zur Ausführung auf dem Zielsystem um den Schritt der Implementierung.

Die SBP Bibliothek kann in ein Javaprojekt eingebunden und verwendet werden, um szenariobasiertes Programmieren zu ermöglichen. Dafür stellt die Bibliothek verschiedene Basisklassen bereit, die von der Modellierungssprache SML inspiriert sind. Diese Klassen können verwendet werden, um Szenarien im Sinne von SML zu entwickeln, die dann in einer Spezifikation zusammengefasst und ausgeführt werden können. Die SBP Bibliothek baut auf der Bibliothek des Behavioral Programming auf. BP bietet eine Methode, Komponenten zu programmieren, die sich untereinander synchronisieren. SBP erweitert diese so, dass sich nicht mehr einzelne Komponenten synchronisieren, sondern stattdessen die Synchronisation zwischen Szenarien stattfindet. Abbildung 4.1 stellt die Erweiterung von BP grafisch dar. Die grünen Elemente stellen die Komponen-

4. Szenariobasierte Programmierung in Java

ten von BP dar, wohingegen die blauen Elemente die Erweiterungen darstellen. Die Erweiterungen sind die Folgenden:

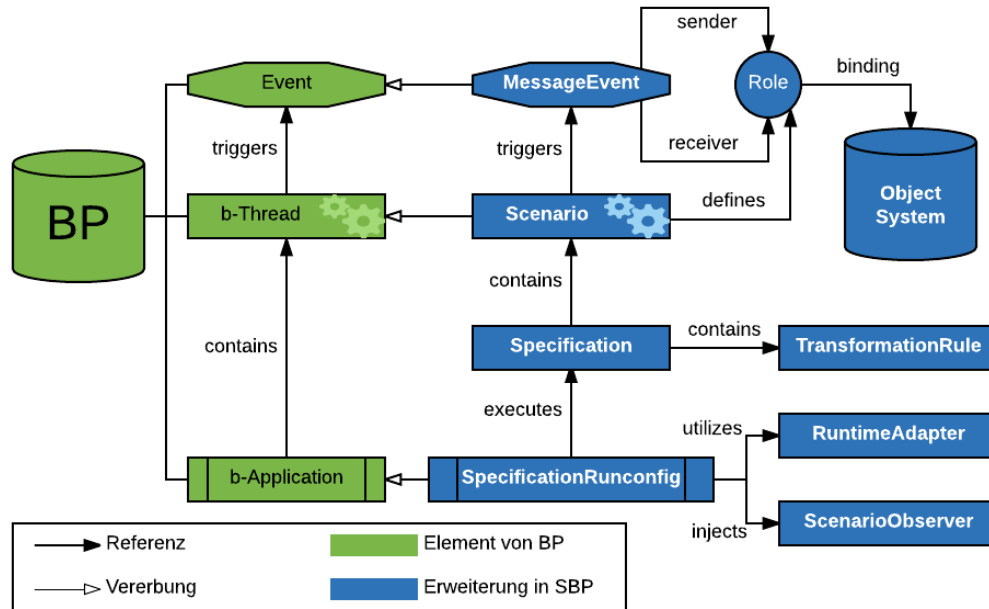


Abbildung 4.1.: Schema der Komponenten von SBP als Erweiterung von BP. Die grünen Elemente symbolisieren Komponenten von BP und die blauen Elemente stehen für die Erweiterung der Komponenten. Das violette Element steht für eine Netzwerkkomponente.

- **Message:** Eine Klasse, die die Klasse Event der BP Bibliothek erweitert. Das Event in BP ist ein einfaches Objekt, welches für die Verwendung im Sinne der modalen Nachrichten in SML um einen Sender und einen Empfänger erweitert werden muss. Der Sender und der Empfänger sind dabei vom neuen Typ Role.
- **Role:** Die Klasse Role beschreibt das Konzept der Rollen in SML. Sie symbolisiert einen Teil des Systems, dem durch die Szenarien ein gewisses Verhalten zugeordnet wird. Rollen können an Objekte aus einem Objektsystem gebunden werden.
- **ObjectSystem:** Das Objektsystem ist eine Klasse, die sich um das Sammeln von Objektreferenzen kümmert. Alle dort registrierten Objekte können an Rollen gebunden werden.

- **Scenario:** Die Klasse Scenario hat die selbe Funktion des Szenarios aus SML. Innerhalb dieser Klasse wird das Verhalten gewisser Komponenten des Systems definiert. Dabei werden die Komponenten als Rollen abstrahiert.
- **Specification:** Die Klasse Specification dient zur Strukturierung der Anwendung. In ihr werden Szenarien und Transformationsregeln registriert, die zur Laufzeit verwendet werden sollen.
- **TransformationRule:** Die TransformationRule ist eine Klasse zur Veränderung des Objektsystems als Reaktion auf bestimmte Nachrichten. Sie hört auf eine Nachricht und führt daraufhin Operationen auf dem Objektsystem aus.
- **SpecificationRunconfig:** Die Klasse SpecificationRunconfig dient der Ausführung der Spezifikation. Sie erweitert die BApplication von BP um eine einfache Methode, die Szenarien einer Spezifikation zu starten. Des Weiteren wird in dieser Klasse das Objektsystem für die Ausführung registriert.
- **RuntimeAdapter:** Der RuntimeAdapter verwaltet die Weitergabe und den Empfang von Nachrichten. Er ist für die tatsächliche Kommunikation zwischen den Systemkomponenten zuständig und ermöglicht eine verteilte Ausführung.
- **ScenarioObserver:** Eine spezielle Klasse zur Beobachtung von Szenarien. Observer können einem Szenario beigefügt werden und werden über den Fortschritt des Szenarios in Kenntnis gesetzt. Dadurch kann eine gewisse Reaktion auf eine Nachricht, das Aktivieren oder Deaktivieren eines Szenarios durchgeführt werden. Beispielsweise kann eine Benutzeroberfläche (GUI) aktualisiert oder die Motorik eines Roboters gesteuert werden.

Im Folgenden werden die Komponenten der Bibliothek vorgestellt. Zu den Bedeutendsten zählen die abstrakten Hauptklassen `Specification`, `Scenario` und `SpecificationRunconfig`, sowie die Klassen `Role` und `Message`, die innerhalb der Scenario-Klassen benötigt werden.

4.1. Java Spezifikation

Die Spezifikation wird durch die Java Klasse `Specification` repräsentiert. Sie ist für die Strukturierung von Szenarien zuständig und bildet damit die Struktur

4. Szenariobasierte Programmierung in Java

der Anwendung. Sie beinhaltet somit sämtliche Komponenten, die das Verhalten des Systems definieren. Das Verhalten wird allerdings nicht nur von Szenarien bestimmt. In SBP gibt es auch *Transformationsregeln*, die für die Änderungen im Objektsystem sorgen, wenn bestimmte Nachrichten auftreten (siehe Absatz 4.1.2). *Specification* ist eine abstrakte Klasse und für ihre Verwendung müssen die Methoden `registerScenarios` und `registerTransformationRules` implementiert werden.

4.1.1. Szenarien einbinden

Szenarien können in der Methode `registerScenarios` (siehe Codebeispiel 11) eingebunden werden. Hier gibt es zwei Methoden: `registerSpecificationScenario` und `registerAssumptionScenario`. Mit diesen Methoden lassen sich Szenarien entweder als *Specification* oder *Assumption* deklarieren.

```
1 @Override
2 protected void registerScenarios() {
3     // Register Scenarios here
4 }
```

Code 11: Die Methode `registerScenarios` in der Klasse *Specification*.

Die Methode `registerSpecificationScenario` (siehe Codebeispiel 12) erwartet eine Klasse, die *Scenario* erweitert, als Parameter. Ein Aufruf dieser Methode sorgt für die Registrierung eines Szenarios als Anforderungsszenario für das System.

```
1 protected boolean registerSpecificationScenario(Class<? extends Scenario>
    scenarioClass)
```

Code 12: Die Methode `registerSpecificationScenario` in der Klasse *Specification*.

Die Methode `registerAssumptionScenario` (siehe Codebeispiel 13) erwartet ebenfalls eine Klasse, die *Scenario* erweitert, als Parameter. Ein Aufruf dieser Methode sorgt für die Registrierung eines Szenarios als Annahmeszenario für die Umwelt.

```
1 protected boolean registerAssumptionScenario(Class<? extends Scenario> scenarioClass)
```

Code 13: Die Methode `registerAssumptionScenario` in der Klasse *Specification*.

Wird die Spezifikation ausgeführt, werden die registrierten Szenarien (sowohl *Specification*, als auch *Assumption*) in die szenariobasierte Anwendung aufgenommen und beim Auftreten einer initialisierenden Nachricht aktiviert.

4.1.2. Transformationsregeln für das Objektsystem

Zusätzlich zu den Szenarien können Transformationsregeln in der Spezifikation registriert werden. Diese Regeln definieren Veränderungen des Objektsystems, die als Reaktion auf aufgetretene Nachrichten angewendet werden. Sie bilden einen Ersatz für die Transformation durch Henshin in SCENARIOTOOLS. Die Regeln können innerhalb der Methode `registerTransformationRules` (siehe Codebeispiel 14) registriert werden.

```

1  @Override
2  protected void registerTransformationRules() {
3      // Register Model Transformation Rules here
4  }

```

Code 14: Die Methode `registerTransformationRules` in der Klasse `Specification`.

Die Methode `registerRule` (siehe Codebeispiel 15) registriert Klassen des Typs `TransformationRule` aus dem Paket `sbp.runtime.objectsystem`. Diese Regeln dienen der Anpassung des Objektsystems als Reaktion auf bestimmte Nachrichten. Das Codebeispiel 16 zeigt ein Beispiel für eine Transformationsregel aus dem Car-To-X Beispiel. Die Regel `RuleForRegister` übernimmt die tatsächliche Registrierung eines Fahrzeugs, wenn dieses die Nachricht `register` an eine `ObstacleControl` schickt. Die Methode `getTriggerMessage` ist eine abstrakte Methode und muss implementiert werden. Der Rückgabewert der Methode ist eine `Message`, die die Regel auslösen soll. In dem Beispiel ist das die Nachricht `register`, die von einem Objekt des Typs `Car` an ein Objekt des Typs `ObstacleControl` geschickt wird. Die abstrakte Methode `execute` beinhaltet die Reaktion der Regel. Im Beispiel wird das sendende Fahrzeug der Liste mit wartenden Fahrzeugen der Baustelle angehängt.

```

1  protected void registerRule(TransformationRule rule)

```

Code 15: Die Methode `registerRule` in der Klasse `Specification`.

```

1  package cartox.transformationrules;
2
3  import cartox.Car;
4  import cartox.ObstacleControl;
5  import sbp.runtime.objectsystem.TransformationRule;
6  import sbp.specification.events.Message;
7  import sbp.specification.role.Role;
8
9  public class RuleForRegister extends TransformationRule {
10
11     protected Role<Car> car = new Role<Car>(Car.class, "car");
12     protected Role<ObstacleControl> obstacleControl = new Role<ObstacleControl>(
13         ObstacleControl.class, "obstacleControl");
14
15     @Override
16     public Message getTriggerMessage() {
17         return new Message(car, obstacleControl, "register");
18     }
19 }

```

4. Szenariobasierte Programmierung in Java

```
17 }
18
19 @Override
20 public void execute(Message message) {
21     Car car = (Car) message.getSender().getBinding();
22     ObstacleControl obstacleControl = (ObstacleControl) message.getReceiver().
        getBinding();
23     obstacleControl.getRegisteredCarsWaiting().add(car);
24 }
25
26 }
```

Code 16: Die Regel RuleForRegister aus dem Car-To-X Beispiel. Diese Regel fügt den Sender der Nachricht `register` der Liste `registeredCarsWaiting` des Empfängers zu. Der Sender ist vom Typ `Car` und der Empfänger ist vom Typ `ObstacleControl`.

4.2. Nachrichten in Java

Die szenariobasierte Entwicklung spricht von einem Nachrichtenaustausch zwischen Komponenten des Systems und der Umwelt. Auch in SBP ist dieses Konzept vertreten und bildet die Funktionsweise einer szenariobasierten Anwendung. Eine solche SBP Nachricht wird in Java mit der Klasse `Message` dargestellt. Eine SBP Nachricht hat die selben Modalitäten wie eine SML-Nachricht, nämlich kann sie `STRICT` oder `non-STRICT` und `REQUESTED` oder `non-REQUESTED` sein. Während der Laufzeit werden diese booleschen Werte berücksichtigt, um das Verhalten des Szenarios, das die Nachricht aufruft, zu bestimmen. Der Codeblock 17 zeigt verschiedene Konstruktoren für die `Message`. Diese Menge an verschiedenen Konstruktoren dient der Lesbarkeit des Codes. Durch das Weglassen bestimmter Werte, wie `strict` oder `requested`, muss der Anwender weniger Code schreiben und der Code wird nicht unnötig lang. Werden die Werte für `STRICT` und `REQUESTED` weggelassen, so haben sie standardmäßig den Wert `false`. Werden Parameter weggelassen, so hat die Nachricht keine Parameter.

```
1 // Eine Message mit Sender, Empfänger und Namen.
2 // Die Message ist nicht strict und nicht requested.
3 Message(Role sender, Role receiver, String message)
4
5 // Diese Message hat zu den oberen Werten zusätzlich Parameter beliebigen Typs als
6 // Array.
7 Message(Role sender, Role receiver, String message, Object... parameters)
8 // Dieser Konstruktor deklariert zusätzlich, ob die Message strict und/oder requested
9 // ist.
10 Message(boolean strict, boolean requested, Role sender, Role receiver, String message)
11
12 // Das selbe wie oben, mit zusätzlichen Parametern als Liste.
13 Message(boolean strict, boolean requested, Role sender, Role receiver, String message
14         , List<Object> parameters)
```

```

13
14 // Das selbe wie zuvor, aber mit Parametern als Array.
15 Message(boolean strict, boolean requested, Role sender, Role receiver, String message
    , Object... parameters)
16
17 // Abschließend ein Konstruktor, der Parameter und den Wert vom Attribut strict
    setzt.
18 Message(boolean strict, Role sender, Role receiver, String message, Object...
    parameters)

```

Code 17: Konstruktoren der Klasse Message.

4.3. Java Szenario

Das Pendant zum SML Szenario bietet die abstrakte Klasse `Scenario`. Diese Klasse verlangt die Implementierung von vier Methoden:

- **registerAlphabet**: Die Methode `registerAlphabet` ist für die Registrierung des Alphabets des Szenarios zuständig. Hier werden *blockierte*, *verbotene* und *unterbrechende* Nachrichten registriert.
 - **Blockierte Nachricht**: Eine Nachricht, die regulär im Szenario vorkommt. Somit ist sie während der Ausführung verboten, außer sie ist gefordert oder wird erwartet. Sollte eine Nachricht in SML als *ignored* deklariert sein, so kann sie hier einfach weggelassen werden. Dadurch wird sie in der Auswahl der blockierten Nachrichten nicht weiter berücksichtigt - also ignoriert. Der Codeblock 18 zeigt den Befehl zum Blockieren einer Nachricht.

```

1 // Fügt eine blockierte Message mit Sender, Empfänger und Nachricht
    hinzu.
2 protected Message setBlocked(Role<?> sender, Role<?> receiver, String
    message)
3
4 // Fügt zusätzlich Parameter hinzu.
5 protected Message setBlocked(Role<?> sender, Role<?> receiver, String
    message, Object... parameters);
6
7 // Blockiert eine vordefinierte Message.
8 protected Message setBlocked(Message message)

```

Code 18: Methoden zum Blockieren einer Nachricht.

- **Verbotene Nachricht**: Eine verbotene Nachricht wird während der Ausführung gesondert behandelt. Tritt eine Nachricht auf, die in einem aktiven Szenario verboten ist, so wird das Szenario mit einer Safety-Violation abgebrochen, egal ob der aktuelle Zustand `strict` ist oder nicht. Dies entspricht der Deklaration einer Nachricht als *forbidden* in SML. Der Codeblock 19 zeigt den Befehl zum Blockieren einer Nachricht.

4. Szenariobasierte Programmierung in Java

```
1 // Fügt eine verbotene Message mit Sender, Empfänger und Nachricht hinzu
2 protected Message setForbidden(Role<?> sender, Role<?> receiver, String
   message)
3
4 // Fügt zusätzlich Parameter hinzu.
5 protected Message setForbidden(Role<?> sender, Role<?> receiver, String
   message, Object... parameters)
6
7 // Verbietet eine vordefinierte Message.
8 protected Message setForbidden(Message message)
```

Code 19: Methoden zum Verbieten einer Nachricht.

- **Unterbrechende Nachricht:** Dies ist eine Nachricht, die in einem Szenario eine Interrupt-Violation hervorruft, egal ob der aktuelle Zustand strict ist oder nicht. Sie wird ebenso gesondert behandelt, wie eine verbotene Nachricht, unterbricht das Szenario allerdings, anstatt eine Safety-Violation hervorzurufen. Dies entspricht der Deklaration einer Nachricht als *interrupting* in SML. Der Codeblock 20 zeigt den Befehl zum Blockieren einer Nachricht.

```
1 // Fügt eine unterbrechende Message mit Sender, Empfänger und Nachricht
   hinzu.
2 protected Message setInterrupting(Role<?> sender, Role<?> receiver,
   String message)
3
4 // Fügt zusätzlich Parameter hinzu.
5 protected Message setInterrupting(Role<?> sender, Role<?> receiver,
   String message, Object... parameters)
6
7 // Fügt eine vordefinierte Message zur Liste der unterbrechenden
   Nachrichten hinzu.
8 protected Message setInterrupting(Message message)
```

Code 20: Methoden zum Deklarieren einer unterbrechenden Nachricht.

- **initialisation:** In dieser Methode werden initialisierende Nachrichten für das Szenario angegeben. Initialisierende Nachrichten sind solche, die ein Szenario aktivieren, sobald sie auftreten. Damit ein Szenario aktiviert werden kann, müssen diese Nachrichten mit der Methode `addInitializingMessage` registriert werden. Der Codeblock 21 zeigt die Signatur dieser Methode.

```
1 protected void addInitializingMessage(Message message)
```

Code 21: Methoden für die Deklaration von initialisierenden Nachrichten.

- **registerRoleBindings:** Diese Methode ist für das Binden von Rollen, die nicht bereits durch die initialisierende Nachricht gebunden sind. Bei

der Initialisierung des Szenarios werden die Rollen, die zur Initialisierung geführt haben, automatisch gebunden. Alle anderen Rollen müssen explizit mittels der Methode `bindRoleToObject` gebunden werden. Der Codeblock 22 zeigt die Signatur dieser Methode.

```
1 protected <T> void bindRoleToObject(Role<T> role, T object)
```

Code 22: Die Methode zum Binden einer Rolle an ein Objekt.

- **body**: In der Methode `body` wird das eigentliche Szenario deklariert. Sobald ein Szenario aktiviert wird, wird diese Methode gestartet. Ist diese Methode beendet, deaktiviert sich auch das Szenario. Sollte innerhalb des Szenarios eine Verletzung auftreten, wird diese weitergeleitet und intern in der Bibliothek aufgefangen. In dieser Methode stehen folgende Möglichkeiten zur Steuerung des Systemverhaltens bereit:
 - Eine Nachricht anfordern: Die Methode `request` bewirkt das Fordern einer Nachricht. Die in Codeblock 23 gezeigten Methoden können verwendet werden, um den Aufruf einer Nachricht als *requested* darzustellen, wie in SML.

```
1 // Fordert eine vordefinierte Message an. Der Zustand ist nicht strict.
2 protected Message request(Message message) throws Violation
3
4 // Fordert ein simples Message mit Sender, Empfänger und Nachricht an.
  // Der Zustand ist nicht strict.
5 protected Message request(Role<?> sender, Role<?> receiver, String
  message) throws Violation
6
7 // Genauso wie zuvor, allerdings hat die Nachricht Parameter als Array.
8 protected Message request(Role<?> sender, Role<?> receiver, String
  message, Object... parameters) throws Violation
9
10 // Genauso wie zuvor, allerdings hat die Nachricht Parameter als Liste.
11 protected Message request(Role<?> sender, Role<?> receiver, String
  message, List<Object> parameters) throws Violation
12
13 // Fordert eine vordefinierte Nachricht an. Setzt den Wert für das
  // Attribut strict.
14 protected Message request(boolean strict, Message message) throws
  Violation
15
16 // Fordert eine simple Nachricht an. Setzt den Wert für das Attribut
  // strict.
17 protected Message request(boolean strict, Role<?> sender, Role<?>
  receiver, String message) throws Violation
18
19 // Fordert eine Nachricht mit Parametern als Array an. Setzt den Wert für
  // das Attribut strict.
20 protected Message request(boolean strict, Role<?> sender, Role<?>
  receiver, String message, Object... parameters) throws Violation
21
22 // Fordert eine Nachricht mit Parametern als Liste an. Setzt den Wert für
  // das Attribut strict.
23 protected Message request(boolean strict, Role<?> sender, Role<?>
  receiver, String message, List<Object> parameters) throws Violation
```

Code 23: Methoden zum Anfordern von Nachrichten.

4. Szenariobasierte Programmierung in Java

- **Auf eine Nachricht warten:** Durch die Methode `waitFor` kann das Warten auf eine Nachricht erwirkt werden. Diese Nachricht wird dadurch nicht direkt ausgeführt, sondern nur erwartet. Das Szenario bleibt an dieser Stelle stehen, bis die angegebene Nachricht aufgetreten ist. Der Codeblock 24 zeigt die Varianten dieser Methode.

```
1 // Erwartet eine vordefinierte Message. Der Zustand ist nicht strict.
2 protected Message waitFor(Message message) throws Violation
3
4 // Erwartet eine simple Message mit Sender, Empfänger und Nachricht. Der
  Zustand ist nicht strict.
5 protected Message waitFor(Role<?> sender, Role<?> receiver, String
  message) throws Violation
6
7 // Genauso wie zuvor, allerdings hat die Nachricht Parameter als Array.
8 protected Message waitFor(Role<?> sender, Role<?> receiver, String
  message, Object... parameters) throws Violation
9
10 // Genauso wie zuvor, allerdings hat die Nachricht Parameter als Liste.
11 protected Message waitFor(Role<?> sender, Role<?> receiver, String
  message, List<Object> parameters) throws Violation
12
13 // Erwartet eine vordefinierte Nachricht. Setzt den Wert für das
  Attribut strict.
14 protected Message waitFor(boolean strict, Message message) throws
  Violation
15
16 // Erwartet eine simple Nachricht. Setzt den Wert für das Attribut
  strict.
17 protected Message waitFor(boolean strict, Role<?> sender, Role<?>
  receiver, String message) throws Violation
18
19 // Erwartet eine Nachricht mit Parametern als Array. Setzt den Wert für
  das Attribut strict.
20 protected Message waitFor(boolean strict, Role<?> sender, Role<?>
  receiver, String message, Object... parameters) throws Violation
21
22 // Erwartet eine Nachricht mit Parametern als Liste. Setzt den Wert für
  das Attribut strict.
23 protected Message waitFor(boolean strict, Role<?> sender, Role<?>
  receiver, String message, List<Object> parameters) throws Violation
```

Code 24: Methoden zum Erwarten von Nachrichten.

- **Fordern und Erwarten von mehreren Nachrichten:** Sollte der Fall eintreten, dass mehrere Nachrichten gefordert oder erwartet werden sollen, kann die Methode `doStep` verwendet werden. Diese Methode dient für den Anwendungsfall, dass eine Alternative modelliert werden soll. Wie genau dieser Fall aussehen kann, zeigt Absatz 5.1.5. Der Codeblock 25 zeigt die Signatur dieser Methode.

```
1 protected void doStep(List<Message> requestedMessages, List<Message>
  waitedForMessages)
```

Code 25: Die Methode `doStep` zum Anfordern oder Erwarten mehrerer Nachrichten.

Zusätzlich bieten Szenarien weitere Methoden, die überschrieben werden kön-

nen, um weitere Funktionalität zu bieten:

- Szenarien bieten zwei Methoden, Verletzungen abzufangen. Dies kann für das Testen von Szenarien relevant sein. Ebenfalls können sie verwendet werden, um eine Reaktion auf Verletzungen zu implementieren. Codeblock 26 zeigt die Methoden, die überschrieben werden können, um eine Verletzung abzufangen.

```

1  @Override
2  protected void catchInterruptViolation(InterruptViolation e) {}
3
4  @Override
5  protected void catchSafetyViolation(SafetyViolation e) {}

```

Code 26: Methoden zum Auffangen von Verletzungen.

- Die Methode `Reset` ermöglicht eine Reaktion auf das Beenden eines Szenarios. Sie wird nach der Vollendung der Methode `body` ausgeführt, wenn das Szenario deaktiviert wird. Sie kann überschrieben werden, um eine Reaktion darauf zu implementieren.

```

1  @Override
2  protected void reset() {}

```

Code 27: Methode, die beim Beenden von Szenarien ausgeführt wird.

4.3.1. Ausführung von Szenarien beobachten

Um die Ausführung eines Szenarios zu beobachten, lassen sich sogenannte Observer definieren und einem Szenario hinzufügen. Ein im Szenario registrierter Observer wird immer dann benachrichtigt, wenn ein Szenario in seiner Ausführung voranschreitet. Dies kann für Debug Zwecke genutzt werden, um die korrekte Funktionalität eines Szenarios sicherzustellen. Der primäre Zweck dieser Observer ist allerdings die Weiterleitung von Nachrichten aus der szenariobasierten Anwendung heraus, bzw. die Verarbeitung einer Nachricht. Beispielsweise kann als Reaktion auf bestimmte Nachrichten eine GUI für den Benutzer aktualisiert werden. Im Falle des Car-To-X Beispiels könnte es eine Streckenkontrolle geben, die die Position der Fahrzeuge auf einer Karte visualisiert. Immer wenn ein Szenario einen neuen Streckenabschnitt betritt, wird dies durch eine Nachricht kommuniziert. Als Reaktion auf diese Nachricht wird dann die Karte aktualisiert. Ein zweiter Anwendungsfall für solch einen Observer ist die Steuerung von Aktoren eines Systems. Beispielsweise kann so die Motorik eines Roboters aktiviert werden. Der Roboter kann also durch die Szenarien gesteuert

4. Szenariobasierte Programmierung in Java

werden. Tritt zum Beispiel eine Nachricht `moveForward` auf, so wird die Motorik des Roboters durch den Observer dazu veranlasst eine bestimmte Strecke vorwärts zu fahren.

Für die Implementierung eines Observers stellt SBP die abstrakte Klasse `ScenarioObserver` bereit. Diese Klasse wird an ein Szenario gebunden und verlangt die Implementierung der Methode `trigger`. Diese Methode wird dann aufgerufen, wenn das beobachtete Szenario voranschreitet. Der Codeblock 28 zeigt die Signatur der Methode. Die Parameter werden an das beobachtete Szenario und die aufgetretene Nachricht gebunden.

```
1 public abstract void trigger(Scenario scenario, Message message)
```

Code 28: Die `trigger` Methode der Klasse `ScenarioObserver`.

4.4. Java Objektsystem

Für die Ausführung der Java Spezifikation muss auch hier ein Objektsystem den Ausgangszustand des Systems bereitstellen. Das Design des Objektsystems kann je nach Anwendungsbereich frei gewählt werden und muss keinen festen Mustern folgen. Demnach gibt SBP hier auch keine Klasse vor. Der Benutzer kann hier frei seine eigene Klasse oder auch mehrere Klassen bereitstellen und in der Runconfiguration registrieren. Es ist allerdings auch möglich, ein Objektsystem von `ScenarioTools` zu übernehmen. Da es sich dabei um eine dynamische Instanz eines EMF Ecore Models handelt, wird dafür nur ein *Wrapper* benötigt. Das Codebeispiel 29 zeigt die Klasse für das Objektsystem des Car-To-X Beispiels. Diese Klasse lädt bei ihrer Initialisierung das EMF Objektsystem des SML Beispiels aus einer vorgegebenen Datei. Dadurch kann ohne viel Mehraufwand ein vorhandenes Objektsystem aus SML in SBP verwendet werden.

```
1 public class CarToXObjectSystem {
2
3     private static CarToXObjectSystem instance;
4
5     public static CarToXObjectSystem getInstance() {
6         if (instance == null)
7             instance = new CarToXObjectSystem();
8         return instance;
9     }
10
11     public CarToX CarToX = (CarToX) loadCarToX("cartox.xmi");
12
13     public CarToX loadCarToX(String uri) {
14         // Load models into JVM
15         CartoxPackage.eINSTANCE.eClass();
16
17         // Register xmi file extension
18         Resource.Factory.Registry reg = Resource.Factory.Registry.INSTANCE;
19         Map<String, Object> m = reg.getExtensionToFactoryMap();
20         m.put("xmi", new XMIResourceFactoryImpl());
```

```

21
22 // Load xmi file and return object system
23 ResourceSet resSet = new ResourceSetImpl();
24 Resource resource = resSet.getResource(URI.createURI(uri), true);
25 CarToX carToX = (CarToX) resource.getContents().get(0);
26 return carToX;
27 }
28
29 }

```

Code 29: Die Klasse für das Java Objektsystem des Car-To-X Beispiels.

4.5. Java Runconfiguration

Den Platz der SML Runtime Configuration nimmt in SBP die abstrakte Klasse `SpecificationRunconfig` ein. Diese Klasse dient dem Start der szenariobasierten Anwendung. Sie ist generisch auf eine Klasse getypt, die die Klasse `Specification` erweitert, und erwartet für die Instanziierung eine solche Klasse als Parameter. In dieser Klasse werden das Objektsystem und die Spezifikation registriert. Außerdem werden Netzwerkschnittstellen bereitgestellt, um eine verteilte Kommunikation von Komponenten zu erreichen. Dafür wird die Implementierung von drei Methoden erwartet:

- **registerParticipatingObjects:** In dieser Methode werden Objekte, die an der Ausführung der Spezifikation teilnehmen, registriert. Zudem wird hier deklariert, ob ein Objekt kontrollierbar oder unkontrollierbar ist. Das beeinflusst, wie sich die Nachrichtenauswahl während der Ausführung verhält. Dafür steht die Methode `registerObject` (siehe Codeblock 30) zur Verfügung. Sie erwartet ein Objekt beliebigen Typs aus dem Objektsystem und eine Zahl, die angibt, ob das Objekt `CONTROLLABLE` oder `UNCONTROLLABLE` ist. Das Codebeispiel 31 zeigt, wie die Objekte des Car-To-X Beispiels in einer `SpecificationRunconfig` registriert werden. Die Objekte sollten aus der Klasse `CarToXObjectSystem` aus dem vorherigen Abschnitt kommen.

```

1 public static final int CONTROLLABLE = 0;
2 public static final int UNCONTROLLABLE = 1;
3
4 protected void registerObject(Object object, int controllableMode)

```

Code 30: Die Methode zum Registrieren von Objekten in der SBP Runtime.

4. Szenariobasierte Programmierung in Java

```
1 protected void registerParticipatingObjects() {
2     CarToXObjectSystem objectSystem = CarToXObjectSystem.getInstance();
3     // JamesCar ist kontrollierbar
4     registerObject(objectSystem.CarToX.getCars().get(0), CONTROLLABLE);
5     // JamesCarDashboard ist unkontrollierbar
6     registerObject(objectSystem.CarToX.getCars().get(0).getDashboard(),
7         UNCONTROLLABLE);
8     // James ist unkontrollierbar
9     registerObject(objectSystem.CarToX.getCars().get(0).getDriver(),
10        UNCONTROLLABLE);
11    // env ist unkontrollierbar
12    registerObject(objectSystem.CarToX.getEnvironment(), UNCONTROLLABLE);
13    // obstacle ist unkontrollierbar
14    registerObject(objectSystem.CarToX.getStreetSections().get(0).getObstacles()
15        .get(0), UNCONTROLLABLE);
16    // obstacleControl ist kontrollierbar
17    registerObject(objectSystem.CarToX.getObstacleControls().get(0),
18        CONTROLLABLE);
19 }
```

Code 31: Registrierung der Objekte für das Car-To-X Beispiel.

- **registerNetworkAddressesForObjects**: Diese Methode dient der Registrierung von Netzwerkadressen der Objekte des Objektsystems. Dies ist für die verteilte Ausführung über das Netzwerk nötig. Über diese Adressen kann die Runtime einer Komponente mit denen der anderen Komponenten kommunizieren. Dafür gibt es die Methode `registerAddress` (siehe Codeblock 32), die zu einem Objekt den Hostnamen und den Netzwerkport verlangt. Dies wird zur Laufzeit ausgewertet und für die Kommunikation benutzt.

```
1 protected void registerAddress(Object object, String hostName, int port)
```

Code 32: Die Methode zum Registrieren von Netzwerkadressen der Objekte in der SBP Runtime.

- **registerObservers**: Diese Methode dient als Schnittstelle für andere Teile der Anwendung. Hier werden Observer registriert, die auf Änderungen in Szenarien reagieren. Registrierte Observer werden in ein spezielles *System-Szenario* der SBP Runtime eingehängt und beobachtet auftretende Nachrichten. Durch diesen Mechanismus kann beispielsweise eine GUI aktualisiert oder die Aktoren eines Roboters gesteuert werden. Die Methode `addScenarioObserver` (siehe Codeblock 33) registriert einen Observer für die Runtime.

```
1 protected void addScenarioObserver(ScenarioObserver scenarioObserver)
```

Code 33: Die Methode zum Registrieren von Observern für die Runtime.

4.5.1. Start einer SpecificationRunconfig

Um eine SpecificationRunconfig zu starten, bietet diese Klasse die statische Methode `run`, die eine Klasse als Parameter erwartet, die die `SpecificationRunconfig` erweitert. Wird diese Methode mit entsprechendem Parameter aufgerufen, wird die SBP Runtime initialisiert. Durch die Initialisierung werden eingestellte Objekte, Netzwerkadressen und Observer registriert und die Liste der Szenarien der eingetragenen Specification eingelesen. Als nächstes wird die B-Application, auf der die SpecificationRunconfig basiert, gestartet und damit ist die szenariobasierte Anwendung aktiv.

```
1 public static void run(Class<? extends SpecificationRunconfig<? extends Specification
  >> specificationRunconfigClass)
```

Code 34: Die Methode zum starten einer SpecificationRunconfig.

4.6. Runtime-Adapter

Der Runtime-Adapter bestimmt, wie die Ausführung der szenariobasierten Anwendung zwischen den Komponenten des Systems synchronisiert wird. Ein Adapter reagiert auf die Ausführung von Nachrichten und leitet sie an andere Komponenten weiter. Zudem ist er verantwortlich für das Einspeisen von Umweltnachrichten in die SBP Runtime. SBP liefert bereits zwei Adapter und ein Interface für die Herstellung weiterer Adapter, die im Folgenden erläutert werden.

4.6.1. Runtime-Adapter Interface und Adapterbau

Das Interface `RuntimeAdapter` definiert die nötigen Funktionen eines Runtime-Adapters für SBP. Für die Implementierung benutzerdefinierter Adapter, bietet SBP allerdings außerdem die abstrakte Klasse `AbstractRuntimeAdapter`, die bereits die organisatorischen Dinge für den Adapter regelt. Die Runtime-Adapter für SBP sind in der Form von Client-Server Strukturen vorgesehen, können allerdings auch beliebig angepasst werden. Folgende Methoden sind noch für den Adapter zu implementieren:

- **run**: Die Methode `run` wird ausgeführt, wenn die szenariobasierte Anwendung ausgeführt wird. Sie bestimmt, wie der Adapter zu initialisieren ist. Beispielsweise können hier Client- und Serverprozesse gestartet werden. Alternativ kann auch ein anderes Kommunikationsprotokoll initialisiert und verwendet werden. Sollte es eine zentralisierte Intelligenz im System

4. Szenariobasierte Programmierung in Java

geben, beispielsweise einen Server, der die Kommunikation leitet, kann auch die Verbindung zu diesem Server hier aufgebaut werden.

- **startServer**: Diese Methode ist für die Initialisierung der Serverschnittstelle der SBP Runtime gedacht. Ein Server hat die Funktion, Nachrichten von außen - also von anderen Komponenten - zu erhalten. Beispielsweise kann ein Server über das Netzwerk gehostet werden.
- **connectClients**: In dieser Methode geht es um die Verbindung zwischen verschiedenen SBP Runtimes. Das ist für eine verteilte Ausführung nötig, da jede verteilte Komponente eine eigene SBP Runtime besitzt, die synchronisiert werden muss. Die Methode **connectClients** ist dafür zuständig, die Kommunikationswege zu den anderen Komponenten aufzubauen.
- **publish**(Message message): Wird eine Message von der Runtime ausgeführt, wird die Methode **publish** aufgerufen, die die Message an die anderen Komponenten weitergibt.
- **receive**(Message message): Ruft eine andere Komponente des Systems eine Nachricht auf und publiziert dies, wird die Methode **receive** aufgerufen. Diese Methode speist die Nachricht dann in die SBP Runtime ein.

4.6.2. Runtime-Adapter für lokale Simulation

Die SBP Bibliothek stellt einen Adapter für die lokale Simulation der Spezifikation bereit. Wird für die Ausführung die Klasse **LocalAdapter** als Runtime-Adapter ausgewählt, so wird die Spezifikation lokal ausgeführt, ohne dass eine Kommunikation mit anderen Komponenten stattfindet. Dies ist dann sinnvoll, wenn das System getestet werden soll, oder bestimmte Komponenten vorerst vom Benutzer gesteuert werden sollen. Die einzige Methode, die in diesem Adapter eine Funktion hat, ist die Methode **receive**, die aufgerufen werden kann, um eine Nachricht in das System einzuspeisen. Beispielsweise kann eine GUI verwendet werden, um Nachrichten von außen zu simulieren.

4.6.3. Runtime-Adapter für verteilte Ausführung

Soll eine Spezifikation verteilt ausgeführt werden, wird der **DistributedRuntimeAdapter** benötigt. Diese Klasse beinhaltet eine Client-Server Struktur, die für eine Kommunikation zwischen den Komponenten des Systems sorgt. Bei der Initialisierung des Adapters wird ein Server über Java Sockets auf einem bestimmten Port gestartet. Dieser Port ist über den Konstruktor des Adapters frei wählbar.

Danach wird für jede registrierte Netzwerkadresse in der `SpecificationRunconfig` ein Client-Thread gestartet, der sich auf den Server bei der gegebenen Netzwerkadresse anmeldet. Sobald alle Netzwerkadressen verbunden sind, ist der Adapter fertig initialisiert und die szenariobasierte Anwendung kann gestartet werden. Sobald eine Nachricht in der SBP Runtime ausgeführt wird, wird sie an alle verbundenen Server weitergeleitet. Wird eine Nachricht in einer anderen Komponente ausgeführt, wird ein Client-Thread benachrichtigt und leitet diese Nachricht weiter in die SBP Runtime. Alle Komponenten des Systems sind dadurch vollständig miteinander verbunden.

4.6.4. Registrierung eines Runtime-Adapters

Damit ein Runtime-Adapter verwendet werden kann, muss er in der Klasse `SpecifacaitonRunconfig` registriert werden. Dafür stehen dir folgenden Methoden bereit:

- **enableLocalSimulationMode**: Diese Methode versetzt die Runtime in den Modus der lokalen Simulation. Der Adapter für die lokale Simulation wird installiert. Das bedeutet, es gibt nur die eine Runtime und es passiert keine Kommunikation nach außen.
- **enableDistributedExecutionMode**: Diese Methode versetzt die Runtime in den Modus der verteilten Ausführung. Der Adapter für die verteilte Ausführung wird installiert. Das bedeutet, es kann mehrere Runtimes geben und es wird die Client-Server-Architektur verwendet. Dafür muss dieser Methode ein Port für den Server mitgegeben werden.
- **setAdapter**: Diese Methode ist für weitere Adapter gedacht. Sie installiert einen Adapter, der für ein spezielles Kommunikationsprotokoll entworfen wurde. Hier können eigene Adapter verwendet werden.

4. Szenariobasierte Programmierung in Java

5. SML-to-SBP Mapping

Ein Teil dieser Arbeit war es, ein Mapping von SML zu SBP zu entwickeln. Auf der Basis dieses Mappings kann mit Hilfe eines Code-Generators aus einer SML Spezifikation eine SBP Anwendung generiert werden. Dadurch ist es möglich, eine SML Spezifikation zu SBP Code umzuwandeln, der dann auf verschiedene Geräte oder Plattformen aufgespielt werden kann. So ist eine verteilte Ausführung einer SML Spezifikation mit Hilfe von SBP möglich. In diesem Kapitel wird dieses Mapping vorgestellt. Abbildung 5.1 zeigt eine Übersicht des Mappings.

5.1. Mapping für ein Szenario

Ein Szenario in SML wird in eine Klasse transformiert, die von `Scenario` erbt. Der Name der Klasse ist der Name des Szenarios, aus Gründen der Wiedererkennung. Ob ein Szenario vom Typ *Specification* oder *Assumption* ist, wird in der `Scenario` Klasse nicht klar, da dies erst in der SBP Specification eingetragen wird. Zudem müssen vier Teile des SML-Szenarios transformiert werden:

1. Die initialisierenden Nachrichten des SML Szenarios müssen in der neuen Klasse registriert werden. Dafür können Funktionen der Utility Klassen aus `ScenarioTools` verwendet werden, die eine Liste der initialisierenden Nachrichten ermittelt. Diese Nachrichten sind diejenigen, die an erster Stelle im Szenario stehen, also als Erstes auftreten müssen, um das Szenario zu aktivieren.
2. Der Body: Nach den initialisierenden Nachrichten stehen weitere Nachrichten und Strukturen im SML-Szenario. Diese müssen in den Body der neuen `Scenario` Klasse übernommen werden. Dafür kann Schritt für Schritt durch die Fragmente des Szenarios iteriert werden, um die Fragmente umzuwandeln. Diese Iteration über den Body darf allerdings erst nach den initialisierenden Fragmenten beginnen.
3. Das Alphabet: Alle Nachrichten, die in einem SML Szenario vorkommen, müssen in der neuen `Scenario` Klasse generell blockiert werden. Dies funktioniert einfach durch eine Iteration über die Liste der `ModalMessages`

5. SML-to-SBP Mapping

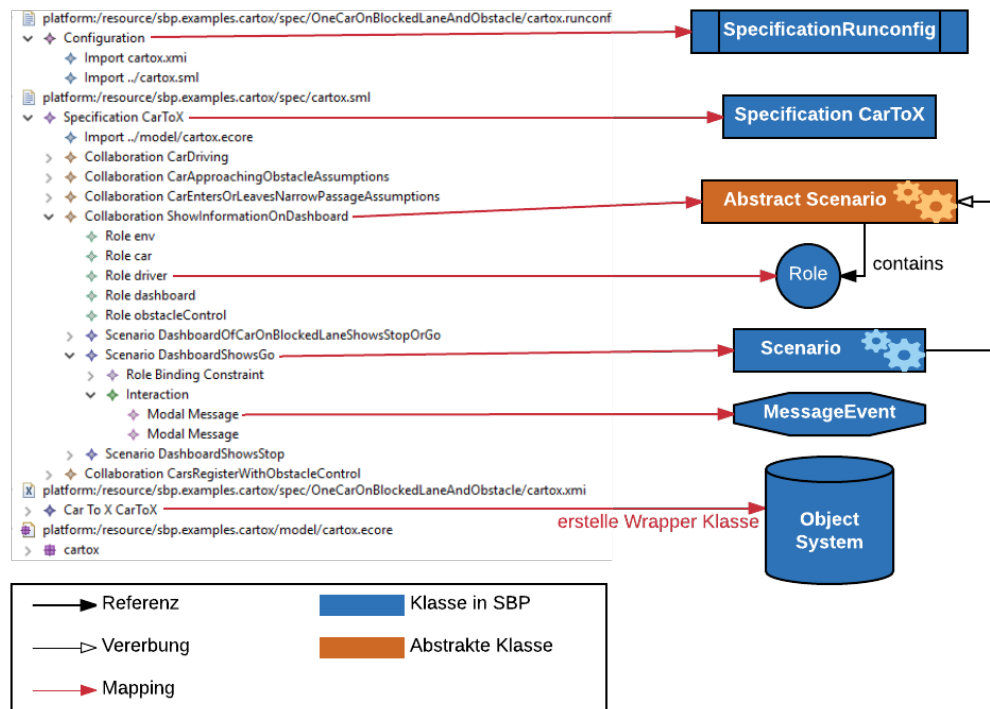


Abbildung 5.1.: Übersicht des Mappings von SML zu SBP. Auf der linken Seite ist eine SML Konfiguration mit Spezifikation und Objektsystem zu sehen. Auf der rechten Seite das dazugehörige Mapping in SBP.

des Szenarios. Allerdings müssen hier schon die Constraints des Szenarios betrachtet werden. Wenn eine Nachricht mit *ignore* deklariert ist, darf sie nicht blockiert werden. Des Weiteren werden die Constraints mit Nachrichten, die als *forbidden* oder *interrupt* deklariert sind, als solche in das Alphabet übernommen.

4. Role Bindings: Die Role Bindings des SML Szenarios müssen auch in dem SBP Scenario übernommen werden. Dafür werden die Bindingausdrücke aus SML eins-zu-eins in Bindingausdrücke in SBP übersetzt. Dabei ist zu beachten: In SML gibt es statische Rollen, die an ein Objekt fest gebunden werden und dynamische Rollen, die an viele Objekte gebunden werden können. In SBP gibt es nur die dynamischen Rollen. Statische Rollen müssen also auf dynamische Rollen abgebildet werden. Dies kann über

Bindingausdrücke auf spezifische Objekte des Objektsystems geschehen.

Ist dies erledigt, fehlen nur noch die Rollen selbst. Da es in SBP keine Klasse für Collaborations gibt, gibt es auch keinen globalen Platz für die Rollen. Eine einfache Möglichkeit wäre es, die Rollen der Collaboration in jedes darin enthaltene Szenario zu generieren. Die SBP Rollen tragen dann dieselben Namen, wie die Rollen in SML. Eine bessere Möglichkeit bietet ein abstraktes Szenario als Superklasse aller Szenarien aus der Collaboration. Mehr dazu in Abschnitt 5.2. Codebeispiel 35 zeigt ein SML Szenario aus dem Car-To-X Beispiel, welches in Codebeispiel 36 in SBP gemappt wurde. Die Rollen sind in dem abstrakten Szenario `ShowInformationOnDashboardCollaboration` hinterlegt.

```

1 specification scenario DashboardShowsGo
2 with dynamic bindings [
3   bind dashboard to car.dashboard
4 ] {
5   message obstacleControl -> car.enteringAllowed
6   message strict requested car -> dashboard.showGo
7 }

```

Code 35: Beispiel Szenario des Car-To-X Beispiels in SML.

```

1 public class DashboardShowsGoScenario extends ShowInformationOnDashboardCollaboration
2 {
3   @Override
4   protected void registerAlphabet() {
5     setBlocked(obstacleControl, car, "enteringAllowed");
6     setBlocked(car, dashboard, "showGo");
7   }
8
9   @Override
10  protected void initialisation() {
11    addInitializingMessage(new Message(obstacleControl, car, "enteringAllowed"));
12  }
13
14  @Override
15  protected void registerRoleBindings() {
16    bindRoleToObject(dashboard, (Dashboard) car.getBinding().getDashboard());
17  }
18
19  @Override
20  protected void body() throws Violation {
21    request(STRICT, car, dashboard, "showGo");
22  }
23 }

```

Code 36: Beispiel Szenario des Car-To-X Beispiels in SBP.

5.1.1. Mapping für eine ModalMessage

Für das Mapping einer ModalMessage ist es wichtig, ob die Nachricht als `requested` deklariert ist. Wenn sie das ist, wird sie im SBP Szenario mit der Methode `request` gefordert. Ansonsten wird sie mit der Methode `waitFor` erwartet. Das

5. SML-to-SBP Mapping

Codebeispiel 37 zeigt zwei Nachrichten in SML, die in Codebeispiel 38 transformiert werden. Ist die Nachricht als strict deklariert, wird in SBP der Wert STRICT in dem Ausdruck gesetzt, ansonsten wird er weggelassen. Das Feature, das in der SML Nachricht angegeben ist, wird für SBP einfach mit seinem Namen als String angegeben.

```
1 // Die Nachricht ist nicht STRICT und nicht REQUESTED
2 message obstacleControl -> car.enteringAllowed
3 // Die Nachricht ist STRICT und REQUESTED
4 message strict requested car -> dashboard.showGo
```

Code 37: Zwei Beispielnachrichten in SML.

```
1 // Die Nachricht ist nicht STRICT und nicht REQUESTED
2 waitFor(obstacleControl, car, "enteringAllowed");
3 // Die Nachricht ist STRICT und REQUESTED
4 request(STRICT, car, dashboard, "showGo");
```

Code 38: Das Mapping der Beispielnachrichten in SBP.

5.1.2. Mapping für eine Condition

Eine Condition ist in SML ein besonderes Fragment, bzw. ein Teil eines Fragmentes. In Java und damit auch in SBP wird sie einfach mit *if*, *then* und *else* abgebildet. Betrachtet man eine Interrupt-Condition in SML (siehe Codebeispiel 39), so wird sie in ein *if*-Statement in Java übersetzt, welches bei Erfüllung die Methode `throwViolation` des Szenarios auslöst. INTERRUPT ist dabei die Schwere der Verletzung und verursacht eine Interrupt-Violation. Alternative wird SAFETY für eine Safety-Violation verwendet. Conditions in Loops oder Alternativen werden ebenfalls mit If-Statements übersetzt (siehe Abschnitt 5.1.4 und Abschnitt 5.1.5).

```
1 // Unterbreche das Szenario, wenn es kein Hindernis gibt.
2 // obstacle ist eine Rolle vom Typ Obstacle.
3 interrupt if [ obstacle == null ]
```

Code 39: Interrupt-Condition in SML.

```
1 if ((obstacle.getBinding() == null)) {
2     throwViolation(INTERRUPT);
3 }
```

Code 40: Das Mapping der Interrupt-Condition in SBP.

5.1.3. Mapping für eine Variable

Variablen in SML und Java sind sich sehr ähnlich. Beide sind eindeutig getypt und können deklariert und zugewiesen werden. Darum ist das Mapping hier auch

sehr leicht, nämlich eins-zu-eins. Codebeispiel 41 zeigt eine Variablendeklaration mit Zuweisung in SML, welche in Java 42 übersetzt wird. Die EMF Typen, wie EInt, werden dabei in Javatypen (int) transformiert.

```
1 var EInt value = 5
2 value = value + 5
```

Code 41: Eine Variablendeklaration und -zuweisung in SML.

```
1 int value = 5;
2 value = value + 5;
```

Code 42: Das Mapping der Variablendeklaration und -zuweisung in SBP.

5.1.4. Mapping für einen Loop

Das Mapping für einen Loop ist relativ simpel. Ein Loop kann eine Condition haben und beinhaltet mehrere Fragmente. Das Pendant zum SML Loop in Java ist das *while*-Statement. Die Codebeispiele 43 und 44 zeigen einen Loop in SML und das passende Mapping in Java. Die Condition wird erneut eins-zu-eins übersetzt und in ein *while*-Statement übertragen. Solange die Bedingung erfüllt bleibt, werden die Nachrichten innerhalb des Loops abgearbeitet. Diese Nachrichten werden wie zuvor transformiert.

```
1 // env und car sind Rollen
2 // Dieser Loop wird niemals enden
3 while [ true ] {
4   message requested env -> car.movedToNextArea
5 }
```

Code 43: Ein Loop in SML.

```
1 // Begin Loop
2 while (true) {
3   request(env, car, "movedToNextArea");
4 }
5 // End Loop
```

Code 44: Das Mapping des Loops in SBP.

5.1.5. Mapping für eine Alternative

Das Mapping einer Alternative ist bereits etwas komplizierter. Ähnlich wie in der Transformation von Szenarien, muss hier pro alternativem Pfad zwischen der ersten Nachricht und dem Rest unterschieden werden. Die Wahl des alternativen Pfades passiert durch die aufgetretene Nachricht. Demnach entscheidet

5. SML-to-SBP Mapping

diese Nachricht, welcher Pfad zu nehmen ist. Die Codebeispiele 45 und 46 zeigen eine Alternative in SML und ein passendes Mapping in Java. Zunächst werden zwei Listen erzeugt. Die Liste `requestedMessages` beinhaltet alle ersten Nachrichten der Alternative, die als *requested* deklariert sind, und die Liste `waitedForMessages` enthält alle anderen ersten Nachrichten. Für jeden alternativen Pfad wird eine Bedingung für das If-Statement erzeugt, die die Condition des Pfades widerspiegelt. Bei Erfüllung der Bedingung, wird die erste Nachricht des Pfades in eine der Listen gespeichert. Anschließend werden mit Hilfe der `doStep` des Szenarios die beiden Listen an die Runtime übergeben. Danach folgt die weitere Auswertung der Pfade. Mit Hilfe der Methode `getLastMessage` des Szenarios kann ermittelt werden, welche Nachricht ausgelöst wurde. Durch weitere *if-then-else*-Statements für jeden Pfad kann so über den Vergleich der Nachrichten ermittelt werden, welcher Pfad ausgewählt werden muss. Innerhalb der Pfade werden die Fragmente des Szenarios normal weiter transformiert.

```
1 // oc, car und dashboard sind Rollen
2 // oc ist vom Typ ObstacleControl
3 alternative if [ oc.carsOnNarrowPassageLaneAllowedToPass.isEmpty() ] {
4   message strict requested oc -> car.enteringAllowed
5   message strict car -> dashboard.showGo
6 } or if [ !oc.carsOnNarrowPassageLaneAllowedToPass.isEmpty() ] {
7   message strict requested oc -> car.enteringDisallowed
8   message strict car -> dashboard.showStop
9 }
```

Code 45: Eine Alternative in SML.

```
1 // Begin Alternative
2 List<Message> requestedMessages = new ArrayList<Message>();
3 List<Message> waitedForMessages = new ArrayList<Message>();
4 if (oc.getBinding().getCarsOnNarrowPassageLaneAllowedToPass().isEmpty()) {
5   requestedMessages.add(new Message(STRICT, oc, car, "enteringAllowed"));
6 }
7 if (!oc.getBinding().getCarsOnNarrowPassageLaneAllowedToPass().isEmpty()) {
8   requestedMessages.add(new Message(STRICT, oc, car, "enteringDisallowed"));
9 }
10 // Insert into BP
11 doStep(requestedMessages, waitedForMessages);
12 // Determine which path has been chosen
13 if (getLastMessage().equals(new Message(oc, car, "enteringAllowed"))) {
14   waitFor(STRICT, car, dashboard, "showGo");
15 } else if (getLastMessage().equals(new Message(oc, car, "enteringDisallowed"))) {
16   waitFor(STRICT, car, dashboard, "showStop");
17 }
18 // End Alternative
```

Code 46: Das Mapping der Alternative in SBP.

5.1.6. Mapping für ein Parallel Fragment

Das Mapping für das Parallel Fragment erfordert eine besondere Mechanik. Wird ein Parallel Fragment aktiviert, wird nicht nur ein Pfad durchlaufen, sondern alle Pfade in beliebiger Reihenfolge. Das bedeutet, dass an dieser Stelle

eine Nebenläufigkeit erzeugt werden muss. Für diesen Fall bietet SBP eine spezielle Art von Szenario. Die Klasse `ParallelCase` kann hier für jeden Pfad des Parallel Fragments initialisiert werden. Die Codebeispiele 47 und 48 zeigen eine Alternative in SML und ein passendes Mapping in Java. Das Beispiel verzweigt in zwei Pfade. Da allerdings beide parallel durchlaufen werden, können die Nachrichten in verschiedenen Reihenfolgen auftreten. Im Mapping werden dafür zwei Instanzen der Klasse `ParallelCase` erstellt. Diese speziellen Szenarien besitzen lediglich einen Body und sind direkt aktiviert. Die Pfade des Parallel Fragments werden in den Bodys dieser Szenarien implementiert. Die Methode `runParallel` startet diese beiden Szenarien und lässt das *Host*-Szenario warten, bis alle `ParallelCases` beendet sind.

```

1  parallel {
2    message requested car -> car.doA
3    message requested car -> car.doB
4  } and {
5    message requested car -> car.doC
6    message requested car -> car.doD
7  }

```

Code 47: Ein Parallel Fragment in SML.

```

1  // Begin Parallel
2  ParallelCase case_1 = new ParallelCase(this) {
3    @Override
4    protected void body() throws Violation {
5      request(car, car, "doA");
6      request(car, car, "doB");
7    }
8  };
9  ParallelCase case_2 = new ParallelCase(this) {
10   @Override
11   protected void body() throws Violation {
12     request(car, car, "doC");
13     request(car, car, "doD");
14   }
15 };
16 runParallel(case_1, case_2);
17 // End Parallel

```

Code 48: Das Mapping des Parallel Fragments in SBP.

5.2. Mapping für eine Collaboration

In SBP gibt es zwar keine Klasse als Pedant zur SML Collaboration, jedoch hat es sich als gute Methode erwiesen, dennoch ein Mapping dafür anzulegen. Durch das Supertyping in Java bietet es sich an, die Klasse `Scenario` als Collaboration zu verwenden. Es wird ein abstraktes *Collaboration-Szenario* definiert, welches lediglich die Rollen definiert. Alle Szenarien innerhalb der Collaboration erben dann von diesem Collaboration-Szenario anstatt von der Klasse `Scenario`. Das

5. SML-to-SBP Mapping

Codebeispiel 49 zeigt eine Collaboration mit Rollen. In Codebeispiel 50 ist das passende Mapping gegeben. Diese Methode erleichtert das Erstellen von Szenarien, da nicht immer wieder der selbe Code für die Rollen geschrieben werden muss. Außerdem ist der Code so lesbarer, da er wesentlich kürzer ist.

```
1 collaboration CarsRegisterWithObstacleControl {
2   dynamic role Environment env
3   dynamic role Car car
4   dynamic role Car nextCar
5   dynamic role Dashboard dashboard
6   dynamic role Obstacle obstacle
7   dynamic role ObstacleControl obstacleControl
8   dynamic role LaneArea currentArea
9   dynamic role LaneArea nextArea
10 }
```

Code 49: Ein Parallel Fragment in SML.

```
1 public abstract class CarsRegisterWithObstacleControlCollaboration extends Scenario {
2   // Roles
3   protected Role<Environment> env = createRole(Environment.class, "env");
4   protected Role<Car> car = createRole(Car.class, "car");
5   protected Role<Car> nextCar = createRole(Car.class, "nextCar");
6   protected Role<Dashboard> dashboard = createRole(Dashboard.class, "dashboard");
7   protected Role<Obstacle> obstacle = createRole(Obstacle.class, "obstacle");
8   protected Role<ObstacleControl> obstacleControl = createRole(ObstacleControl.class,
9     "obstacleControl");
9   protected Role<LaneArea> currentArea = createRole(LaneArea.class, "currentArea");
10  protected Role<LaneArea> nextArea = createRole(LaneArea.class, "nextArea");
11 }
```

Code 50: Das Mapping des Parallel Fragments in SBP.

5.3. Mapping für eine Spezifikation

Eine Spezifikation in SML beinhaltet Collaborations, in denen sich die Szenarien befinden. In SBP beinhaltet eine Spezifikation lediglich die Szenarien. Ein Mapping ist hier demnach relativ simpel: Für jedes Szenario in jeder Collaboration wird ein passender Eintrag des Szenarios in der `registerScenarios` Methode der Java Spezifikation erstellt. Beinhaltet die SML Spezifikation *Spec1* beispielsweise die Specification Szenarien *Scenario1*, *Scenario2* und *Scenario3*, so ist das Mapping in Codebeispiel 51 passend.

```
1 public class Spec1 extends Specification {
2   @Override
3   protected void registerScenarios() {
4     // Collaboration Collaboration1
5     registerSpecificationScenario(Scenario1.class);
6     registerSpecificationScenario(Scenario2.class);
7     registerSpecificationScenario(Scenario3.class);
8   }
9
10  @Override
11  protected void registerTransformationRules() {}
12 }
```

Code 51: Das Mapping einer SML Spezifikation mit drei Szenarien in SBP.

5.4. Mapping für eine Runconfiguration

Eine Runconfiguration gibt an, wie die Spezifikation ausgeführt werden soll. Dazu gehört die Wahl des Objektsystems. In SML gibt eine Runconfiguration genau diese Dinge an. In SBP hat sie noch mehr Einstellungen, die jedoch für das Mapping erst einmal irrelevant sind. Das Mapping übernimmt das Objektsystem der SML Spezifikation, indem ein Wrapper dafür erstellt wird (siehe Abschnitt 4.4). Über diesen Wrapper werden die Objekte als *controllable* oder *uncontrollable* in der SpecificationRunconfig eingetragen. Ob ein Objekt *controllable* oder *uncontrollable* ist, kann aus der Spezifikation entnommen werden, die in der Runconfiguration angegeben ist. Ein Mapping könnte beispielsweise aussehen wie in Codeblock 52.

```

1  @Override
2  protected void registerParticipatingObjects() {
3      // Objektsystem-Wrapper für das EMF Model
4      CarToXObjectSystem objectSystem = CarToXObjectSystem.getInstance();
5      // JamesCar
6      registerObject(objectSystem.CarToX.getCars().get(0), CONTROLLABLE);
7      // JamesCarDashboard
8      registerObject(objectSystem.CarToX.getCars().get(0).getDashboard(), UNCONTROLLABLE);
9      // James
10     registerObject(objectSystem.CarToX.getCars().get(0).getDriver(), UNCONTROLLABLE);
11     // env
12     registerObject(objectSystem.CarToX.getEnvironment(), UNCONTROLLABLE);
13     // obstacle
14     registerObject(objectSystem.CarToX.getStreetSections().get(0).getObstacles().get(0)
15                    , CONTROLLABLE);
16     // obstacleControl
17     registerObject(objectSystem.CarToX.getObstacleControls().get(0), CONTROLLABLE);
18 }

```

Code 52: Das Mapping einer SML Runconfiguration.

5.5. Codegenerator für SML

Ich habe einen Codegenerator implementiert, der das angegebene Mapping verwendet, um eine bestehende SML Spezifikation in eine SBP Spezifikation umzuwandeln. Dabei wird eine Runconfig-Datei eingelesen und eine Repräsentation dieser erstellt. Dann wird die Spezifikation mit den enthaltenen Szenarien transformiert und ein Wrapper für das Objektsystem erstellt. Die entstandene SBP Spezifikation ist direkt ausführbar. Zusätzlich kann eine GUI generiert werden, die eine Simulation der Spezifikation sofort möglich macht. Für die Verwendung

5. *SML-to-SBP Mapping*

des Codegenerators siehe Anhang A.4. Die Implementierung des Codegenerators erfolgt in der Sprache Xtend.

6. Methodik

SBP ermöglicht das szenariobasierte Programmieren von verteilten reaktiven Systemen in Java. Durch die Vielseitigkeit von Java als General Purpose Language bietet dies eine Vielfalt an Einsatzmöglichkeiten. Es ist nun problemlos möglich, in eine Spezifikation eigenen Code einzubetten, der das Zielsystem direkt beeinflusst. Zum Beispiel kann durch die Szenarien eine GUI-Anwendung aktualisiert oder Fahrzeug- bzw. Roboteraktoriik gesteuert werden. Außerdem kann die szenariobasierte Anwendung zwecks einer Simulation einfach durch einen Benutzer gesteuert werden. Java bietet zudem verschiedene Möglichkeiten, ein Objektsystem zu implementieren. Im Folgenden stelle ich Vorzüge von SBP heraus und beschreibe Methodiken für die effektive Verwendung von SBP.

6.1. Einbinden von UI und anderer Software

Eine Benutzeroberfläche (GUI) ist für viele Anwendungsbereiche hilfreich, sei es zum Überwachen der szenariobasierten Anwendung, für deren Steuerung oder für die Benutzung durch den Endanwender. Durch die Beschaffenheit der SBP Bibliothek ist es einfach, eine passende GUI in die Anwendung zu integrieren. Für eine GUI sind zwei funktionale Dinge besonders wichtig: Das Anzeigen von Daten, was einen Zugriff auf den Zustand der Anwendung nötig macht. Dazu ist ein Updatemechanismus nötig, der die Anzeige der GUI bei Änderungen des Zustands aktualisiert. Als zweites ist eine Steuerung der Anwendung über die GUI wichtig. Dazu müssen Benutzereingaben auf der GUI an die szenariobasierte Anwendung weitergegeben werden können (siehe Abschnitt 6.2). Für diese Anwendungsfälle bietet SBP bestimmte Schnittstellen, die ich im Folgenden anhand des Car-To-X Beispiels genauer erläutere:

- **Einbinden einer GUI in die Anwendung:** Eine GUI für die Anzeige und Steuerung des Car-To-X Systems benötigt Elemente, die zeigen, wo sich ein Fahrzeug momentan befindet und welche Aktionen es ausführen kann. Abbildung 6.1 zeigt Screenshots zweier Zustände einer GUI, die die Elemente des Car-To-X Systems visuell darstellt und damit den Zustand des Systems zeigt. Zudem erkennt die GUI bestimmte Zustände des Systems, in denen eine gewisse Handlung der Umwelt erwartet wird. Die

6. Methodik

visuellen Elemente der GUI sind in gewissen Zuständen farblich markiert, um Ereignisse der Umwelt darzustellen. Damit die GUI nun den Systemzustand darstellen kann, benötigt sie Zugriff auf das Objektsystem. Dies ist über die `SpecificationRunconfig` möglich, da dort das Objektsystem erstellt oder registriert wird. Da die Klasse `SpecificationRunconfig` beim Start der szenariobasierten Anwendung initialisiert wird, bietet es sich an, die GUI im Konstruktor der Klasse ebenfalls zu initialisieren und dort das Objektsystem zu übergeben.

- **Aktualisierung der GUI:** Um die GUI zu aktualisieren bietet SBP die Möglichkeit, das Verhalten der Anwendung zu überwachen. Dafür kann eine Klasse implementiert werden, die `ScenarioObserver` erweitert. Diese Klasse kann als Beobachter der Anwendung in der `SpecificationRunconfig` eingetragen werden. Sie wird immer dann benachrichtigt, wenn ein Ereignis aufgetreten ist. Je nachdem, welches Ereignis das war, kann die GUI darauf reagieren und die Anzeige des Objektsystems aktualisieren.

6.2. Einbinden einer Steuerung der Umgebung

Ist die GUI in die Anwendung integriert, fehlt nur noch die Steuerung durch den Benutzer, also das Einspeisen von Ereignissen in die Anwendung. Die Klasse `SpecificationRunconfig` definiert einen Runtime-Adapter, der für das Senden und Empfangen von Ereignissen zuständig ist. Dieser kann verwendet werden, um selbstständig Ereignisse in die Anwendung einzuspeisen und damit beispielsweise die Umwelt zu steuern. Betrachten wir die GUI in Abbildung 6.1 erneut, so erweitern wir die visuellen Elemente des Systems um eine Steuerung. Die GUI erkennt bereits bestimmte Situationen des Objektsystems und kann die benötigten Ereignisse der Umwelt farblich darstellen. Diese farblich markierten Elemente der GUI werden um eine `OnClick`-Aktion erweitert. Das bedeutet, dass der Benutzer auf sie klicken kann, um das farblich dargestellte Ereignis der Umwelt auszuführen. Die GUI erzeugt nun dieses Ereignis und gibt es an die Methode `receive` des Runtime-Adapters weiter. Dadurch wird das Ereignis in die szenariobasierte Anwendung eingespeist und das System kann darauf reagieren. Im Falle einer verteilten Ausführung ist dabei darauf zu achten, dass die GUI-Komponente die Objekte, die gesteuert werden sollen, als `controllable` deklariert. Nur so werden die Ereignisse über den Runtime-Adapter auch an die anderen Komponenten weitergegeben. Abbildung 6.2 zeigt die Architektur der vier Komponenten des Systems mit ihren Kommunikationswegen. Es gibt zwei Fahrzeuge, eine Baustelle und unsere GUI, die die Umwelt kontrolliert. Diese

6.3. Überwachung des Nachrichtenverlaufes

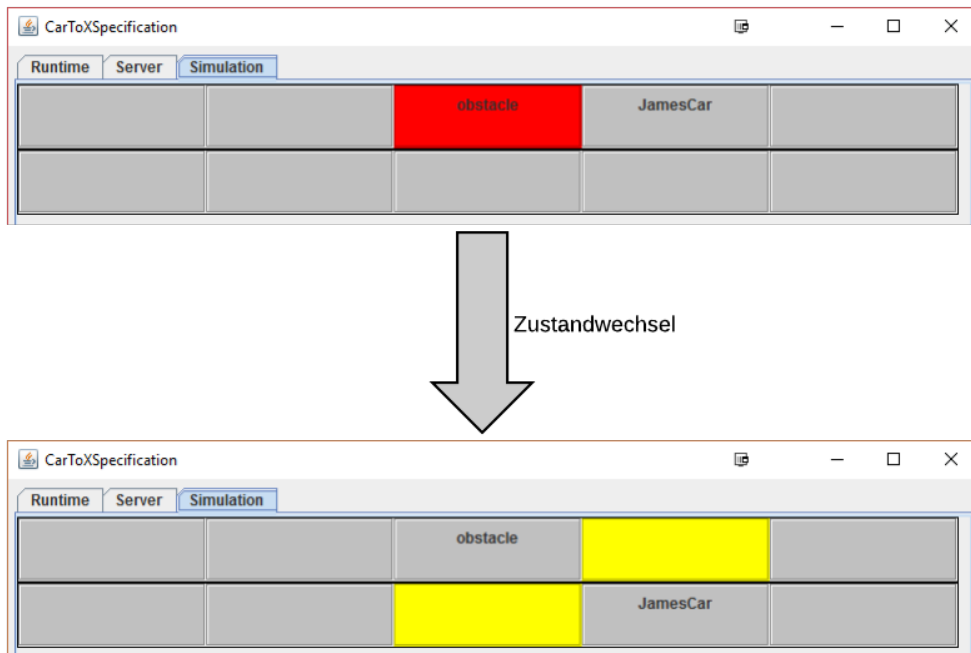


Abbildung 6.1.: Eine GUI zur Simulation des Car-To-X Beispiels. Sie stellt zwei Straßenseiten dar, die in Sektionen eingeteilt sind. Ein Fahrzeug (*JamesCar*) bewegt sich auf diesen Sektionen. Eine Baustelle (*obstacle*) blockiert ein der Sektionen. Farblich markierte Sektoren stellen Nachrichten der Umwelt dar. Rot bedeutet, das Fahrzeug erkennt die Baustelle. Gelb bedeutet, das Fahrzeug bewegt sich auf eine andere Sektion.

Methode benötigt natürlich keine GUI. Stattdessen kann auch beispielsweise ein Sensor oder Algorithmus Ereignisse in die Anwendung einspeisen und so die Umwelt steuern. Dies funktioniert dann über den selben Weg - den Runtime-Adapter.

6.3. Überwachung des Nachrichtenverlaufes

Wie in SCENARIOTOOLS kann es auch in SBP interessant sein, den Nachrichtenverlauf auszulesen. SCENARIOTOOLS bietet hier ein textuelles Logging und einen grafischen Zustandsgraph. SBP bietet ebenfalls ein textuelles Logging

6. Methodik

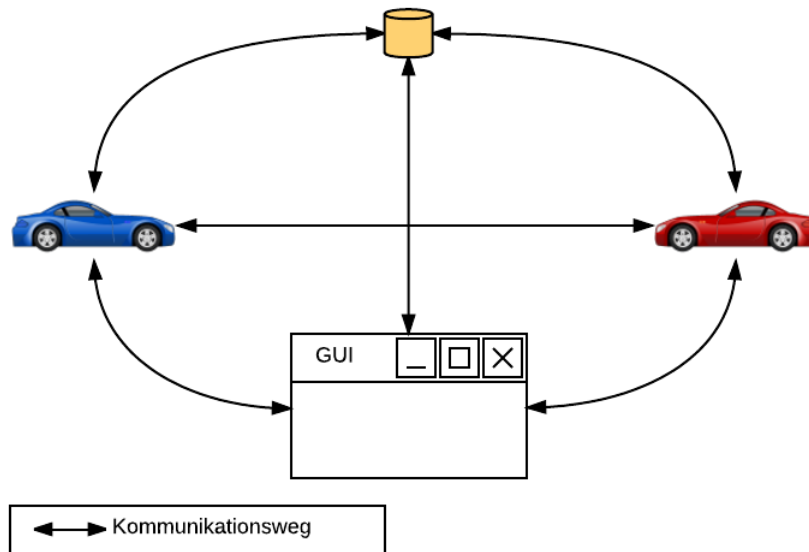


Abbildung 6.2.: Die vier Komponenten des Car-To-X Systems mit ihren Kommunikationswegen. Das System besteht hier aus zwei Fahrzeugen, einer Baustelle und einer GUI, die die Umgebung kontrolliert.

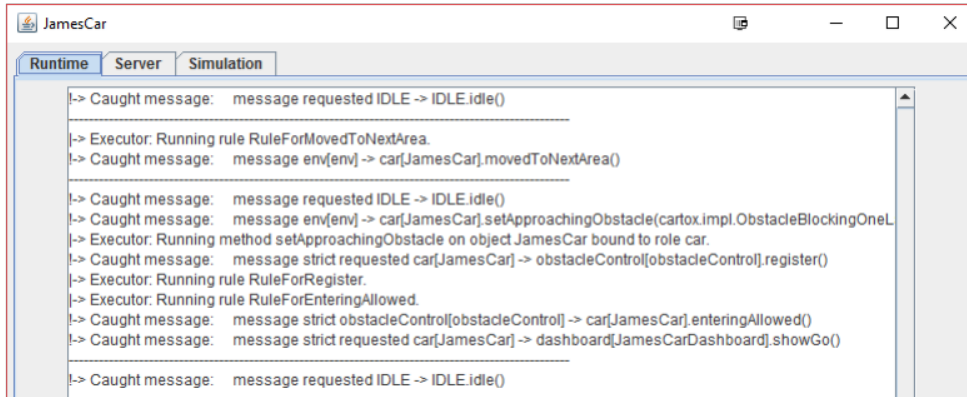
der auftretenden Ereignisse. In diesem Log werden Ereignisse, Violations und Transformationen angezeigt, die während der Ausführung auftreten. Dieser Log wird standardmäßig auf `System.out` geleitet, kann allerdings durch die Klasse `Settings` beliebig umgeleitet werden. Durch diese Klasse können Lognachrichten an- oder ausgeschaltet werden. Außerdem können Lognachrichten nach Kategorien in verschiedene Outputstreams geleitet werden. Die Abbildung 6.3 zeigt den Screenshot einer GUI, mit zwei Logausgaben. Beim Logging wird hier zwischen Ereignissen in der szenariobasierten Anwendung und dem Netzwerktransfer unterschieden.

6.4. Bedeutung von Assumptions in SBP

Assumptions stellen Annahmen an die Umwelt in Form von Szenarien dar. In `SCENARIOTOOLS` dient dies primär dem Zweck, die Umwelt korrekt simulieren zu können. Die Assumptions bedingen, wie die Umwelt für die Simulation zu steuern ist. Implementiert man die Spezifikation jetzt in SBP und bringt sie auf

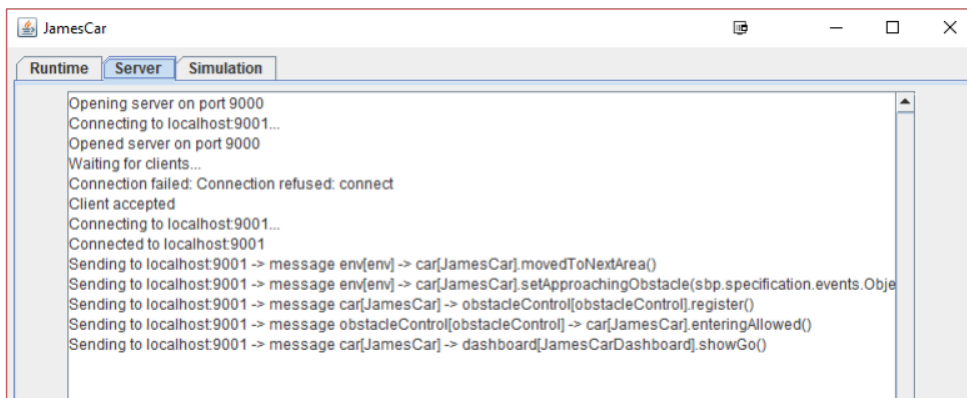
6.4. Bedeutung von Assumptions in SBP

Ereignisse der Runtime



```
JamesCar
Runtime Server Simulation
|-> Caught message: message requested IDLE -> IDLE.idle()
-----
|-> Executor: Running rule RuleForMovedToNextArea.
|-> Caught message: message env[env] -> car[JamesCar].movedToNextArea()
-----
|-> Caught message: message requested IDLE -> IDLE.idle()
|-> Caught message: message env[env] -> car[JamesCar].setApproachingObstacle(carrox.impl.ObstacleBlockingOneL
|-> Executor: Running method setApproachingObstacle on object JamesCar bound to role car.
|-> Caught message: message strict requested car[JamesCar] -> obstacleControl[obstacleControl].register()
|-> Executor: Running rule RuleForRegister.
|-> Executor: Running rule RuleForEnteringAllowed.
|-> Caught message: message strict obstacleControl[obstacleControl] -> car[JamesCar].enteringAllowed()
|-> Caught message: message strict requested car[JamesCar] -> dashboard[JamesCarDashboard].showGo()
-----
|-> Caught message: message requested IDLE -> IDLE.idle()
```

Netzwerkkommunikation



```
JamesCar
Runtime Server Simulation
Opening server on port 9000
Connecting to localhost:9001...
Opened server on port 9000
Waiting for clients...
Connection failed: Connection refused: connect
Client accepted
Connecting to localhost:9001...
Connected to localhost:9001
Sending to localhost:9001 -> message env[env] -> car[JamesCar].movedToNextArea()
Sending to localhost:9001 -> message env[env] -> car[JamesCar].setApproachingObstacle(sbp.specification.events.Obj
Sending to localhost:9001 -> message car[JamesCar] -> obstacleControl[obstacleControl].register()
Sending to localhost:9001 -> message obstacleControl[obstacleControl] -> car[JamesCar].enteringAllowed()
Sending to localhost:9001 -> message car[JamesCar] -> dashboard[JamesCarDashboard].showGo()
```

Abbildung 6.3.: Eine GUI für das Logging in SBP. Der obere Log zeigt Ereignisse der lokalen szenariobasierten Ausführung. Der untere Log zeigt die Netzwerkkommunikation.

das Zielsystem, muss die Umwelt nicht mehr simuliert werden. Die Umwelt, die zuvor noch gesteuert werden musste, wird nun durch die reale Umwelt - wie zum Beispiel Sensoren oder Benutzereingaben - ersetzt. Dies macht Assumptions somit erst einmal obsolet. Dennoch können Assumptions in SBP implementiert und während der verteilten Ausführung verwendet werden. Dies hat auch einen Zweck. Assumptions können hilfreich sein, Fehlerquellen im Zielsystem zu identifizieren. Auch wenn die Spezifikation in SCENARIO TOOLS oder in SBP erfolg-

6. Methodik

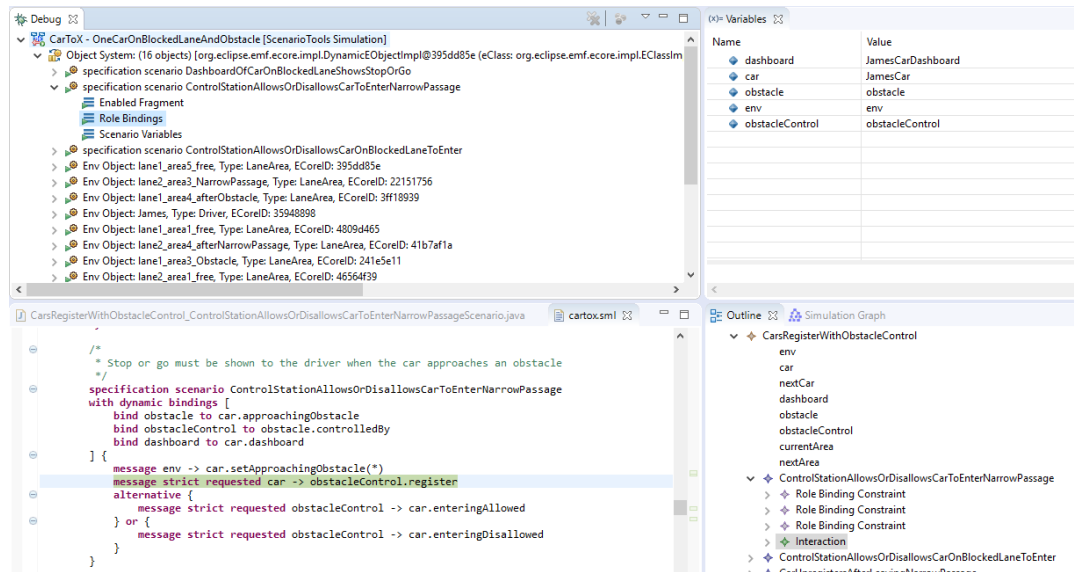


Abbildung 6.4.: Eine Ansicht der Debuggingumgebung von SCENARIOTOOLS.

reich simuliert wurde, kann es auf dem Zielsystem immer noch zu Problemen kommen. Wurde zum Beispiel die Umwelt anfangs falsch oder nicht vollständig modelliert, kann dies eine Fehlerquelle sein. Denn wenn die Assumptions in SML schon nicht der Realität entsprachen, kann SCENARIOTOOLS den Fehler nicht finden. Tritt also in der verteilten Ausführung auf dem Zielsystem ein Fehler auf, kann dies auf eine Unstimmigkeit in der Umwelt hinweisen. Lassen wir also die Assumption-Szenarien weiterhin während der Ausführung mitlaufen, können wir durch Verletzungen dieser Szenarien auf Fehlerquellen außerhalb des Systems schließen. Softwaresysteme sind zudem oft mit Updatemechanismen versehen, die auch zu Änderungen des Verhaltens der Umwelt führen können. Wenn das System nicht entsprechend angepasst wird, können wir auch durch die Assumptions ein Fehlverhalten erfassen.

6.5. Debugging mit SBP

Bei der Implementierung in SBP ist es wichtig, auch Möglichkeiten des Debuggings zu haben. SCENARIOTOOLS bietet eine Simulation, die den Ingenieur Schritt für Schritt durch die Szenarien führt. Dadurch kann er in jedem Systemzustand herausfinden, welche Szenarien aktiv, welche Nachrichten enabled und welche Rollen gebunden sind. Abbildung 6.4 zeigt das Debugging in SML mit SCENARIOTOOLS.

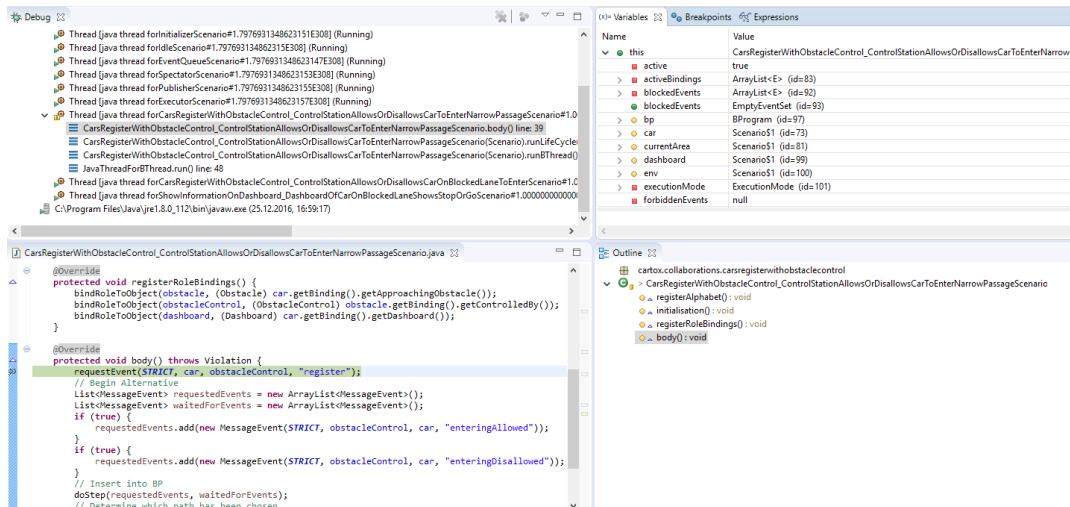


Abbildung 6.5.: Eine Ansicht der Debuggingumgebung von Eclipse für SBP.

In SBP läuft diese Simulation oder auch die verteilte Ausführung nicht in einer speziellen Simulationsumgebung, wie in SCENARIOTOOLS. Allerdings basiert die SBP Bibliothek vollständig auf Java und bietet damit eine Vielzahl von Debuggern. Beispielsweise kann die Debuggingumgebung von Eclipse verwendet werden, um ein ähnlich umfangreiches Debugging zu ermöglichen, wie mit SCENARIOTOOLS. Die aktiven Szenarien in SBP sind jeweils einzelne Threads, deren Ausführung immer an einer bestimmten Zeile des Szenarios-Quelltextes steht. Über die Eclipse Debuggingumgebung kann für jeden Prozess einzeln der aktuelle Stack-Frame angezeigt werden. Dadurch kann man in die Zeile springen, die gerade in einem Szenario bearbeitet wird. Dies kann eine Nachricht sein, die gerade mittels `request` oder `waitFor` gefordert wird, oder eine Condition, die das Szenario aufhält. Welche Szenarien gerade aktiv sind, verrät die Prozessübersicht. Da jedes Szenario in einem eigenen Thread läuft, taucht es auch in der Prozessübersicht auf. Dadurch kann für jedes Szenario einzeln ein Debugging durchgeführt werden. Die Variablen eines Methodenaufrufes innerhalb eines Threads können ebenfalls über Bordmittel der Debuggingumgebung analysiert werden, um Rollenbindungen und Variablenzuweisungen zu überwachen. Abbildung 6.5 zeigt das Debugging des Szenarios aus Abbildung 6.4 in SBP.

Dies bietet dasselbe ausführliche Debugging, wie SCENARIOTOOLS. Außerdem bietet das Debugging in SBP durch Breakpoints eine weitere Möglichkeit, die SCENARIOTOOLS bisher nicht bietet. Breakpoints sind markierte Zeilen im Quellcode, ab denen die Ausführung eines Programms pausiert wird, um in ein

6. Methodik

Debugging zu wechseln. So kann die Ausführung der Szenarien normal ablaufen, bis eine vom Tester bestimmte Programmzeile erreicht ist, ab der ein Debugging gestartet wird. Es ist damit noch einfacher, nur bestimmte Szenarien oder sogar einzelne Schritte innerhalb der Szenarien zu überwachen.

6.6. Objektmodelle ohne EMF

Ebenso wie SML benötigt SBP für die Ausführung ein Objektsystem, das die Objekte und den Startzustand des Systems vorgibt. Für dieses Objektsystem gibt es in SBP keine festen Vorgaben. Bei der Implementierung kann hier eine beliebige Struktur von Objekten angegeben werden. Wichtig ist es, alle Objekte, die an Rollen gebunden werden sollen, vorher in der `SpecificationRunconfig` zu registrieren. Ein einfaches Objektsystem kann wie in Codebeispiel 53 aussehen. Das Beispiel zeigt eine Klasse, die mehrere Objekte des Car-To-X Systems initialisiert und in öffentlichen Feldern speichert. Natürlich können hier im Sinne der Datenkapselung auch Getter und Setter an Stelle von öffentlichen Feldern verwendet werden. In der `SpecificationRunconfig` werden diese Felder dann als `controllable` oder `uncontrollable` angegeben.

```
1 public class CarToXObjectSystem {
2
3     public Car car = new Car();
4     public Driver driver = new Driver();
5     public Obstacle obstacle = new Obstacle();
6     public Obstaclecontrol obstaclecontrol = new Obstaclecontrol();
7
8     ...
9
10 }
```

Code 53: Beispiel eines simplen Objektsystems ohne EMF.

6.7. Objektmodelle aus EMF importieren

Will man eine SML Spezifikation mit Hilfe von SBP verteilt ausführbar machen, so liegt meist bereits ein Objektsystem in Form einer dynamischen Instanz eines EMF-Models vor. Dies kann für SBP ebenfalls als Objektsystem verwendet werden. Dafür muss diese Instanz lediglich von einer Methode des SBP Objektsystems geladen und wiederum für die `SpecificationRunconfig` bereitgestellt werden. Das Codebeispiel 54 zeigt eine Klasse für ein SBP Objektsystem mit einer Methode zum Laden der dynamischen Instanz eines EMF-Models. Dies ist die empfohlene Methode zur Verwendung von EMF-Objektsystemen aus SML

(siehe auch Abschnitt 4.4). Im Sinne der modellbasierten Entwicklung schlage ich dieses Verfahren nicht nur für die Übersetzung von SML nach SBP vor, sondern empfehle bei der szenariobasierten Entwicklung in SBP ebenfalls ein solches modellbasiertes Objektsystem zu verwenden.

```

1 public class CarToXObjectSystem {
2
3     private static CarToXObjectSystem instance;
4
5     public static CarToXObjectSystem getInstance() {
6         if (instance == null)
7             instance = new CarToXObjectSystem();
8         return instance;
9     }
10
11     public CarToX carToX = (CarToX) loadCarToX("cartox.xmi");
12
13     public CarToX loadCarToX(String uri) {
14         // Load models into the JVM
15         CartoxPackage.eINSTANCE.eClass();
16
17         // Register xmi file extension
18         Resource.Factory.Registry reg = Resource.Factory.Registry.INSTANCE;
19         Map<String, Object> m = reg.getExtensionToFactoryMap();
20         m.put("xmi", new XMIResourceFactoryImpl());
21
22         // Load xmi file and return object system
23         ResourceSet resSet = new ResourceSetImpl();
24         Resource resource = resSet.getResource(URI.createURI(uri), true);
25         CarToX carToX = (CarToX) resource.getContents().get(0);
26         return carToX;
27     }
28
29 }

```

Code 54: Beispiel eines Objektsystems, das eine dynamische EMF-Instanz lädt.

6.8. Objekterzeugung und -zerstörung

Ein großes Problem von SCENARIOTOOLS ist die Objekterzeugung und die Objektzerstörung. Durch die Art der Analyse der SML Spezifikationen unterliegt das Objektsystem bestimmten Restriktionen, die es verbieten, während der Laufzeit Objekte zu zerstören oder neue Objekte hinzuzufügen. Das liegt daran, dass bei einer vollständigen Analyse des Zustandsraums eine Zustandsraumexplosion auftreten kann. Erzeugt man in einer Spezifikation einen Ausführungspfad, in dem immer neue Objekte erzeugt werden, die dann weitere Aktionen durchführen, gibt es kein Ende des Zustandsraums und die Analyse findet niemals ein Ende. Da wir in SBP keine Analyse des Zustandsraums durchführen wollen, kann diese Restriktion hier aufgehoben werden. SBP ermöglicht es daher, sehr wohl neue Objekte der Ausführung hinzuzufügen oder Objekte zu zerstören. Dies kann in Situationen sinnvoll sein, wenn wir von

6. Methodik

großräumigen Systemen ausgehen. Beispielsweise kann eine Baustelle auf Dauer viele Fahrzeuge neu kennenlernen (Objekterzeugung) und wieder vergessen (Objektzerstörung). Ein anderes Beispiel, in denen eine solche Modellierung nötig ist, sind Fabrikanlagen, die Objekte herstellen und weiter verarbeiten. Diese Objekte können jetzt als tatsächliche Objekte im Objektsystem modelliert werden. Werden neue Objekte erzeugt oder gelöscht, muss allerdings strikt auf drei Dinge geachtet werden:

- **Registrierung der Objekte:** Damit Objekte an Rollen gebunden werden können, müssen sie in der `SpecificationRunconfig` registriert werden. Während der Laufzeit ist dies allerdings nicht mehr möglich. Neu erzeugte Objekte müssen also bei ihrer Erzeugung manuell in der `ObjectRegistry` registriert werden. Dadurch erhalten diese Objekte IDs für das Binden an Rollen.
- **Zerstören der Objekte:** Werden Objekte zerstört, können diese aus der `ObjectRegistry` gelöscht werden. Dabei ist allerdings darauf zu achten, dass diese Objekte in keinem Szenario mehr gebunden sind. Ist dies nämlich doch der Fall, kann es zu schweren Fehlern in den Szenarien kommen. Durch die Löschung der Objekte in der `ObjectRegistry` sind die Rollenbindungen nicht mehr gültig und die Rollen somit nicht mehr gebunden.

Werden diese Regeln eingehalten, löst SBP ein großes Problem in der Modellierung in SCENARIOTOOLS- die dynamische Erzeugung oder Zerstörung von Objekten.

6.9. Testing

Bei der Entwicklung von Softwaresystemen ist das Testen immer ein äußerst wichtiges Thema. Der Prozess des Testens stellt sich für verschiedene Entwicklungsverfahren immer unterschiedlich dar. Im Falle der szenariobasierten Entwicklung können Tests in der Form einer gezielten Ausführung der Szenarien durchgeführt werden. SCENARIOTOOLS selbst bietet mit der Methodik der Synthese die Möglichkeit zu testen, ob ein modelliertes System realisierbar ist. Dies stellt jedoch keine zufriedenstellende Methode dar, einzelne Szenarien oder situative Abläufe des Systems zu testen. Es fehlt eine einheitliche Methodik zum Testen von Szenarien. Die auf Java basierende SBP Bibliothek kann verwendet werden, um SBP Spezifikationen und damit Szenarien mittels JUnit Testfällen zu testen. Eine Methode, die sich hier anbietet, ist die der Test-Szenarien. Eine

Spezifikation kann beliebig viele verschiedene Specification- und Assumption-Szenarien beinhalten. Bei dem Aufruf der Spezifikation in der `SpecificationRunconfig` können zusätzlich Test-Szenarien eingebunden werden, deren Ausführung von JUnit Testfällen überwacht werden kann. Das Codebeispiel 55 zeigt ein Beispiel eines Test-Szenarios, welches die Funktionalität des Dashboards vom Fahrzeug im Car-To-X Beispiel überprüft. Laut der Anforderung soll das Dashboard Go anzeigen, wenn sich das Fahrzeug der Baustelle nähert und fahren darf. Für die Ausführung des Tests laden wir ein Objektsystem, in dem es lediglich das eine Fahrzeug gibt. Dadurch ist die Bedingung der Anforderung generell erfüllt und das Dashboard sollte die Nachricht `showGo` erhalten. Das Test-Szenario spielt exakt einen Lauf des Systems durch, indem es eine gewisse Folge von Nachrichten erfordert. Wird das Test-Szenario an irgendeiner Stelle verletzt - es tritt eine Safety- oder Interrupt-Violation auf - folgt ein Aufruf zum Scheitern des Testfalls. Betrachten wir das Test-Szenario etwas genauer:

- Die Rollenbindungen sind strikt vorgegeben, und beziehen sich auf das Objektsystem mit einem Fahrzeug. Sie sind also für genau die Situation gebunden, die das Szenario testen soll. Hier können noch beliebige andere Rollen gebunden werden, die durch Conditions oder Ereignisse zum Fehlschlag des Tests führen können.
- Das Alphabet ist ebenfalls strikt vorgegeben und beinhaltet all die Nachrichten, die während der Ausführung einen Fehler verursachen können. Hier können beliebige andere Nachrichten hinzugefügt werden, die für den Fehlschlag des Testfalls sorgen sollen.
- Der Body des Szenarios beinhaltet die Nachrichten, die getestet werden sollen. Es fällt auf, dass Systemnachrichten als `waitFor` und Umgebungsnachrichten als `requested` eingetragen sind. Dies kann als Schema betrachtet werden und hat folgenden Zweck: Die Systemnachrichten sollen getestet werden, was bedeutet, dass sie in den Specification-Szenarien gefordert werden müssen. Darum sollte der Test diese Nachrichten lediglich erwarten. Die Umgebung muss allerdings durch das Test-Szenario angetrieben werden. Darum werden speziell für das Test-Szenario alle Objekte als kontrollierbar betrachtet, um auch Umgebungsereignisse ausführen zu können. Würde das Test-Szenario auf Umgebungsereignisse warten, müsste die Umgebung zusätzlich an einer anderen Stelle gesteuert werden. Dies kann allerdings ebenfalls durch einen zusätzlichen Thread geschehen (siehe Abschnitt 6.2).

6. Methodik

```
1 public class TestScenario_DashboardOfJamesCarShowsGo extends
2     CarsRegisterWithObstacleControlCollaboration {
3
4     @Override
5     protected void initialisation() {
6         // No initialization
7     }
8
9     @Override
10    protected void registerRoleBindings() {
11        bindRoleToObject(car, CarToXObjectSystem.getInstance().CarToX.getCars().get(0));
12        // JamesCar
13        bindRoleToObject(env, CarToXObjectSystem.getInstance().CarToX.getEnvironment());
14        // Environment
15        bindRoleToObject(dashboard, (Dashboard) car.getBinding().getDashboard()); //
16        // JamesCar
17        // Dashboard
18        bindRoleToObject(obstacleControl, CarToXObjectSystem.getInstance().CarToX.
19        getObstacleControls().get(0)); // Obstacle
20        // Control
21        bindRoleToObject(obstacle, (Obstacle) obstacleControl.getBinding().
22        getControlledObstacle()); // Obstacle
23    }
24
25    @Override
26    protected void registerAlphabet() {
27        setBlocked(env, car, "movedToNextArea");
28        setBlocked(env, car, "setApproachingObstacle", obstacle.getBinding());
29        setBlocked(car, obstacleControl, "register");
30        setBlocked(obstacleControl, car, "enteringAllowed");
31        setBlocked(car, dashboard, "showGo");
32        setBlocked(obstacleControl, car, "enteringDisallowed");
33        setBlocked(car, dashboard, "showStop");
34    }
35
36    @Override
37    protected void body() throws Violation {
38        // Retrieve targeted obstacle control from object system
39        request(new Message(env, car, "movedToNextArea"));
40        request(STRICT, new Message(env, car, "setApproachingObstacle", obstacle.
41        getBinding()));
42        waitFor(STRICT, car, obstacleControl, "register");
43        waitFor(STRICT, new Message(obstacleControl, car, "enteringAllowed"));
44        waitFor(STRICT, car, dashboard, "showGo");
45    }
46
47    @Override
48    protected void catchSafetyViolation(SafetyViolation e) {
49        super.catchSafetyViolation(e);
50        fail(e.getMessage());
51    }
52
53    @Override
54    protected void catchInterruptViolation(InterruptViolation e) {
55        super.catchInterruptViolation(e);
56        fail(e.getMessage());
57    }
58
59 }
```

Code 55: Test-Szenario zum Testen eines kontrollierten Ablaufes.

Für die Ausführung des Tests muss jetzt nur noch eine Konfiguration gestartet werden, die das Test-Szenario verwendet. Codebeispiel 56 zeigt den Aufruf des Test-Szenarios in einem JUnit-Testfall. Der Testfall startet eine Konfiguration, die auch für die lokale Simulation verwendet werden kann und speist das

6.10. Andere Protokolle für die verteilte Ausführung verwenden

Test-Szenario ein. Danach wartet der Testfall, bis das Szenario in der szenario-basierten Anwendung abgeschlossen wurde.

```
1 public class CarToXTest {
2
3     @Test
4     public void test() throws InterruptedException {
5         // Die übliche Konfiguration für die lokale Simulation
6         SpecificationRunconfig.run(CarToXRunconfig.class);
7
8         // Initialisierung des Test-Szenarios und Einspeisung in die Runtime
9         Scenario testScenario = new TestScenario_DashboardOfJamesCarShowsGo();
10        SpecificationRunconfig.getInstance().runTestScenario(testScenario);
11
12        // Warte, bis der Test abgeschlossen ist.
13        while(!testScenario.isFinished()) {
14            Thread.sleep(0);
15        }
16    }
17
18 }
```

Code 56: Aufruf des Tests mit dem Test-Szenario.

6.10. Andere Protokolle für die verteilte Ausführung verwenden

Nicht für jedes System ist es möglich, die verteilte Ausführung über Java-Sockets zu realisieren. Dafür bieten sich diverse andere Protokolle an, die beispielsweise über das Internet Daten senden. Ein konkretes Beispiel dafür ist das *MQTT*¹. Dieses leichtgewichtige Publish-Subscribe-Protokoll ist für die Maschine-zu-Maschine-Kommunikation über das Internet konzipiert. Es bietet die Möglichkeit, eine Art Chatraum (genannt Topic) zu betreten und darüber Nachrichten zu senden und zu empfangen. Dies kann für ein großflächig verteiltes System interessant sein, welches über mobile Netze am Internet angebunden ist. Dafür kann ein eigenständiger Runtime-Adapter konstruiert werden, der einen MQTT-Service implementiert und die Kommunikation über einen Topic leitet. Dafür müssen lediglich die Methoden wie in Abschnitt 4.6.1 implementiert und der MQTT-Service eingebettet werden.

¹<http://mqtt.org/>

6. Methodik

7. Implementierung

Die Java Bibliothek für das szenariobasierte Programmieren setzt auf die Bibliothek vom Behavioral Programming auf. Dieses Kapitel befasst sich mit der Implementierung der umgesetzten Konzepte, die in dieser Arbeit dargestellt werden. Zudem werden Herausforderungen, Lösungen und Designentscheidungen aufgezeigt.

7.1. Rollen und Bindings

Das Binden von Rollen an Objekte ist elementarer Bestandteil der SML Szenarien und deshalb auch der SBP Szenarien. In SBP werden Rollen mittels der Methode `bindRoleToObjekt` bei der Aktivierung eines Szenarios gebunden. Über die Methode `getBinding` der Rolle lässt sich wiederum das Objekt erhalten, an welches die Rolle gebunden ist. Die Rolle speichert allerdings nicht das Objekt, sondern eine spezielle ID, die einzigartig für jedes Objekt vorhanden ist. Diese ID stammt aus einem Mapping der Klasse `ObjectRegistry`. Diese Klasse hält IDs für die an der Ausführung teilnehmenden Objekte. Sie implementiert Methoden für das Mapping in beide Richtungen (ID -> Objekt und Objekt -> ID). Um dieses Mapping aufzubauen, gibt es die Methode `registerObject` in der Klasse `SpecificationRunconfig`. Dadurch werden die Objekte für die Ausführung registriert und es wird ihnen eine ID zugeordnet. Die Verwaltung der Rollenbindungen über IDs hat die Bewandnis, dass die Rollen dadurch für den Netzwerktransfer plättbar sind. Ereignisse, die während der verteilten Ausführung in einer Komponente auftreten, müssen an andere Komponenten in Form von Nachrichten weitergegeben werden. Die in SBP implementierte verteilte Ausführung versendet diese Nachrichten über Java Sockets. Dabei wird die Nachricht auf der einen Seite in das Netzwerk geschrieben und auf der anderen Seite gelesen. Diese Nachricht hat natürlich die Rollen für den Sender und den Empfänger. Abbildung 7.1 zeigt grafisch, was passiert, wenn Rollen mit Objekten oder mit IDs versendet würden. Nehmen wir einmal an, es gäbe keine IDs, sondern die Objekte würden mit den Rollen über das Netzwerk geschickt. Dies würde natürlich mehr Daten verursachen, die für jedes Ereignis transportiert werden müssen. Außerdem weiß die sendende Komponente (A), in

7. Implementierung

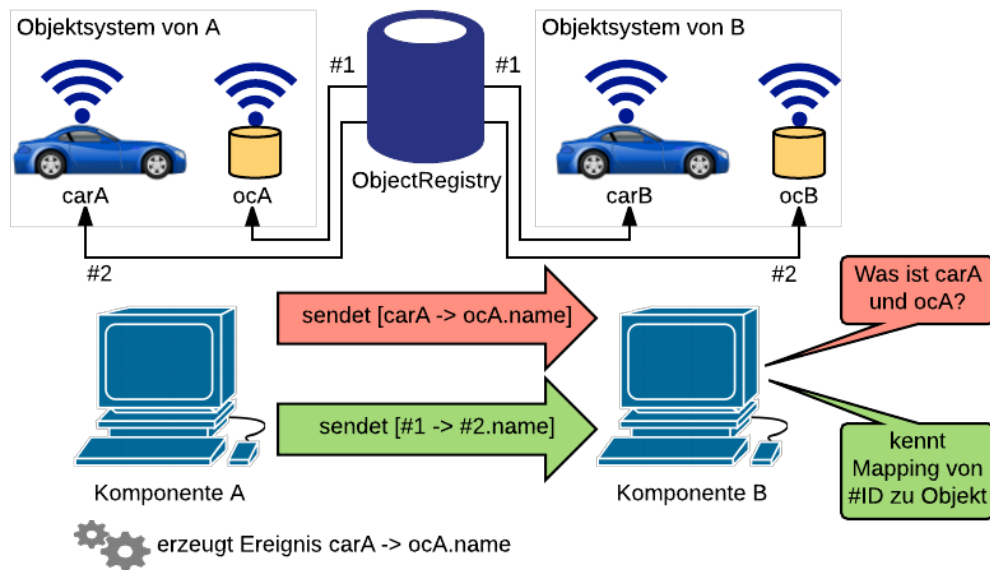


Abbildung 7.1.: Diese Grafik zeigt die Auswirkungen des Mappings von Objekten auf IDs bei der verteilten Ausführung. Komponente A sendet eine Nachricht an Komponente B. Einmal mit Objekten und einmal mit IDs.

welchem Zusammenhang die Objekte im Objektsystem stehen, allerdings weiß die empfangende Komponente (B) dies nicht und kann demnach nicht genau bestimmen, welche Objekte das tatsächlich sind. Jede Komponente hat ihr eigenes Objektsystem. Diese Objektsysteme sind zwar synchronisiert, jedoch sind es nicht **dieselben** Objekte, da sie in verschiedenen Java Virtual Machines existieren. Darum senden wir IDs anstatt der Objekte und sorgen dafür, dass die IDs in allen Komponenten identisch gemappt werden. Jetzt wissen die sendende und empfangene Komponente genau welche Objekte gemeint sind, da sie über simple Zahlen sprechen, die erst später wieder zu einem Objekt gemappt werden.

7.2. Nachrichten mit Rollen und Parametern

Die Grundlage der Ereignisse in SBP wird von Behavioral Programming gestellt. In BP werden Ereignisse durch die Klasse `Event` dargestellt, die erst einmal nur einen Namen enthält. Der Umfang der Nachrichten in SML übersteigt

allerdings bei Weitem den der Klasse `Event`, weshalb wir die Klasse `Message` einführen. Diese Klasse repräsentiert die SBP Nachricht. Sie referenziert zwei Rollen (Sender und Empfänger), sowie den Namen der Nachricht und eine beliebige Anzahl von Parametern. Zudem gibt es zwei boolesche Felder für die Modalitäten *strict* und *requested*. Für die Instanziierung dieser Klasse zeigt der Codeblock 57 unterschiedliche Konstruktoren für verschiedene Situationen.

```

1 // Eine Nachricht mit Sender, Empfänger und Name.
2 Message(Role sender, Role receiver, String message)
3 // Zusätzlich mit Parametern als Array.
4 Message(Role sender, Role receiver, String message, Object... parameters)
5 // Mit Modalitäten.
6 Message(boolean strict, boolean requested, Role sender, Role receiver, String message
7 )
8 // Mit Modalitäten und Parametern als Array.
9 Message(boolean strict, boolean requested, Role sender, Role receiver, String message
10 , Object... parameters)
11 // Eine Nachricht mit der Modalität strict und Parametern als Array.
12 Message(boolean strict, Role sender, Role receiver, String message, Object...
13 parameters)

```

Code 57: Konstruktoren der Klasse `Message`.

Die Klasse `Message` bietet zusätzliche Funktionen für die Laufzeit. Sie besitzt eine eigene `equals`-Methode, die auf Unifizierbarkeit prüft. Dazu kommen zwei Methoden `serializeForNetwork` und `deserializeForNetwork`, die sich um die Plättung der Rollen der Nachricht für das Netzwerk kümmern (siehe Abschnitt 7.10).

7.3. System-Szenarien

Während der Ausführung von SBP treten verschiedene Fälle auf, bei denen ein Eingreifen in oder Verwalten der Runtime nötig wird. Für diese Fälle habe ich verschiedene *System-Szenarien* entwickelt, die bereits beim Start der Anwendung Teil der Ausführung sind. Diese Szenarien werden niemals unterbrochen und sind niemals beendet. Sie laufen in einem ständigen Loop und beobachten die Ausführung. Im Folgenden beschreibe ich die verschiedenen System-Szenarien und ihre Anwendung.

- **Idle-Szenario:** Dieses Szenario zeigt das Ende eines lokalen Supersteps an. Ein Superstep ist beendet, wenn das System keine Systemnachrichten mehr anfordert, sondern auf eine Aktion der Umgebung wartet. Bei der verteilten Ausführung ist das System in Komponenten unterteilt, wodurch jede Komponente sich selbst als System und die anderen Komponenten als Umwelt ansieht. Dadurch sieht jede Komponente den Superstep aus seiner eigenen Perspektive und hat einen eigenen lokalen Superstep. Ist

7. Implementierung

ein (lokaler) Superstep beendet, aktiviert sich das Idle-Szenario und es veranlasst die Runtime zur Ausführung des Ereignisses *IDLE*. Dieses spezielle Ereignis sagt aus, dass der Superstep beendet ist und das System auf die Umwelt wartet. Es folgt die Aktivierung des Event-Queue-Szenarios.

- **Event-Queue-Szenario:** Dieses Szenario beobachtet die Ausführung und wartet auf das spezielle Ereignis *IDLE*. Das Szenario hat eine Queue von Nachrichten, die aus der Umgebung in das System gespeist werden sollen. Wird das *IDLE*-Ereignis ausgeführt, aktiviert sich das Event-Queue-Szenario und überprüft seine Queue. Wenn dort Nachrichten enthalten sind, wird die erste Nachricht der Queue ausgeführt und das Event-Queue-Szenario schläft wieder, bis *IDLE* erneut auftaucht.
- **Spectator-Szenario:** Ein Spectator beobachtet die Ausführung von Nachrichten und reagiert auf eine bestimmte Weise darauf. Eine bereits implementierte Version dieses Szenarios wird in SBP für das Logging verwendet. Der Spectator loggt das Verhalten der Anwendung auf einem Output-Stream (standardmäßig `System.out`). Das Spectator-Szenario wird ebenfalls dazu verwendet Szenario-Observer zu registrieren. Wird ein Observer in der `SpecificationRunconfig` registriert, beobachtet er standardmäßig das Spectator-Szenario. Dieses Szenario kann beliebig erweitert werden, um weitere reaktive Funktionalitäten zur Runtime hinzuzufügen.
- **Publisher-Szenario:** Der Publisher ist eine spezielle Form des Spectator-Szenarios. Er überwacht die Ausführung von Nachrichten und gibt diese über den Runtime-Adapter an die Komponenten der verteilten Ausführung weiter.
- **Initializer-Szenario:** Der Initializer ist ebenfalls eine spezielle Form des Spectator-Szenarios. Werden Nachrichten ausgeführt, oder von der Umwelt empfangen, überprüft dieses Szenario, ob dadurch ein Szenario der Spezifikation aktiviert wird. Ist dies der Fall, erstellt es eine Kopie des Szenarios und fügt diese Kopie der Runtime hinzu.
- **Executor-Szenario:** Der Executor ist der dritte spezielle Spectator. Er ist zuständig für die Modifikation des Objektsystems, indem er zwei Funktionen beinhaltet. Die erste ist die Ausführung von Methoden auf den Objekten des Objektsystems. Tritt ein Ereignis „register“ mit dem Senderobjekt „JamesCar“ und dem Empfängerobjekt „ObstacleControl“ auf, kann der Executor die Methode `register` der Klasse `Car` auf dem Objekt `JamesCar` aufrufen. Sofern diese implementiert ist, kann so domänenspezifischer Code in die Spezifikation eingebunden werden. Die zweite Methode

ist die Ausführung von Transformationsregeln. Diese Regeln können in der Spezifikation angegeben werden und werden zur Laufzeit dem Executor übergeben. Tritt ein Ereignis auf, das mit der Trigger-Nachricht einer Regel unifizierbar ist, führt der Executor diese Regel aus. Diese beiden Funktionen lassen sich über die Klasse `Settings` aktivieren oder deaktivieren (siehe Abschnitt 7.5).

7.4. Parallel-Szenarien

Parallellaufende Pfade in SML Szenarien sind ein kompliziertes Konzept. Bei der Implementierung eines `Pendants` in SBP gab es daher ein paar Dinge zu beachten. Die SML-Semantik besagt, dass die Nachrichten in zwei parallelen Pfaden innerhalb eines Pfades in gegebener Reihenfolge, aber pfadübergreifend in beliebiger Reihenfolge auftreten dürfen. Dies kann bei vielen langen Pfaden zu einer kombinatorischen Explosion führen, die per Hand nicht mehr zu implementieren ist. Wir benötigen also eine Methode zur simplen Programmierung paralleler Pfade, die die kombinatorischen Aspekte automatisiert im Hintergrund verarbeitet. Dafür stelle ich das `Parallel-Szenario` vor. Das Szenario, das sich aufspaltet, nennen wir jetzt das *Host-Szenario* oder auch den *Host*. Betrachten wir ein `Host-Szenario`, das sich in zwei Pfade aufspaltet. Der `Host` erzeugt dafür zwei zusätzliche `Parallel-Szenarien`. Diese Szenarien erben das Alphabet und die Rollenbindungen des `Hosts` und besitzen ansonsten lediglich einen `Body`. In diesem `Body` kann ein Pfad ganz regulär implementiert werden. Während der Ausführung durchlaufen die `Parallel-Szenarien` ganz normal ihren `Lifecycle`. Sie können dabei Nachrichten lediglich anfordern oder erwarten, aber nicht blockieren. Zudem synchronisieren sie sich bei jedem Nachrichtenaufruf mit dem `Host` und teilen ihm dabei mit, welche Nachrichten sie fordern bzw. erwarten. Der `Host` sammelt diese Nachrichten und fordert sie selbst noch einmal, als seien es seine eigenen. Der `Host` blockiert zudem die Nachrichten seines Alphabets abzüglich der Nachrichten der `Parallel-Szenarien`. Dadurch entsteht eine Nebenläufigkeit, die wir simpel programmieren können, ohne uns um die kombinatorischen Aspekte kümmern zu müssen. Abbildung 7.2 visualisiert diesen Prozess noch einmal. Das `Host-Szenario` erzeugt zwei neue `Parallel-Szenarien`. Das Alphabet aller Szenarien ist identisch und umfasst drei Nachrichten (a , b , c). Die `Parallel-Szenarien` erfordern die Nachrichten a bzw. b . Ob die Nachrichten hier `requested` sind oder nicht, ist erst mal irrelevant. Das `Host-Szenario` erhält die Nachrichten aller Pfade. Normalerweise würden die `Pfad-Szenarien` nun die jeweils andere Nachricht blockieren, da sie sich in ihrem Alphabet befinden. Hier haben wir die `Parallel-Szenarien` allerdings so definiert, dass sie keine

7. Implementierung

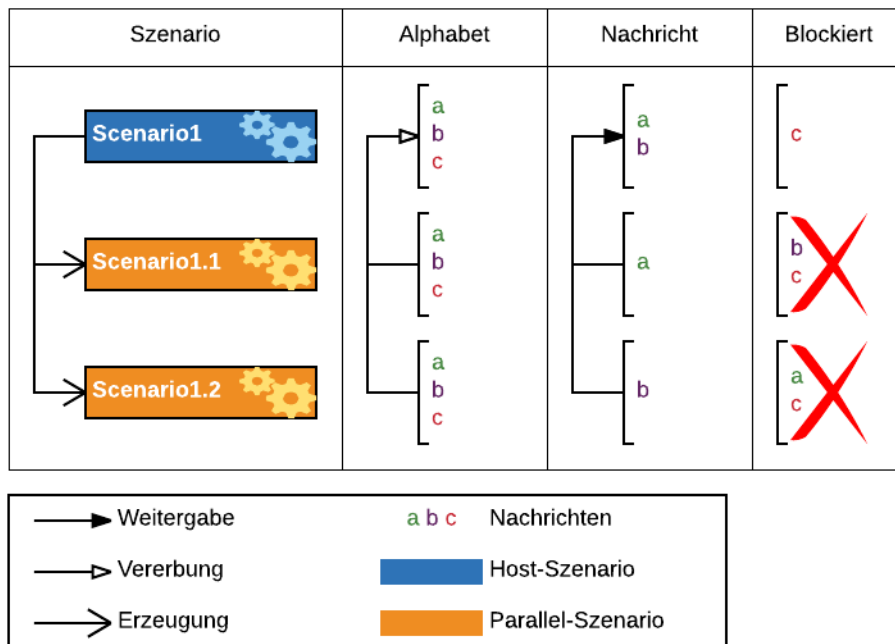


Abbildung 7.2.: Die Nachrichtenverwaltung von Host- und Parallel-Szenarien. Die Spalte Nachricht fasst requested- und waitFor-Nachrichten zusammen.

Nachrichten blockieren. Nur das Host-Szenario blockiert Nachrichten.

7.5. Einstellungen in SBP

SBP bietet viele Einstellungsmöglichkeiten für verschiedene Funktionalitäten. Dies dient dem Zweck der Anpassbarkeit an verschiedene Plattformen und Anwendungen. Die Klasse `Settings` hält eine Reihe von Werten, die bei der Ausführung gelesen und verwendet werden. Diese Werte können verändert werden, um bestimmte Funktionen an- bzw. auszuschalten oder das Verhalten der Anwendung zu modifizieren. Es werden die folgenden Einstellungen geboten:

7.5.1. Einstellungen für das Logging

SBP bietet einige Methoden zum Loggen der Ausführung, die an- oder ausgeschaltet werden können. Das Logging erstreckt sich über mehrere Klassen und

kann beliebig ausführlich gemacht werden. Der Codeblock 58 zeigt die globalen Einstellungen zum Logging in SBP und ihre Standardwerte. Zum Anpassen der Einstellungen kann der Wert der statische Felder geändert werden.

```

1 // Zeige auftretende Nachrichten in der Logging-Konsole.
2 public static boolean SPECTATOR__PRINT_CAUGHT_MESSAGE = false;
3 // Zeige in jedem Schritt die aktiven Specification-Szenarios in der Logging-Konsole.
4 public static boolean SPECTATOR__PRINT_ACTIVE_SPECIFICATION_SCENARIOS = false;
5 // Zeige in jedem Schritt die aktiven Assumption-Szenarios in der Logging-Konsole.
6 public static boolean SPECTATOR__PRINT_ACTIVE_ASSUMPTION_SCENARIOS = false;
7 // Zeige in jedem Schritt die aktiven System-Szenarios in der Logging-Konsole.
8 public static boolean SPECTATOR__PRINT_ACTIVE_SYSTEM_SCENARIOS = false;
9
10 // Zeige das Ende eines Supersteps in der Logging-Konsole.
11 public static boolean IDLE__PRINT_SUPERSTEP_IN_CONSOLE = true;
12
13 // Zeige an, wenn eine Methode durch den Executor ausgeführt wurde.
14 public static boolean EXECUTOR__PRINT_ON_METHOD_EXECUTION = false;
15 // Zeige an, wenn eine Transformationsregel durch den Executor ausgeführt wurde.
16 public static boolean EXECUTOR__PRINT_ON_RULE_EXECUTION = false;
17
18 // Zeige an, wenn ein System-Szenario voranschreitet.
19 public static boolean SCENARIO__PRINT_SYSTEM_PROGRESS = false;
20 // Zeige an, wenn ein Specification-Szenario voranschreitet.
21 public static boolean SCENARIO__PRINT_SPECIFICATION_PROGRESS = false;
22 // Zeige an, wenn ein Assumption-Szenario voranschreitet.
23 public static boolean SCENARIO__PRINT_ASSUMPTION_PROGRESS = false;

```

Code 58: Statische Felder zur Einstellung des Loggings.

7.5.2. Einstellungen für den Executor

Für das Executor-Szenario stehen drei Einstellungsmöglichkeiten bereit. Der Executor kann Methoden auf Objekten des Objektsystems ausführen. Passt ein Ereignis zu einer Methode oder einem Feld mit Setter auf dem Empfängerobjekt, so kann der Executor diese Methode bzw. diesen Setter mit den Parametern des Ereignisses ausführen. Zudem kann der Executor Transformationsregeln der Spezifikation ausführen, wenn ein Ereignis zur Trigger-Nachricht einer Regel passt. Diese drei Funktionen können mit den in Codeblock 59 gezeigten statischen Feldern an- bzw. ausgeschaltet werden.

```

1 // Führe zu Ereignissen passende Methoden auf dem Objektsystem aus.
2 public static boolean EXECUTOR__EXECUTE_METHODS = false;
3 // Führe zu Ereignissen passende Setter-Methoden auf dem Objektsystem aus.
4 public static boolean EXECUTOR__EXECUTE_SETTERS = true;
5 // Führe zu Ereignissen passende Transformationsregeln aus.
6 public static boolean EXECUTOR__EXECUTE_RULES = true;

```

Code 59: Statische Felder zur Einstellung des Executor-Szenarios.

7.5.3. Einstellungen für Output-Streams

Für das Logging stehen verschiedene Streams zur Verfügung. Dadurch kann das Logging thematisch aufgeteilt werden. Dabei können folgende Logging-

7. Implementierung

Ereignisse in verschiedene Streams aufgespalten werden: Die Beobachtungen des Spectators, die Ausführung des Executors, das Ende eines Supersteps, das Voranschreiten der Szenarien und die Netzwerkkommunikation. Standardmäßig werden diese Streams auf `System.out` gelenkt, können allerdings über die in Codeblock 60 angegebenen statischen Felder umgelenkt werden. Zur einfacheren Änderung dieser Streams bietet die `Settings`-Klasse passende Methoden (siehe Abschnitt 7.5.4).

```
1 // Der Output-Stream für das Logging des Spectator-Szenarios.
2 public static PrintStream SPECTATOR_OUT = System.out;
3 // Der Output-Stream für das Logging des Executor-Szenarios.
4 public static PrintStream EXECUTOR_OUT = System.out;
5 // Der Output-Stream für das Logging eines Supersteps.
6 public static PrintStream IDLE_OUT = System.out;
7 // Der Output-Stream für das Logging der Szenarios der Spezifikation.
8 public static PrintStream SCENARIO_OUT = System.out;
9 // Der Output-Stream für das Logging der Netzwerkkommunikation.
10 public static PrintStream NETWORK_OUT = System.out;
```

Code 60: Statische Felder zur Einstellung der Output-Streams.

7.5.4. Methoden für die Output-Streams

Die Klasse `Settings` bietet zur einfacheren Handhabung der Output-Streams zwei Methoden zum Setzen der Streams. Dabei unterschieden die Methoden zwischen Streams der Runtime und der Netzwerkkommunikation. Der Codeblock 61 zeigt die statischen Methoden zum Setzen dieser Streams. Dies vereinfacht das Einstellen der Streams und kann zum Beispiel für das Umleiten des Loggings auf eine GUI benutzt werden. Die GUI in Abbildung 6.3 in Abschnitt 6.3 verwendet diese Methoden zur Aufspaltung des Loggings auf Runtime und Netzwerk.

```
1 public static void setRuntimeOut(PrintStream printStream) {
2     SPECTATOR_OUT = printStream;
3     EXECUTOR_OUT = printStream;
4     IDLE_OUT = printStream;
5     SCENARIO_OUT = printStream;
6 }
7
8 public static void setServerOut(PrintStream serverStream) {
9     NETWORK_OUT = serverStream;
10 }
```

Code 61: Statische Methoden zur Einstellung der Output-Streams.

7.6. Aktivierung von Szenarien

Beim Start der Ausführung einer Spezifikation sind erst einmal nur die System-Szenarien aktiv. Um die Szenarien der Spezifikation zu aktivieren, müssen Ereignisse auftreten, die mit ihren initialisierenden Nachrichten unifizierbar sind.

Dies muss allerdings bei jedem auftretenden Ereignis geprüft werden. Dafür ist das Initializer-Szenario zuständig. Die Klasse `InitializerScenario` ist ein System-Szenario, das beim Start der Ausführung aktiviert ist. Es bewacht die Ausführung, indem es den auftretenden Ereignissen zuhört und diese mit den Listen der initialisierenden Nachrichten der Szenarien vergleicht. Passt eines der Ereignisse zu einer initialisierenden Nachricht, erstellt das Initializer-Szenario eine aktive Kopie des Szenarios aus der Spezifikation. Dieses Szenario wird der Ausführung hinzugefügt und läuft parallel zu den System-Szenarien. Ein Szenario kann daraufhin beliebig oft wieder aktiviert werden und kann auch mehrfach aktiv sein, da das Initializer-Szenario immer eine Kopie des inaktiven Szenarios erstellt. Auf diese Weise ist sichergestellt, dass ein Szenario nicht nach einmaliger Ausführung verschwinden kann. In einer ersten Entwicklungsstufe von SBP waren Szenarien aus der Spezifikation von Anfang an an der Ausführung beteiligt und kopierten sich selbst bei Aktivierung. Dies hat durchaus funktioniert, jedoch machte es das Debugging unnötig kompliziert, da nur schwer zu sehen war, welche Szenarien tatsächlich aktiviert waren. Außerdem waren so Threads für Szenarien aktiv, die jedoch nichts taten, außer zuzuhören. Durch das Initializer-Szenario haben wir diese Problematik gelöst, da nun lediglich ein Szenario durchgängig zuhört und die Szenarien erst dann aktiviert, wenn sie benötigt werden.

7.7. Lifecycle von Szenarien

Nach der Aktivierung eines Szenarios durch das Initializer-Szenario beginnt dessen *Lifecycle*. Der Lifecycle ist die Ausführung eines Szenarios während dessen Lebenszeit. Abbildung 7.3 zeigt die Aktivierung und die weitere Ausführung eines Szenarios. Nachdem ein Szenario kopiert und gestartet wurde, startet die Ausführung des Szenarios mit dem Binden der Rollen der initialisierenden Nachricht. Danach werden die zusätzlichen Rollen gebunden (`registerRoleBindings`). Danach wird das Alphabet mit blockierten, unterbrechenden und verbotenen Nachrichten registriert. Ab diesem Zeitpunkt gilt das Szenario als aktiv und beginnt mit der Ausführung der Methode `body`. Ist der Body durchlaufen, oder durch eine Verletzung unterbrochen, gilt das Szenario wieder als inaktiv. Zuletzt wird die Methode `reset` ausgeführt. Diese Methode tut in der Basisimplementierung nichts, kann allerdings überschrieben werden, um eine Reaktion auf das Beenden des Szenarios zu erwirken. Das inaktive Szenario wird dann aus der Runtime entfernt und kann zerstört werden.

7. Implementierung

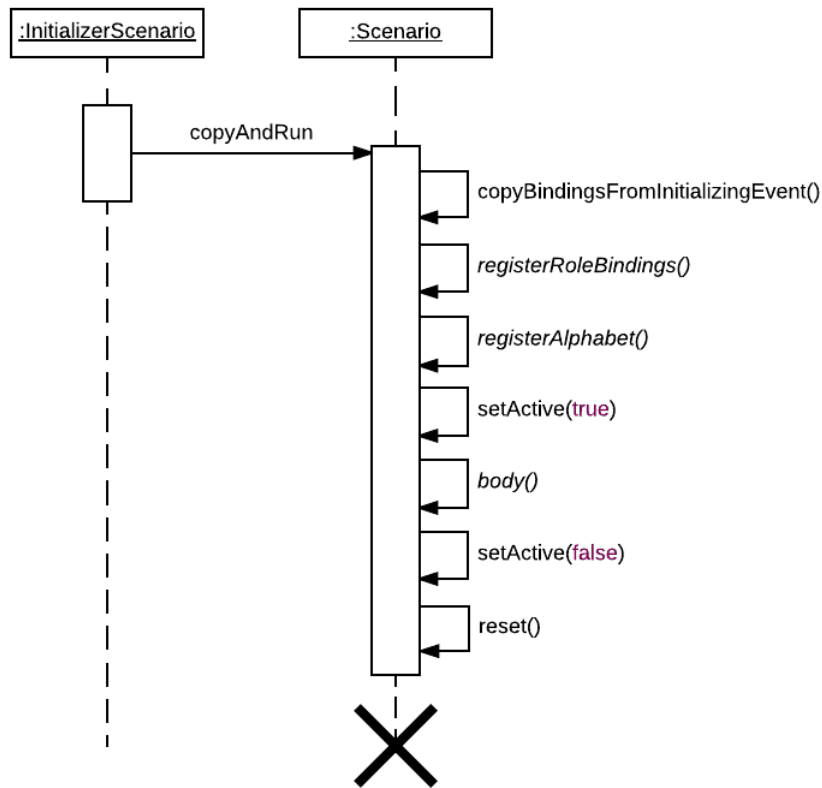


Abbildung 7.3.: Der Lifecycle eines Szenarios nach der Aktivierung durch das Initializer-Szenario.

7.7.1. Interpretation von Nachrichten in verteilten Komponenten

Wenn wir über ein verteiltes reaktives System sprechen, müssen wir dieses System immer aus verschiedenen Blickwinkeln betrachten. So betrachten wir die verteilten Komponenten des Systems jede für sich als einzelnes Teilsystem. Jede dieser Komponenten erkennt die anderen Komponenten zwar als Teil des Systems an, kann sie jedoch nicht kontrollieren. Darum müssen sie für die Ausführung von Nachrichten als Teil der Umwelt betrachtet werden. Während der Entwicklung betrachten wir dies allerdings anders. Wir modellieren das System als Ganzes. Die Unterscheidung, welche Komponente etwas steuert und wie demnach modellierte Nachrichten zu interpretieren sind, muss zur Laufzeit entschieden werden. Betrachten wir ein SML-Szenario mit den Nachrichten aus

Codebeispiel 62. Übersetzen wir diese Nachrichten nach SBP, so ergibt sich der Code in Beispiel 63. Zuerst tritt eine Umweltnachricht auf, gefolgt von zwei Systemnachrichten unterschiedlicher Rollen und abschließend wird wieder eine Umweltnachricht erwartet. Abbildung 7.4 zeigt verschiedene Interpretationen dieser Nachrichten in der SBP Runtime. Links wird die lokale Ausführung betrachtet, in der alle Komponenten von einer Maschine gespielt werden. Dies ist der Fall der lokalen Simulation. Auf der rechten Seite ist die verteilte Ausführung auf drei Komponenten zu sehen. Die Komponenten steuern jeweils ein Objekt und somit können sie genau eine Rolle übernehmen. Wir sehen, dass die Komponente ENV keine der Systemnachrichten ausführen kann und daher an dieser Stelle *requested* als *wait for* interpretiert. Die Komponenten A und B hingegen können jeweils eine der Systemnachrichten ausführen und interpretieren die andere ebenfalls als Umwelt. Da wir in der `SpecificationRunconfig` jeder Komponente zuordnen können, welche Objekte sie kontrollieren kann, kann sie während der Laufzeit entschieden, welche Nachricht der Spezifikation für sie selbst System oder Umwelt ist. Der Message-Selection Algorithmus von SBP prüft bei jeder Nachricht, die als *requested* deklariert ist, ob der Sender kontrollierbar ist oder nicht. Daran entscheidet sich, ob ein *requested* als *waitFor* betrachtet wird. Das ermöglicht es uns, auch in SBP das System als Ganzes zu betrachten und später trotzdem verteilt auszuführen.

```

1 message env -> a.request
2 message requested a -> b.trigger
3 message requested b -> env.response
4 message env -> a.finish

```

Code 62: Beispielnachrichten eines SML-Szenarios mit den Rollen *env*, *a* und *b*.

```

1 waitForEvent(env, a, "request");
2 requestEvent(a, b, "trigger");
3 requestEvent(b, env, "response");
4 waitForEvent(env, a, "finish");

```

Code 63: Übersetzung der Nachrichten aus Codebeispiel 62 nach SBP.

7.8. Event-Selection in SBP

Der Event-Selection Algorithmus von SBP basiert auf dem von BP, ist jedoch dank der SML-Semantik deutlich komplexer. In der Basis erwartet der Algorithmus drei Listen von Nachrichten:

- **Requested Events (R)**: Die Events, die zur Ausführung angefordert werden. Ein Event kann nur von der Event-Selection ausgewählt werden, wenn

7. Implementierung

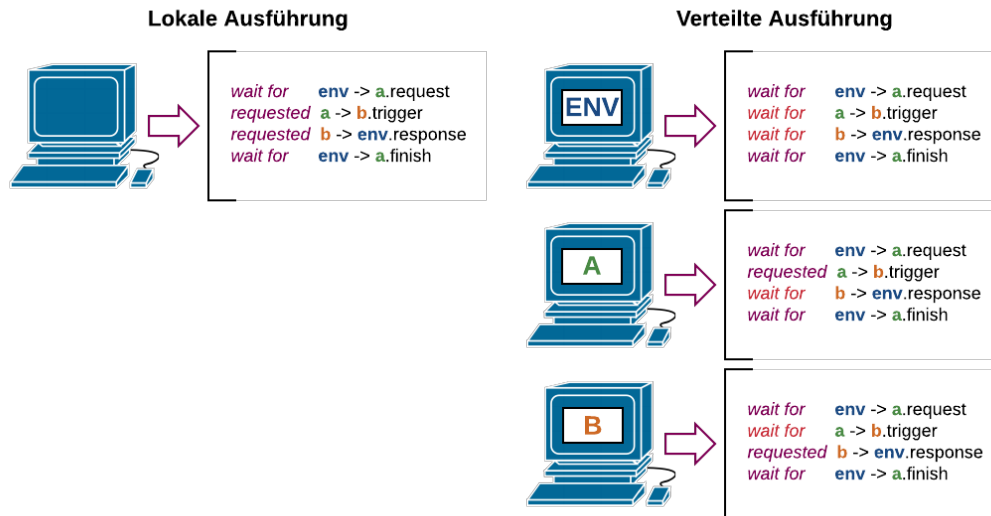


Abbildung 7.4.: Interpretation der SML Nachrichten in Codebeispiel 62 in der lokalen Ausführung auf einer Maschine (links) und der verteilten Ausführung auf drei Komponenten (rechts).

sie in dieser Liste enthalten ist.

- **Waited for Events (W):** Die Liste von Events, die erwartet werden. Wird eine Nachricht erwartet, treibt sie die Ausführung voran. Diese Nachricht kann jedoch nicht ausgewählt werden, wenn sie nicht ebenfalls in der R-Liste enthalten ist.
- **Blocked Events (B):** Blockierte Events können nicht ausgewählt werden. Ist ein Event in dieser Liste, ist es blockiert und kann nicht ausgeführt werden, auch wenn es in der R-Liste steht.

Die SML-Semantik kommt allerdings mit diesen drei Listen nicht aus. Szenarien in SML und damit auch in SBP haben weitere Listen, die für BP in bestimmter Weise zusammengeführt werden müssen. Abbildung 7.5 zeigt grafisch, wie das Sortieren der Nachrichten für BP funktioniert. Speziell müssen wir an zwei Stellen je nach Situation entscheiden, wie eine Nachricht behandelt wird. Es ist zu unterscheiden, ob der Zustand des Szenarios strict ist oder nicht. Dadurch ändert sich die Sortierung der blockierten Nachrichten. Außerdem ist zu beachten, ob der Sender einer Nachricht kontrollierbar ist, wonach

sich die Sortierung der geforderten Nachrichten richtet (siehe Abschnitt 7.7.1). Im Folgenden werden die Listen der Szenarien und die Sortierung beschrieben:

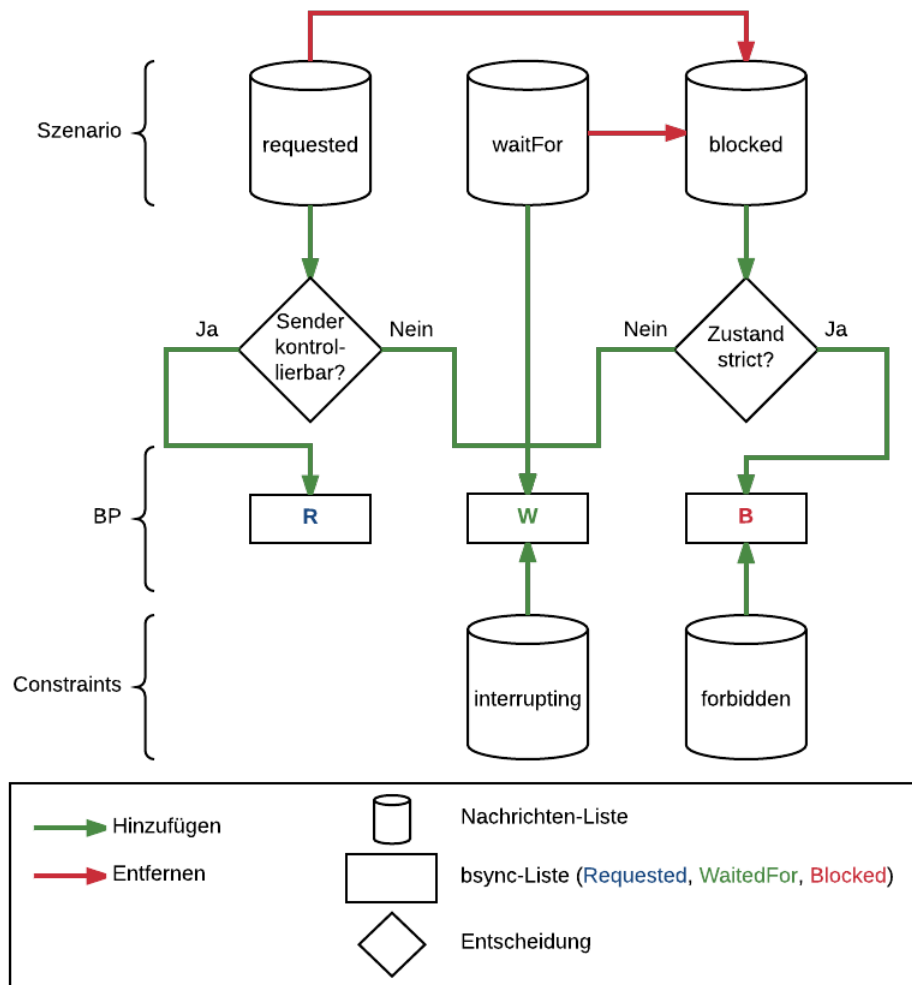


Abbildung 7.5.: Die Nachrichten-Listen von Szenarien werden in die drei Listen von BP einsortiert.

- **Requested:** Dies sind die Nachrichten, die vom Szenario angefordert werden. An dieser Stelle ist für jede Nachricht einzeln zu beachten, ob der Sender der Nachricht kontrollierbar ist. Wenn er das ist, wird die Nachricht der R-Liste von BP hinzugefügt. Andernfalls landet sie in der W-Liste.

7. Implementierung

- **WaitFor:** Die Nachrichten in dieser Liste werden erwartet. Sie gleicht der **W**-Liste in BP und wird einfach übernommen.
- **Blocked:** Die Liste der blockierten Nachrichten basiert auf den als blockierend eingetragenen Nachrichten im Alphabet des Szenarios. Zur Laufzeit allerdings können Nachrichten gefordert oder erwartet werden. Ist dies der Fall, werden diese Nachrichten von der Blocked-Liste abgezogen. Die übrig bleibenden Nachrichten sind die blockierten Nachrichten. Wie diese Nachrichten einsortiert werden müssen, bestimmt der Zustand des Szenarios. Ist dieser strict, bedeutet das Auftreten einer dieser Nachrichten eine Safety-Violation. Dies darf nicht passieren, weshalb die Nachrichten in die **B**-Liste von BP eingeordnet werden. Ist der Zustand nicht strict, lösen diese Nachrichten lediglich eine Interrupt-Violation aus. Dies darf sehr wohl passieren, weshalb die Nachrichten in die **W**-Liste von BP einsortiert werden.
- **Interrupting:** Im Alphabet des Szenarios können Nachrichten als *interrupting* deklariert werden. Dies bedeutet, dass die auftreten können, jedoch eine Verletzung verursachen. Demnach werden sie in die **W**-Liste von BP einsortiert.
- **Forbidden:** Verbotene Nachrichten können ebenfalls im Alphabet des Szenarios deklariert werden. Diese Nachrichten sorgen beim Auftreten für eine Safety-Violation, also eine Verletzung, die nicht passieren darf. Darum dürfen diese Nachrichten nicht auftreten und zählen zu der **B**-Liste von BP.

Ist diese Sortierung erledigt, läuft der normale Event-Selection Algorithmus von BP mit den drei Listen ab. Das ausgewählte Ereignis wird ausgeführt und an die Szenarien weitergegeben, die eine damit unifizierbare Nachricht gefordert haben. Diese Szenarien wachen wieder auf und prüfen, aus welcher Liste die Nachricht stammte. Stammt sie beispielsweise aus der Interrupt-Liste, wird das Szenario mit einer Interrupt-Violation unterbrochen.

7.9. Prozessprioritäten von Szenarien

Eine Besonderheit der Event-Selection von BP ist die Prozesspriorität. Jeder B-Thread hat eine bestimmte Priorität nach der die Nachrichten ausgeführt werden. Diese Prioritäten sind als Double-Werte implementiert und bestimmen, welche Nachricht von der Event-Selection ausgewählt wird. Fordern nun

7.10. Verteilte Ausführung über das Netzwerk

zwei B-Threads A und B jeweils eine Nachricht a bzw. b an, wird die Nachricht bevorzugt ausgewählt, die vom B-Thread mit höherer Priorität angefordert wird. Eine kleine Zahl entspricht dabei einer hohen Priorität und eine große Zahl einer niedrigen. Hat B-Thread A also die Priorität 1 und B-Thread B die Priorität 5, wird Nachricht a vor Nachricht b ausgewählt. SBP Szenarien erweitern diese B-Threads und müssen sich daher diesen Prioritäten anpassen. Da es in Szenarien an sich keine Prioritäten gibt, können wir sie bis auf ein paar Spezialfälle ignorieren. Abbildung 7.6 zeigt die Aufteilung der verschiedenen Arten von Szenarien auf die Prioritäten. Die ersten Prioritäten gehen an die Specification-Szenarien, da sie Nachrichten des Systems anfordern sollen. Danach folgen die Assumption-Szenarien. Sie bedingen die Umwelt und können zur Fehlerfindung beitragen, können aber selbst keine Ereignisse auslösen. Zuletzt kommen die System-Szenarien. Hier gibt es nur zwei Szenarien, die Nachrichten fordern. Das Idle-Szenario und die Event-Queue, die erst aktiviert wird, wenn das Idle-Szenario ein Ereignis auslöst. Das Idle-Szenario fordert jederzeit die spezielle Nachricht *IDLE*, wird allerdings durch die Nachrichten der Specification-Szenarien blockiert. Erst wenn die nichts mehr fordern, kann die Nachricht *IDLE* ausgewählt werden und das System wartet auf die Umwelt. Ganz wichtig ist, dass die Event-Queue immer eine höhere Priorität hat, als das Idle-Szenario. Die Event-Queue kann sowieso erst etwas fordern, wenn *IDLE* aufgetreten ist. Darum kann das Idle-Szenario ungehindert *IDLE* senden, sobald keine Specification-Szenarien mehr etwas fordern. Nachdem *IDLE* aufgetreten ist, wartet die Event-Queue auf die Umwelt und fordert diese Nachricht. Das Idle-Szenario ist dann wieder blockiert, da die Event-Queue eine höhere Priorität hat.

7.10. Verteilte Ausführung über das Netzwerk

Die SBP Bibliothek bietet eine verteilte Ausführung über das lokale Netzwerk. Über eine Client-Server-Architektur verbinden sich die verteilten Komponenten des Systems. Diese Komponenten spielen jeweils bestimmte Rollen und können durch Szenarien Ereignisse ausführen. Diese Ereignisse werden in Form von Nachrichten an die anderen Komponenten im Netzwerk weitergegeben. Dafür werden Nachrichten in das Netzwerk gesendet und wieder empfangen. Technisch funktioniert das über Java Sockets, über die sich die Komponenten verbinden.

7. Implementierung

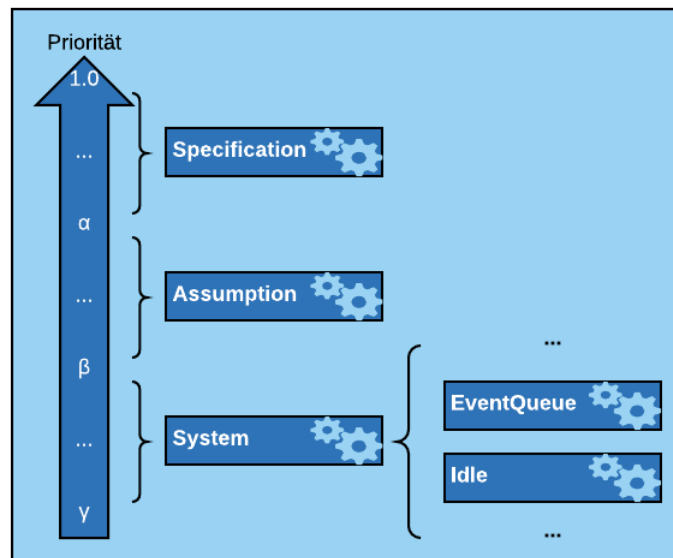


Abbildung 7.6.: Prioritäten der Szenarien. Die Prioritäten 1 bis α gehen an die Specification-Szenarien, α bis β an die Assumption-Szenarien und β bis γ an die System-Szenarien.

7.10.1. Architektur der verteilten Ausführung

Die verteilte Ausführung in SBP ist durch den `DistributedRuntimeAdapter` implementiert. Abbildung 7.7 zeigt die Client-Server-Architektur dieses Adapters. Jede verteilte Komponente hostet einen Server für die Annahme der Nachrichten und einen Client für jede andere Komponente zum Senden von Nachrichten. Gesendet und empfangen wird über Java Sockets. Jede Komponente muss die Netzwerkadressen der anderen Komponenten kennen, um Nachrichten versenden zu können. Die Netzwerkadressen werden in der `SpecificationRunconfig` in die `ObjectRegistry` eingetragen. Die Clients und Server verwenden das dadurch gegebene Mapping von Objekt nach IP und umgekehrt.

7.10.2. Senden einer Nachricht

Sobald die lokale Event-Selection ein Ereignis ausführt, wird es als Nachricht in das Netzwerk gesendet. Wird ein Ereignis ausgeführt, wird dies vom Publisher-Szenario erkannt. Der Publisher gibt dieses Ereignis dann an die `publish`-Methode des registrierten Runtime-Adapters der `SpecificationRunconfig` weiter. Die Imple-

7.10. Verteilte Ausführung über das Netzwerk

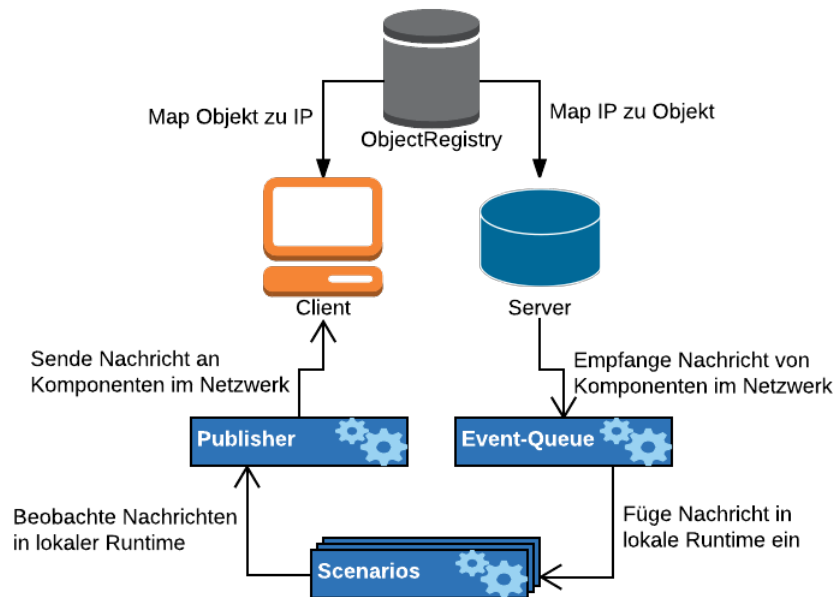


Abbildung 7.7.: Client-Server-Architektur der verteilten Ausführung in SBP.

mentierung des `DistributedRuntimeAdapters` hostet einen Server und verbindet einen Client zu den anderen Komponenten im Netzwerk. Dieser Client kann durch die Methode `publish` dazu bewegt werden, eine Nachricht in das Netzwerk zu senden. Dabei wird die Nachricht an die Server jeder anderen Komponente per Broadcast verschickt. Besonders zu beachten war hier das Plätten der Nachricht. Eine Nachricht kann verschiedene Parameter haben. Dies können auch Rollen bzw. Objekte sein, die an Rollen gebunden sind. Diese Objekte können auf der Empfängerseite nicht zugewiesen werden. Darum findet hier eine Umwandlung der Parameter statt. Anstatt das Objekt über das Netzwerk zu senden, wird die ID des Bindings aus der `ObjectRegistry` gesendet.

7.10.3. Empfangen einer Nachricht

Jede Komponente der verteilten Ausführung hört ständig auf Nachrichten aus dem Netzwerk. Sendet eine Komponente im Netzwerk nun per Broadcast, erhalten die empfangenen Komponenten diese Nachricht durch ihre Server. Der Server erhält die Nachricht und gibt sie an die Methode `receive` des aktuellen Runtime-Adapters weiter. Der Adapter reicht diese Nachricht an das Event-Queue-Szenario weiter. Die Nachricht wird in die Queue angehängt und dadurch

7. Implementierung

in die Laufzeitanwendung gespeist. Auch hier ist es wichtig, dass eine geplättete Nachricht empfangen wird. Das bedeutet, dass nicht-primitive Parameter als IDs übertragen werden. Diese IDs werden durch die `ObjectRegistry` wieder zu Objekten gemappt.

8. Evaluation und fortführende Ansätze

Für das Ziel der Arbeit haben wir in Kapitel 3 verschiedene Anforderungen an die entwickelte Methodik und an die Implementierung gesetzt und sie in vier Themen unterteilt. Entscheidungen bei der Entwicklung, die zur Erfüllung dieser Anforderungen nötig waren, wurden bereits in den vorherigen Kapiteln ab und zu genannt. In diesem Kapitel wollen wir noch einmal evaluieren, inwieweit wir diese Anforderungen erfüllt haben.

8.1. Java Bibliothek

Es sollte eine Java Bibliothek entwickelt werden, die BP als Basis benutzt und die Konzepte und die Semantik der Scenario Modeling Language implementiert. Die Bibliothek soll das szenariobasierte Entwickeln nach dem Vorbild von SML in Java möglich machen. Es soll eine Möglichkeit geben, plattformspezifischen Code einzupflegen und Sensorik und Aktorik anzusteuern. Abbildung 8.1 zeigt diese Anforderungen noch einmal grafisch.

In Kapitel 4 habe ich die Bibliothek und ihre Elemente vorgestellt. Sie implementiert die Konzepte der Spezifikation, Szenarien, Nachrichten und Rollen. Dazu kommt eine Konfiguration wie beim Vorbild SCENARIOTOOLS. Die Szenarien basieren auf den B-Threads von BP und verwenden die Synchronisation der B-Application. In Kapitel 5 haben wir ein Mapping vorgestellt mit dessen Hilfe eine SML Spezifikation zu SBP Code umgewandelt werden kann. Zudem habe ich einen Code-Generator implementiert, der für bestehende Spezifikationen per Knopfdruck Code generiert.

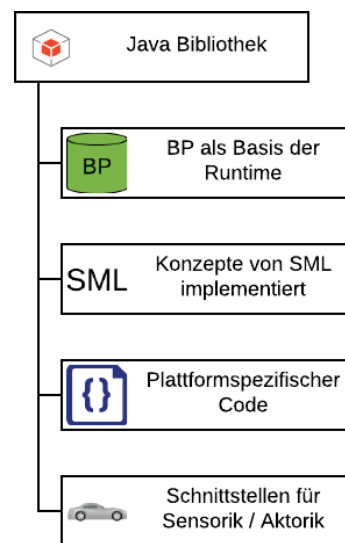


Abbildung 8.1.: Anforderungen an die Java Bibliothek.

Wir haben Konzepte für das

8. Evaluation und fortführende Ansätze

Einbinden von Fremdcode in die SBP Spezifikation vorgestellt. Dazu zählen die Transformationsregeln und die Szenario-Observer. Die Szenario-Observer können zudem als Schnittstelle zur Aktorik des Zielsystems genutzt werden. Für die Sensorik wurden in Kapitel 6 Methoden zur Einspeisung von Ereignissen in die Runtime vorgestellt. Wir sehen, dass die Anforderungen an die Implementierung erfüllt sind.

8.2. Ausführung

Auch an die Ausführung von SBP haben wir mehrere Anforderungen gestellt. Eine SBP Spezifikation soll sowohl lokal simuliert, als auch verteilt ausgeführt werden können. Für letzteres ist eine Methode zur Kommunikation zwischen den verteilten Komponenten des Systems notwendig. Es sollte außerdem eine Möglichkeit zur Steuerung der Anwendung durch einen Benutzer geben.

In dem Kapitel 4 habe ich einen Runtime-Adapter für die lokale Simulation und einen für die verteilte Ausführung über ein Netzwerk vorgestellt. Bei der lokalen Simulation werden alle Komponenten von einer Maschine gespielt. Für die verteilte Ausführung können mehrere Maschinen miteinander verbunden werden und tauschen Nachrichten über eine Client-Server-Architektur aus, wie in Kapitel 7 dargestellt. Es können auch weitere Kommunikationsmechanismen implementiert und verwendet werden (siehe Abschnitt 6.10). In Abschnitt 6.2 erkläre ich, wie die Steuerung der Umwelt durch den Benutzer realisiert wird. Damit sind die Anforderungen, die wir an die Ausführung gesetzt haben, erfüllt.

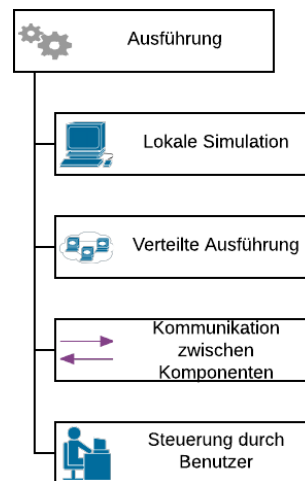


Abbildung 8.2.: Anforderungen an die Ausführung.

8.3. Entwicklungsmethodik

An die Entwicklungsmethodik für SBP haben wir drei Anforderungen gesetzt. Es soll ein Mapping geben, durch das wir eine Code-Generierung von SML nach SBP ermöglichen können. Die entwickelte Software soll testbar sein und es soll eine vernünftige Möglichkeit zum Debugging geben.

Kapitel 5 zeigt ein Mapping von SML nach SBP, welches für eine Code-Generierung geeignet ist. Ich habe zudem einen Code-Generator als Eclipse-Plugin implementiert, der per Knopfdruck eine bestehende SML Spezifikation zu SBP umwandelt. Auch für das Testen der Software habe ich eine Methode vorgestellt. In Abschnitt 6.9 stelle ich Test-Szenarien vor, die mittels JUnit in das SBP Programm integriert werden können. In JUnit kann dann getestet werden, ob das Test-Szenario erfolgreich beendet wurde. Außerdem kann das Szenario selbst erkennen, wenn eine Verletzung auftritt. Das Debugging für SBP stellt sich relativ einfach dar, da wir einfach einen Debugger für Java verwenden können. In Abschnitt 6.5 stelle ich das Debugging mit der Eclipse Debug-Perspective vor. Damit sind unsere Anforderungen an die methodischen Aspekte der Arbeit erfüllt.

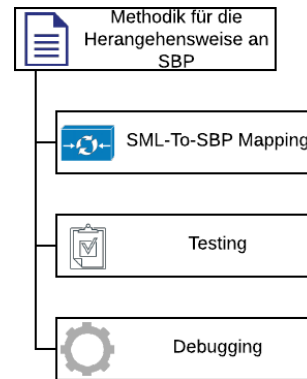


Abbildung 8.3.: Anforderungen an die Entwicklungsmethodik.

8.4. Qualitätsaspekte

Wir haben an die szenariobasierte Programmierung in Java eine Reihe von Anforderungen, die Qualität betreffend, gestellt. Dazu zählt die Lesbarkeit des Programmcodes, die Erweiterbarkeit der SBP Bibliothek, die Performance der Ausführung, die Skalierbarkeit für große Systeme und die Plattformunabhängigkeit bezüglich der Entwicklungswerkzeuge. Die Lesbarkeit betrachten wir hier im Vergleich zur Modellierungssprache SML. Die SBP Bibliothek orientiert sich stark an der Syntax dieser Sprache, um einen gewissen Wiedererkennungswert zu generieren. Durch die Ähnlichkeit von Schlüsselwörtern und Methodennamen können wir eine gewisse Lesbarkeit des Codes annehmen. Zudem habe ich in Abschnitt 5.2 eine Möglichkeit gezeigt, wie Code sinnvoll durch Supertyping versteckt und dadurch leserlicher gemacht werden kann. Die weiteren Qualitätsaspekte werde ich im Folgenden etwas genauer analysieren.



Abbildung 8.4.: Qualitätsanforderungen.

8.4.1. Erweiterbarkeit

Eine wichtige Anforderung an SBP war die Erweiterbarkeit. Wir wollen eine Entwicklungsmethodik, die für unterschiedliche Anwendungsfälle eine Lösung bietet. Dafür ist an mehreren Stellen gesorgt.

Ein wichtiger Aspekt ist die verteilte Ausführung und damit ein Kommunikationsweg zwischen verschiedenen Runtimes der verteilten Komponenten. Dies ist in der aktuellen Implementierung über Java-Sockets gelöst. Allerdings kann es Situationen geben, in der Java-Sockets nicht passend sind. Dafür kann über die Klasse `AbstractRuntimeAdapter` ein eigener Runtime-Adapter geschaffen werden, der ein anderes Protokoll implementiert. Es können hier beispielsweise Internetprotokolle, wie *MQTT*¹ verwendet werden. MQTT ist ein Protokoll für die Kommunikation zwischen Maschinen, welches über ein leichtgewichtiges Publish-Subscribe-Verfahren Nachrichten über das Internet sendet. Für sicherheitskritische Systeme kann auch eine Verschlüsselung oder ein Rechte-System in den Runtime-Adapter eingebaut werden. SBP eignet sich daher für beliebige Kommunikationsschnittstellen und -verfahren.

Auch besonders wichtig ist die Ausführung von plattformspezifischem Code. Nehmen wir an, wir wollen ein Fahrassistenzsystem entwickeln, das auf dem Dashboard des Fahrzeugs Nachrichten für den Fahrer bereitstellt. Die Spezifikation dieses Systems können wir mit SML oder SBP realisieren. Dann müssen wir allerdings das wirkliche Dashboard des Fahrzeugs daran anschließen. Bekommt das Objekt Dashboard durch die Szenarien die Nachricht *Go*, soll auf dem realen Gerät diese Nachricht stehen. Dafür muss spezieller Code für die GUI des Dashboards in die Spezifikation eingebaut werden. SBP bietet hier die Möglichkeit die Ausführung von Nachrichten zu überwachen. Über die Szenario-Observer können Nachrichten aufgefangen und weiter verarbeitet werden. So kann ein Observer implementiert werden, der eine GUI aktualisiert (siehe Abschnitt 6.1) bzw. die Hardware des Dashboards ansteuert. Auch hier bietet SBP die geforderte Erweiterbarkeit, um fremden Code hinzuzufügen.

Gehen wir von einem intelligenten Fahrzeugsystem aus, müssen wir auch mit Sensorwerten rechnen. Wenn wir die spezifizierte Software auf dem Zielsystem verwenden, müssen wir die Umwelt nicht länger simulieren. Stattdessen empfangen wir reale Ereignisse aus der Umwelt über die Sensorik des Fahrzeugs. Diese Sensorereignisse müssen wir in unsere szenariobasierte Anwendung einspeisen. Hier bietet SBP die Möglichkeit, diese Ereignisse in den Runtime-Adapter einzuspeisen (siehe Abschnitt 6.2).

Wir sehen, SBP bietet die gewünschte Erweiterbarkeit in den Bereichen Eingabe, Ausgabe und Kommunikation und erfüllt daher diese Anforderung.

¹<http://mqtt.org/>

8.4.2. Plattformunabhängigkeit

Ein wichtiges Ziel von SBP war es, nicht mehr von einer spezifischen Plattform abhängig zu sein. So kann ein Zielsystem für die Spezifikation beispielsweise auch ein Android-System sein. Für ein intelligentes Fahrzeugsystem kann es durchaus sein, dass ein Teil der Software auf einem Smartphone läuft. So kann das Smartphone eine GUI zur Visualisierung von Werten oder zur Benutzereingabe anzeigen. Für die Implementierung dieser Android-Applikation können wir unsere szenariobasierte Anwendung auch einfach auf Android laufen lassen, da die SBP Bibliothek auf Java basiert. Zur Veranschaulichung habe ich eine solche Android-Applikation implementiert, die das Dashboard eines Fahrzeugs aus dem Car-To-X Beispiel simuliert. Abbildung 8.5 zeigt den Screenshot dieser Applikation, die ein Logging-Feld für die Nachrichten und den Netzwerktransfer, sowie eine Anzeige für die Erlaubnis zu fahren zeigt. Für die GUI habe ich einen Szenario-Observer implementiert, der auf die Ereignisse *showGo* und *showStop* reagiert. Daraufhin wird auf der GUI der Text **Go** oder **Stop** angezeigt. Die Android-Applikation ist nun eine eigenständige Komponente des Systems mit einer eigenen Runtime. Die Runtime kommuniziert dann mit den Runtimes der anderen Komponenten. Die SBP Spezifikation konnte ich hier ohne weitere Modifikationen wie in der lokalen Simulation auf dem PC verwenden. Die einzige Besonderheit war es, dass die `SpecificationRunconfig` in einem zusätzlichen Thread gestartet werden musste, um nicht den UI-Thread zu blockieren. Die SBP Bibliothek kann also auf beliebigen Plattformen verwendet werden, solange dort Java verwendet werden kann. Für weitere Plattformen müsste eine entsprechende Bibliothek in einer anderen General Purpose Language implementiert werden. Die einfache Installation der Software auf einem Android-System genügt unserer Anforderung aber erst einmal. Dies ist bereits ein Fortschritt gegenüber den momentanen Möglichkeiten von SCENARIOTOOLS.

8.4.3. Performance

Vergleicht man ausführbaren Code und die Interpretation von Modellen, so stellt man häufig fest, dass ersteres schneller ist. Dies wollen wir auch mit der szenariobasierten Programmierung in Java erreichen. Um zu überprüfen, welche Performance SBP im Vergleich zu SCENARIOTOOLS zeigt, habe ich ein Experiment durchgeführt.

8. Evaluation und fortführende Ansätze

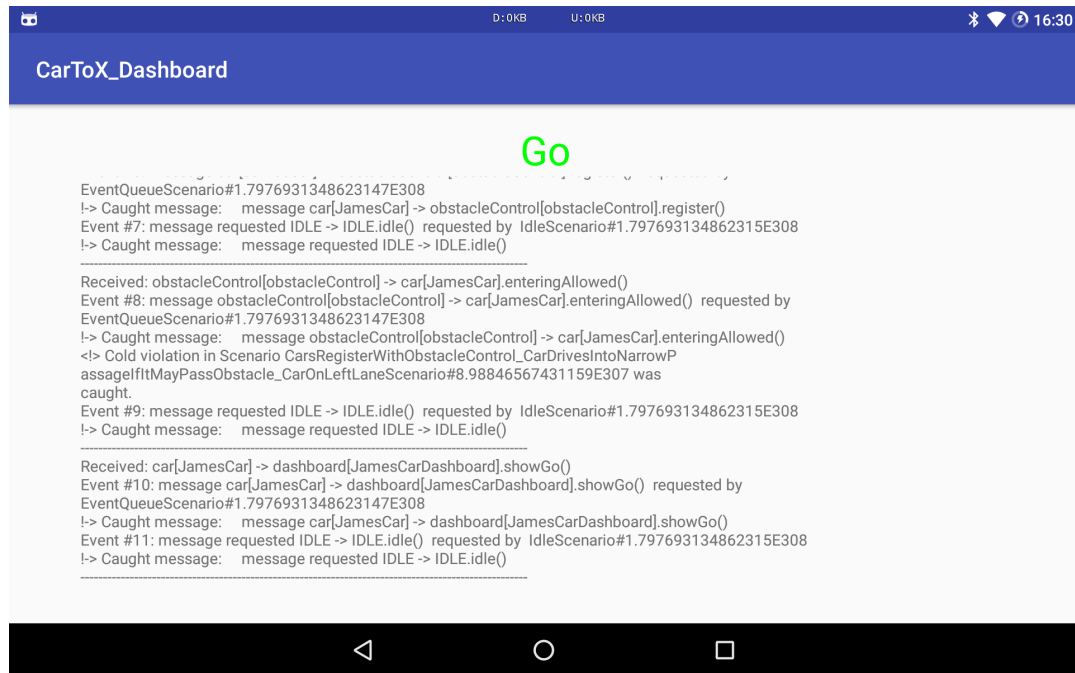


Abbildung 8.5.: Der Screenshot einer Android-Applikation zur Simulation des Dashboards eines Fahrzeugs des Car-To-X Beispiels.

Aufbau des Experiments

Für dieses Experiment wird eine Spezifikation in der Runtime von SCENARIO-TOOLS durchgespielt und anschließend in der von SBP. Die Laufzeit des Supersteps dieser Spezifikation wird gemessen und verglichen. Die Spezifikation beschreibt ein System, das Nachrichten stufenförmig verbreitet. Codeblock 64 zeigt die ersten zwei Szenarien dieser Spezifikation. Das erste Szenario *Cascade1* reagiert auf eine Nachricht aus der Umgebung und ruft zwei weitere Nachrichten auf. Das zweite Szenario *Cascade2* reagiert auf die Nachrichten des ersten Szenarios und ruft wiederum zwei neue Nachrichten auf. Weitere Szenarien werden in diesem Schema ergänzt. Die Umgebungsnachricht beginnt den Superstep des Systems. Dieser endet nachdem alle Nachrichten der Szenarien aufgetreten sind. Das Experiment wird in fünf Stufen getestet. Jede Stufe erhöht die Anzahl der Szenarien um zwei. Ich messe die Zeit für den Superstep und die Anzahl der aufgetretenen Nachrichten.

```

1  collaboration Doublecascading {
2
3     static role Environment e
4     static role A a
5     static role B b
6
7     specification scenario Cascade1 {
8         message e->a.startCascade()
9         message strict requested a->b.msg1()
10        message strict requested a->b.msg1()
11    }
12
13    specification scenario Cascade2 {
14        message a->b.msg1()
15        message strict requested a->b.msg2()
16        message strict requested a->b.msg2()
17    }
18
19    ...
20
21 }

```

Code 64: Szenarien der Spezifikation Doublecascading für eine stufenförmige Nachrichtenweitergabe.

Ergebnisse und Auswertung

Die Ergebnisse des Experiments habe ich in drei Tabellen zusammengefasst. Tabelle 8.1 zeigt die gemittelten Ergebnisse verschiedener Laufzeitmessungen. Die Ergebnisse zeigen, dass die Laufzeit der SBP Runtime und der SCENARIO-TOOLS Runtime scheinbar identisch sind. Die Laufzeitmessungen der verschiedenen Versuche schwanken um kleinere Werte, scheinen aber immer im gleichen Bereich zu bleiben. Die Anzahl der gesendeten Nachrichten (siehe Tabelle 8.2) in beiden Runtimes ist ebenfalls identisch. Das war zu erwarten, da die Spezifikationen identisch sind. Betrachten wir die gleichzeitig aktiven Szenarien in SCENARIOTOOLS und die gleichzeitig aktiven Threads in SBP, so sehen wir einen gleichmäßigen Anstieg der Werte. Die Werte von SBP sind in jedem Versuch um den Wert sechs höher, als die von SCENARIOTOOLS. Das ist einfach zu erklären, da es in SBP gewisse System-Szenarien gibt, die interne Funktionalitäten umsetzen. Dafür gibt es genau sechs.

Wir haben anfangs eine bessere Performance von SBP erwartet, da hier keine Modelle interpretiert werden, sondern lediglich Code abläuft. Dafür kann es allerdings verschiedene Erklärungen geben:

- Der Schritt von der SML-Semantik zur BP-Semantik von Nachrichten benötigt viel Verwaltungsarbeit. Diese Arbeit steckt in dem Vergleich und der Modifikation von Listen. In Java gibt es verschiedene Implementierungen für Listen, die unterschiedlich performant sind. Möglicherweise geht hier viel Rechenzeit verloren.

8. Evaluation und fortführende Ansätze

- Die Erstellung und Verwaltung von Threads kann in Java teuer sein, da sie auf Threads des Betriebssystems abgebildet werden. Das kann die Performance der Runtime einschränken.
- Die Synchronisation der Threads kann ebenfalls für Performanceverluste sorgen. Für Funktionalitäten außerhalb von Szenarien ist zusätzliche Verwaltung der Threads nötig. So müssen Szenarien auf die Ausführung von Transformationsregeln warten. In dieser Spezifikation gibt es zwar keine Transformationsregeln, dennoch kann diese zusätzliche Verwaltung Performance kosten.

Die Laufzeitperformance ist also nicht wie erwartet besser als die von SCENARIO-TOOLS. Die Anforderung ist also nicht vollständig erfüllt. Gründe dafür und mögliche Lösungen oder Verbesserungen gilt es weiter zu erforschen.

Zeit in ms	2 Szenarien	4 Szenarien	6 Szenarien	8 Szenarien	10 Szenarien
ScenarioTools	31	218	497	1638	6121
SBP Java	33	137	457	1823	5864

Tabelle 8.1.: Gemittelte Ergebnisse der Laufzeitmessung des Supersteps für SCENARIOTOOLS und SBP Java.

# Nachrichten	2 Szenarien	4 Szenarien	6 Szenarien	8 Szenarien	10 Szenarien
ScenarioTools	6	30	126	510	2046
SBP Java	6	30	126	510	2046

Tabelle 8.2.: Anzahl der Nachrichten während der Ausführung.

Szenarien / Threads	2 Szenarien	4 Szenarien	6 Szenarien	8 Szenarien	10 Szenarien
ScenarioTools	2	4	6	8	10
SBP Java	8	10	12	14	16

Tabelle 8.3.: Anzahl gleichzeitig aktiver Szenarien bei der Ausführung in SCENARIOTOOLS und Anzahl gleichzeitig aktiver Threads bei der Ausführung in SBP Java.

8.4.4. Skalierbarkeit

Skalierbarkeit ist bei verteilten reaktiven Systemen ein kompliziertes Thema. Die Laufzeitumgebung von SCENARIOTOOLS verlangt, dass sich alle Komponenten gegenseitig kennen. Außerdem müssen alle Komponenten auf demselben

Stand der Ausführung sein, was bedeutet, dass eine ständige Synchronisation zwischen den Komponenten passieren muss. Jedes auftretende Ereignis muss an alle Komponenten gesendet werden. Bei einem großflächigen System mit vielen Komponenten kann dies einen großen Datenaufwand bedeuten. Einerseits muss jede Komponente alle anderen kennen, was ein großes Objektsystem erfordert. Zudem müssen Unmengen an Nachrichten zwischen den Komponenten versendet werden (Broadcast). Die Implementierung von SBP ist stark an die Ausführungssemantik von SCENARIOTOOLS angelehnt und hält sich daher auch an einige Restriktionen. Daher sind das Kennen des Objektsystems und der Broadcast der Nachrichten in SBP ebenfalls ein Problem. Allerdings bietet SBP die Möglichkeit, Objekte zur Laufzeit zu ergänzen und zu zerstören (siehe Abschnitt 6.8). Daher kann das Objektsystem durchaus zur Laufzeit angepasst werden und muss nicht zwangsläufig bei der Initialisierung komplett vorhanden sein. Dennoch muss darauf geachtet werden, dass die neu hinzukommenden Komponenten einen anderen Systemzustand haben können. So muss dafür gesorgt sein, dass neue Komponenten den nötigen Teil des Objektsystems kennen oder beim Auftreten einer Inkonsistenz eine Synchronisation starten. Im Folgenden stelle ich Ansätze für die Verbesserung der Skalierbarkeit in SBP vor.

Nachrichtenaustausch

Der Nachrichtenaustausch in SCENARIOTOOLS und auch in SBP erwartet eine vollständige Kommunikation. Das bedeutet, dass jede auftretende Nachricht an alle anderen Komponenten gesendet wird. Dies kann bei großen Systemen zu einer Flut von Nachrichten im Netzwerk führen. Um dieses Problem anzugehen, stelle ich hier einen alternativen Nachrichtenaustausch vor. Nicht jede Nachricht muss für alle Komponenten relevant sein. Abbildung 8.6 zeigt das Car-To-X Beispiel mit einer Parallelstraße, auf der das blaue Fahrzeug fährt. Da das blaue Fahrzeug jedoch ein Teil des Systems ist, bekommt es die Nachricht `register` zwischen dem roten Fahrzeug und der Obstacle Control ebenfalls mit. Alternativ können wir sagen: Das blaue Fahrzeug ist sein eigenes Teilsystem, in dem die Baustelle und das rote Fahrzeug irrelevant sind. Daher muss das blaue Fahrzeug nicht über die Nachricht `register` informiert werden. Dies können wir ermitteln, indem wir uns die Szenarien anschauen, die durch die Nachricht gestartet werden. Durch die Nachricht selbst werden zwei Rollen an Objekte (rotes Fahrzeug und Obstacle Control) gebunden. Zudem können durch Szenarien weitere Rollen an Objekte gebunden werden. Diese Objekte werden in den Szenarien in irgendeiner Form referenziert. Es können Nachrichten auftreten oder verboten und Bedingungen an die Objekte geknüpft werden. Das bedeutet,

8. Evaluation und fortführende Ansätze

dass genau diese Objekte für die Szenarien und damit für die Nachricht **register** relevant sind. Alle anderen Objekte können diese Nachricht getrost ignorieren, bzw. müssen diese Nachricht gar nicht erst empfangen. Löst eine Komponente - in diesem Fall das rote Fahrzeug - ein Ereignis aus, muss es lediglich an die Obstacle Control eine Nachricht schicken. Dadurch vermindert sich die Datenflut enorm. So kann der Nachrichtenaustausch auch funktionieren, wenn es Tausende oder Millionen von Komponenten im System gibt.

Implementierung des verminderten Datenaustausches

Für die Evaluation dieses Ansatzes zum verminderten Datenaustausches habe ich diese Methode in einem experimentellen Runtime-Adapter getestet. Der `DistributedMinimizedSendingRuntimeAdapter` implementiert diese Methode. Wenn eine Nachricht in das Netzwerk gesendet werden soll, überprüft dieser Adapter erst, welche Komponenten sich dafür interessieren. Dafür werden die Rollen der aktiven Szenarien betrachtet und alle gebundenen Objekte gesammelt. Nur an die Netzwerkadressen dieser Objekte wird die Nachricht weitergegeben. Die restlichen Komponenten bekommen davon nichts mit. Das sorgt für einen verringerten Datenaustausch. Jedoch sorgt diese Methode auch dafür, dass Szenarien nicht überall aktiviert werden. Das ist zwar auch Sinn der Methode, jedoch wird das Objektsystem durch Nachrichten und Szenarien verändert. Dies geschieht nun nicht in allen Komponenten und führt zu einer Inkonsistenz der Objektsysteme. Dies führt ohne weitere Eingriffe in die Ausführung zu schweren Fehlern, da Rollen in bestimmten Szenarien nun falsch oder gar nicht gebunden werden.

Es besteht also das Problem, dass durch das Weglassen von Nachrichten eine Inkonsistenz zwischen den lokalen Objektsystemen der verschiedenen Komponenten entsteht. Es muss hier eine Art der Synchronisation des Objektsystems stattfinden, wenn diese Komponenten erneut miteinander interagieren sollen.

Synchronisation des Objektsystems

Gibt es Inkonsistenzen zwischen den lokalen Objektsystemen verteilter Komponenten eines Systems, kann es zu unerwünschtem Verhalten führen. So können sich Objektreferenzen im Objektsystem ändern und neue Objekte an eine vorher belegte Stelle treten. Werden jetzt Rollen durch Szenarien gebunden, geschieht dies über diese Referenzen. So können in zwei Komponenten durch das selbe Szenario zwei verschiedene Objekte an eine Rolle gebunden werden. Dies kann zu unvorhersehbarem Verhalten führen und schwere Fehler verursachen. Die lokalen Objektsysteme müssen also synchron gehalten werden. Hier bieten sich zwei Möglichkeiten an, die in den folgenden Abschnitten beschrieben wer-

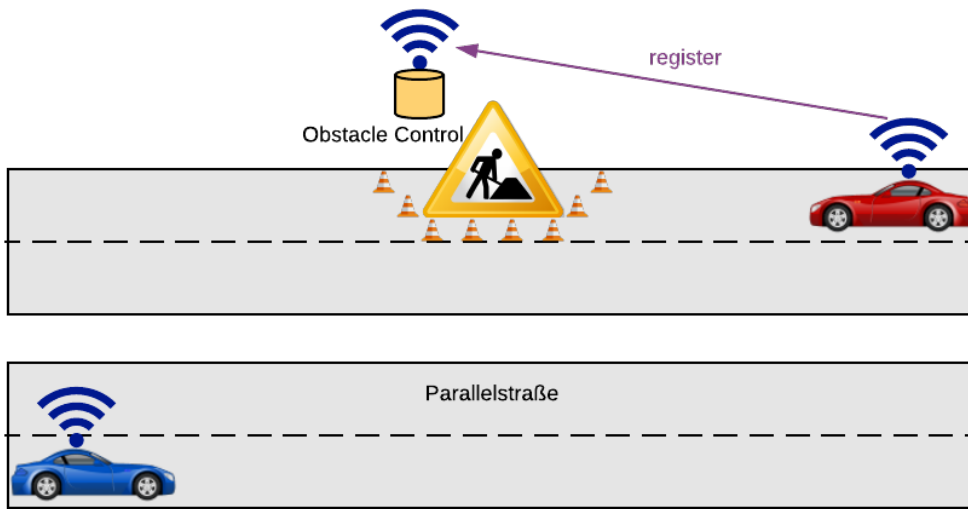


Abbildung 8.6.: Das Car-To-X Beispiel mit einer Parallelstraße. Das blaue Fahrzeug befindet sich auf einer Parallelstraße und trifft nicht auf die Baustelle.

den. SBP ermöglicht durch seine Erweiterbarkeit die Implementierung beliebiger Runtime-Adapter für das Netzwerk. Hier kann ebenfalls eine Synchronisation des Objektsystems implementiert werden.

Dezentralisierte Synchronisation

Da die Rollen über Referenzen auf einem Objekt gebunden werden, können wir davon ausgehen, dass dieses Objekt die korrekte Referenz besitzt. Nehmen wir an, die Komponente car1 führt den Beispielausdruck „binde Rolle car2 an obstacleControl.waitingCar“ aus. So soll die Rolle car2 an die Referenz waitingCar des Objekts obstacleControl gebunden werden. Für die Komponente car1 ist diese Referenz allerdings nicht gesetzt, für die Komponente obstacleControl allerdings schon. Komponente car1 kann hier einen Fehler vorhersehen und fragt daher bei der Komponente obstacleControl an, was seine Referenz von waitingCar beinhaltet. Mit dieser Information kann car1 sein lokales Objektsystem aktualisieren und die Rolle korrekt binden. Dies ist eine dezentralisierte Synchronisation der einzelnen Komponenten untereinander. Abbildung 8.7 zeigt ein Schema dieser dezentralisierten Synchronisation.

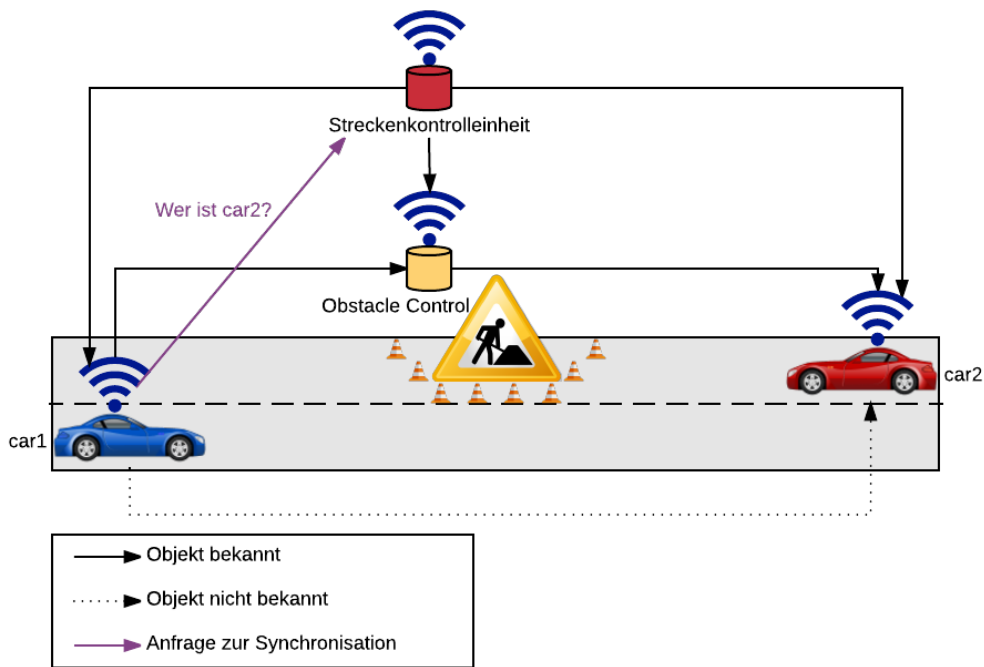


Abbildung 8.8.: Die zentralisierte Synchronisation mit Hilfe einer Streckenkontrollleinheit.

dieser Fahrzeuge speichern wir dann dessen Zustand in unserem Objektsystem und gegebenenfalls Netzwerkdaten, wie IPs oder Hostnames. Diese Datensammlung wird immer umfangreicher und nimmt irgendwann inakzeptabel viel Speicher ein. Vielen Fahrzeugen begegnet man nur einmal oder kommuniziert mit ihnen nur für kurze Zeit. Diese Fahrzeuge können wir dann ab einem bestimmten Zeitpunkt vergessen. Wenn dieser Zeitpunkt gekommen ist, können wir wieder über die Szenarien ermitteln. Kommt ein Fahrzeug in keinem Szenario mehr vor, ist es für uns irrelevant geworden und kann aus dem lokalen Objektsystem gelöscht werden. SBP ermöglicht diese Löschung von Objekten zur Laufzeit und kann somit dieses Speicherproblem lösen.

8. *Evaluation und fortführende Ansätze*

9. Verwandte Arbeiten

In diesem Kapitel stelle ich Arbeiten vor, die sich mit denselben oder ähnlichen Themen befassen, wie diese Arbeit. Es gibt bereits verschiedene Arbeiten im Bereich der szenariobasierten Modellierung. Einige davon haben den Grundstein für die Entwicklung von SCENARIOTOOLS und der Scenario Modeling Language gelegt.

9.1. Scenario-based Programming mit PlayGo

PlayGo ist eine IDE für die szenariobasierte Entwicklung. Es wurde von einem Team um Harel et al. [7] vorgestellt und soll für die Entwicklung und Analyse von szenariobasierten Systemen verwendet werden. PlayGo implementiert die visuelle Sprache der *Live Sequence Charts* (LSC) [1, 6], die eine Erweiterung der bekannten UML Sequenzdiagramme darstellen. Die Sequenzdiagramme werden um Modalitäten und eine Ausführungssemantik erweitert. Dadurch sind diese Diagramme in Sinne eines Play-Out-Verfahrens [8] ausführbar. Die Idee dahinter ist es, das Programmieren bei der Entwicklung vollständig zu eliminieren [5] und der Maschine einfach zu sagen, was sie zu tun hat. Aus dieser Idee und der Sprache der LSCs ist auch SCENARIOTOOLS und SML entstanden. Wir haben die Problematik an der visuellen Sprache erkannt und den Prototypen einer textuellen Sprache [10] entwickelt. Daraus ist nach weiterer Entwicklung die Scenario Modeling Language¹ entstanden, die den Grundstein der Methodik von PlayGo weiterführt.

9.2. Codegenerierung von LSCs nach AspectJ

In ihrer Arbeit stellen Maoz und Harel [12] eine Methode zur Umwandlung von szenariobasierten Anforderungen in ausführbaren, aspektorientierten Code. Sie zeigen, wie sich multimodale szenariobasierte Spezifikationen in Form von LSCs zu Javacode generieren lassen, der sich anschließend in bestehenden Code einpflegen lässt. Durch den szenariobasierten Ansatz haben sie allerdings Probleme

¹<http://scenariotools.org/scenario-modeling-language/>

9. Verwandte Arbeiten

der Performance bemerkt, da während der Ausführung viel Koordination nötig war.

9.3. Codegenerierung aus UML Sequenzdiagrammen

In ihrer Arbeit stellen Kundu et al. [11] eine Codegenerierung ausgehend von UML Sequenzdiagrammen vor. Dabei verwenden sie Diagramme in dem Format *XML Metadata Interchange* (XMI). Dies ist ein Instanzmodell des Diagramms. Aus diesem Modell erzeugen sie einen *Sequence Integration Graph*, der zusätzliche Informationen zu Nachrichten und Kontrollfluss beinhaltet. Daraus wird anschließend ausführbarer Javacode generiert. Der generierte Code repräsentiert den Ablauf des Sequenzdiagramms, ist allerdings aufgrund von fehlenden Informationen des Quelldiagramms unvollständig.

9.4. Akka

*Akka*² ist ein Toolkit mit einer Runtime, zur Herstellung von Nachrichtenorientierten Applikationen. Es bedient sich einem Konzept, das sich *Actor* nennt. Ein Actor ist eine Softwarekomponente mit einer reaktiven Natur. Er kann Nachrichten empfangen und darauf mit Nachrichten reagieren. Akka bietet eine Implementierung in Scala und Java. Durch die Java-Implementierung bietet sich hier auch die Möglichkeit, plattformspezifischen Code einzupflegen, was eine Anforderung an diese Arbeit war. Akka bedient sich verschiedener Verfahren, die es sehr performant und erweiterbar machen soll. Zudem bietet es die Möglichkeit einer dezentralisierten Ausführung. Akka hat zwar keinen szenariobasierten Ansatz, aber einen reaktiven, ähnlich wie BP. Es wäre für zukünftige Arbeiten interessant, die in dieser Arbeit vorgestellte szenariobasierte Entwicklungsmethodik für Java mit Akka statt BP als Unterbau umzusetzen. Als Beispiel für ein mit Akka implementiertes System wird eine Fahrstuhlsteuerung³ geboten.

²<http://akka.io/>

³<https://github.com/tombray/akka-fsm-examples>

10. Fazit

10.1. Zusammenfassung

Moderne Softwaresysteme werden immer komplexer, weshalb wir Methoden für die Entwicklung immer weiter verbessern müssen. Die moderne Softwareentwicklung verwendet eine szenariobasierte Entwicklungsmethodik, die sich bislang auf die Modellierungsphase beschränkt. Um bei der Implementierung keinen methodischen Bruch hinnehmen zu müssen, wollten wir eine Methodik für die szenariobasierte Programmierung in Java schaffen.

In dieser Arbeit wurde eine solche Methodik vorgestellt. Sie umfasst eine Java Bibliothek, die die Konzepte der Modellierungssprache SML nachbildet und Patterns und Klassen bereitstellt. Dadurch bietet sich die Möglichkeit, in Java selbst Szenarien mit entsprechender Semantik und Ausführungslogik zu entwickeln. Diese Szenarien können in einer Spezifikation gesammelt und als szenariobasierte Anwendung auf ein Zielsystem gebracht werden. Die Ausführungssemantik von SBP entspricht derjenigen von SCENARIOTOOLS und ist damit genauso zu verwenden. Das ist gut, da eine solche SML Spezifikation leicht auf eine SBP Spezifikation abgebildet werden kann. Für diese Abbildung habe ich ein Mapping vorgestellt, welches für einen Codegenerator verwendet werden kann. Die SBP Runtime bietet die Möglichkeit, eine verteilte Ausführung über ein Netzwerk schnell zu realisieren. Außerdem können durch den implementierten Codegenerator nun auch bereits bestehende SML Spezifikationen umgewandelt und verteilt ausgeführt werden, was als großer Fortschritt betrachtet werden kann. Für die Anwendung dieser szenariobasierten Programmierung in Java habe ich einige methodische Ansätze gegeben, die bei der Umsetzung eines Softwaresystems in SBP verwendet werden können. Besonders will ich die Erweiterbarkeit der SBP Bibliothek für Java hervorheben. So können ohne viel Aufwand weitere Kommunikationswege implementiert werden. Auch plattformspezifischer Code kann auf mehreren Wegen in eine Spezifikation integriert werden. Durch die Leichtgewichtigkeit von Java gegenüber größeren Toolplattformen ist diese Methodik auf verschiedenen Plattformen verwendbar und kann beispielsweise auch auf Android einfach verwendet werden.

Wir haben allerdings auch gesehen, dass es in den Bereichen Performance und

10. Fazit

Skalierbarkeit noch Nachbesserungsbedarf gibt. Die Laufzeit der SBP Runtime ist entgegen unserer Erwartungen nicht schneller als die von SCENARIOTOOLS, was weiter erforscht werden sollte. Außerdem haben wir ähnliche Probleme bezüglich der Skalierbarkeit für große Systeme, wie wir sie auch in SCENARIOTOOLS haben. SBP bietet dahingehend bereits Fortschritte, was die Objekterzeugung und -zerstörung angeht, löst jedoch das Problem der Nachrichtenflut noch nicht. Dafür habe ich Lösungsansätze für eine Synchronisation der Komponenten und ein Weglassen gewisser Nachrichten vorgestellt.

Zusammenfassend haben wir mit der szenariobasierten Programmierung in Java eine Methodik kennengelernt, die den methodischen Bruch beim Übergang von der Modellierung zur Implementierung eines Systems vermeidet und die verteilte Ausführung einer Spezifikation ermöglicht.

10.2. Ausblick

Das szenariobasierte Programmieren für Java stellt eine Entwicklungsmethodik dar, die die Konzepte von SML zum Startzeitpunkt ihrer Entwicklung auf Java abbildet. Während diese Arbeit entstanden ist, wurde SML allerdings ebenfalls weiterentwickelt und besitzt nun neue Konzepte, die SBP noch nicht betrachtet. Für zukünftige Arbeiten kann es ein Ziel sein, diese neuen Konzepte in SBP umzusetzen. So wurde ein neues Ranking für die Wichtigkeit des Auftretens von Nachrichten den Superstep betreffend entwickelt. Da BP bereits selbst eine prioritätsgesteuerte Ausführung bietet, die wir für Szenarien nur bedingt nutzen, kann dieses neue Konzept möglicherweise eine interessante Verwendung dafür finden.

Die Erweiterbarkeit von SBP bietet genügend Anlaufstellen für verschiedene Projekte. Es können beispielsweise neue Runtime-Adapter für eine effiziente oder verschlüsselte Kommunikation entwickelt werden. Zudem bieten sich weitere Protokolle zur Implementierung an, die eine Kommunikation über das Internet ermöglichen. Ich habe auch Probleme der Skalierbarkeit genannt, für die es bereits Lösungsansätze gibt. Diese Ansätze umzusetzen und weiter auszuführen stellt eine interessante Fortführung dieser Arbeit dar.

Für weitere Arbeiten an SBP kann es auch interessant sein, eine andere Basis für die Runtime zu testen. Da die Übersetzung der Semantik von SML nach BP einiges an Verwaltungsaufwand benötigt, könnten andere Ansätze effizienter sein. Es wäre zum Beispiel interessant, ob Akka eine geeignetere Basis sein könnte oder ob eine ganz eigene Implementierung die gewünschte Effizienz bietet.

A. Anhang

A.1. Repository zum Projekt

Das gesamte Projekt mit dem Sourcecode der SBP Bibliothek für Java ist zu finden im Repository auf <https://bitbucket.org/fwhkoenig/scenario-based-programming-java>. Der aktuelle Commit hat die ID 5807f35.

A.2. Download der Bibliothek

Die SBP Bibliothek für Java kann heruntergeladen und in ein Java Projekt eingebunden werden. Damit das funktioniert, muss BP als Bibliothek ebenfalls eingerichtet sein. Die Bibliothek ist hier zu finden: <https://bitbucket.org/fwhkoenig/scenario-based-programming-java/downloads/SBP.jar>

Für die GUI-Implementierung des Codegenerators ist zudem eine Erweiterung der Bibliothek nötig. Zu finden ist sie hier: https://bitbucket.org/fwhkoenig/scenario-based-programming-java/downloads/SBP_UI.jar

A.3. Inhalt der DVD

Im Folgenden ist der Inhalt der beigelegten DVD aufgelistet:

- Ausarbeitung im Format PDF
- Die SBP Bibliothek
- VirtualBox Image einer Lubuntu Installation. Darauf befindet sich eine Installation von ScenarioTools mit den in dieser Arbeit erstellten Plugins zur Codegenerierung. Dazu sind die Car-To-X Spezifikation und eine SBP Implementierung vorhanden.

A. Anhang



Abbildung A.1.: Menü des Codegenerators für die Transformation von SML nach SBP.

A.4. Verwendung des Codegenerators

Zur Verwendung des Codegenerators muss zusätzlich zu SCENARIOTOOLS das Plugin SBP.Generator aus dem Repository installiert werden. Dann kann auf eine beliebige Runconfig-Datei von SCENARIOTOOLS mit einem Rechtsklick das Menü „Scenario-based Programming“ geöffnet werden (siehe Abbildung A.1). Darin sind die beiden Befehle für die Codegenerierung enthalten. Der Erste erzeugt Code für die SBP Spezifikation und der Zweite erzeugt den Code für die GUI.

A.5. Car-To-X Spezifikation in SML

Der Codeblock 65 zeigt die vollständige Spezifikation des vorgestellten Car-To-X-Systems.

```
1 import "../model/cartox.ecore"
2
3 system specification CarToX {
4
5     domain cartox
6
7     define Car as controllable
8     define ObstacleControl as controllable
9     define Environment as uncontrollable
10    define Driver as uncontrollable
11    define Dashboard as uncontrollable
12    define Obstacle as uncontrollable
13    define LaneArea as uncontrollable
14    define StreetSection as uncontrollable
15
16    collaboration CarDriving {
17
18        dynamic role Environment env
19        dynamic role Car car
20
21        specification scenario CarMovesToNextArea {
22            message env -> car.movedToNextArea
23        }
24
25        specification scenario CarChangesToOppositeArea {
26            message env -> car.changedToOppositeArea
27        }
28
29        specification scenario CarMovesToNextAreaOnOvertakingLane {
```


A.5. Car-To-X Spezifikation in SML

```
30     message env -> car.movedToNextAreaOnOvertakingLane
31   }
32
33 }
34
35 collaboration CarApproachingObstacleAssumptions {
36
37   dynamic role Environment env
38   dynamic role Car car
39   dynamic role LaneArea currentArea
40   dynamic role LaneArea nextArea
41   dynamic role LaneArea oppositeNextArea
42   dynamic role Obstacle obstacle
43
44   /*
45    * When a car approaches an obstacle on the blocked lane,
46    * an according event will occur
47    */
48   assumption scenario ApproachingObstacleOnBlockedLaneAssumption_CarOnRightLane
49   with dynamic bindings [
50     bind currentArea to car.inArea
51     bind nextArea to currentArea.next
52     bind obstacle to nextArea.obstacle
53   ] {
54     message env -> car.movedToNextArea
55     interrupt if [ obstacle == null ]
56     message strict requested env -> car.setApproachingObstacle(obstacle)
57   } constraints [
58     forbidden message env -> car.movedToNextAreaOnOvertakingLane()
59   ]
60
61   /*
62    * When a car approaches an obstacle but is currently driving on the lane that is
63    * not blocked, an according event will occur
64    */
65   assumption scenario ApproachingObstacleOnBlockedLaneAssumption_CarOnLeftLane
66   with dynamic bindings [
67     bind currentArea to car.inArea
68     bind nextArea to currentArea.next
69     bind oppositeNextArea to nextArea.opposite
70     bind obstacle to oppositeNextArea.obstacle
71   ] {
72     message env -> car.movedToNextArea
73     interrupt if [ obstacle == null ]
74     message strict requested env -> car.setApproachingObstacle(obstacle)
75   } constraints [
76     forbidden message env -> car.movedToNextArea()
77   ]
78
79 }
80
81 collaboration CarEntersOrLeavesNarrowPassageAssumptions {
82
83   dynamic role Environment env
84   dynamic role Car car
85   dynamic role LaneArea currentArea
86   dynamic role LaneArea oppositeArea
87   dynamic role LaneArea oppositePrevArea
88
89   /*
90    * When a car enters the narrow passage area,
91    * a corresponding event "entersNarrowPassage" will occur.
92    */
93   assumption scenario CarEntersNarrowPassage_ObstacleOnRightLane
94   with dynamic bindings [
95     bind currentArea to car.inArea
96     bind oppositeArea to currentArea.opposite
97     bind oppositePrevArea to oppositeArea.previous
98   ] {
99     message env -> car.movedToNextAreaOnOvertakingLane
100    // If on the right side is an obstacle AND before there was no obstacle
```

A. Anhang

```
101 // then the car enters the narrow passage
102 interrupt if [ oppositeArea.obstacle == null | oppositePrevArea.obstacle !=
      null ]
103 message strict requested env -> car.hasEnteredNarrowPassage
104 }
105
106 assumption scenario CarEntersNarrowPassage_ObstacleOnLeftLane
107 with dynamic bindings [
108   bind currentArea to car.inArea
109   bind oppositeArea to currentArea.opposite
110   bind oppositePrevArea to oppositeArea.next
111 ] {
112   message env -> car.movedToNextArea
113   // If on the left side is an obstacle AND before there was no obstacle
114   // then the car enters the narrow passage
115   interrupt if [ oppositeArea.obstacle == null | oppositePrevArea.obstacle !=
      null ]
116   message strict requested env -> car.hasEnteredNarrowPassage
117 }
118
119 /*
120  * When a car has left the narrow passage area,
121  * a corresponding event "hasLeftNarrowPassage" will occur.
122  */
123 assumption scenario CarHasLeftNarrowPassage_ObstacleOnRightLane
124 with dynamic bindings [
125   bind currentArea to car.inArea
126   bind oppositeArea to currentArea.opposite
127   bind oppositePrevArea to oppositeArea.previous
128 ] {
129   message env -> car.movedToNextAreaOnOvertakingLane
130   interrupt if [ oppositeArea.obstacle != null | oppositePrevArea.obstacle ==
      null ]
131   message strict requested env -> car.hasLeftNarrowPassage
132 } constraints [
133   forbidden message env -> car.changedToOppositeArea()
134 ]
135
136 assumption scenario CarHasLeftNarrowPassage_ObstacleOnLeftLane
137 with dynamic bindings [
138   bind currentArea to car.inArea
139   bind oppositeArea to currentArea.opposite
140   bind oppositePrevArea to oppositeArea.next
141 ] {
142   message env -> car.movedToNextArea
143   interrupt if [ oppositeArea.obstacle != null | oppositePrevArea.obstacle ==
      null ]
144   message strict requested env -> car.hasLeftNarrowPassage
145 } constraints [
146   forbidden message env -> car.changedToOppositeArea()
147 ]
148
149 }
150
151 collaboration ShowInformationOnDashboard {
152
153   dynamic role Environment env
154   dynamic role Car car
155   dynamic role Driver driver
156   dynamic role Dashboard dashboard
157   dynamic role ObstacleControl obstacleControl
158
159   /*
160    * Show stop or go to the driver when approaching the obstacle
161    * before actually reaching it
162    */
163   specification scenario DashboardOfCarOnBlockedLaneShowsStopOrGo
164   with dynamic bindings [
165     bind driver to car.driver
166     bind dashboard to car.dashboard
167   ] {
```

A.5. Car-To-X Spezifikation in SML

```
168     message env -> car.setApproachingObstacle(*)
169     alternative {
170         message strict car -> dashboard.showGo
171     } or {
172         message strict car -> dashboard.showStop
173     }
174 }
175
176 /*
177  * The dashboard shows a go or a stop light as reaction to the obstacle
178  * controls response after registering
179  */
180 specification scenario DashboardShowsGo
181 with dynamic bindings [
182     bind dashboard to car.dashboard
183 ] {
184     message obstacleControl -> car.enteringAllowed
185     message strict requested car -> dashboard.showGo
186 }
187
188 specification scenario DashboardShowsStop
189 with dynamic bindings [
190     bind dashboard to car.dashboard
191 ] {
192     message obstacleControl -> car.enteringDisallowed
193     message strict requested car -> dashboard.showStop
194 }
195
196 }
197
198 collaboration CarsRegisterWithObstacleControl {
199
200     dynamic role Environment env
201     dynamic role Car car
202     dynamic role Car nextCar
203     dynamic role Dashboard dashboard
204     dynamic role Obstacle obstacle
205     dynamic role ObstacleControl obstacleControl
206     dynamic role LaneArea currentArea
207     dynamic role LaneArea nextArea
208
209     /*
210      * Stop or go must be shown to the driver when the car
211      * approaches an obstacle
212      */
213     specification scenario ControlStationAllowsOrDisallowsCarToEnterNarrowPassage
214     with dynamic bindings [
215         bind obstacle to car.approachingObstacle
216         bind obstacleControl to obstacle.controlledBy
217         bind dashboard to car.dashboard
218     ] {
219         message env -> car.setApproachingObstacle(*)
220         message strict requested car -> obstacleControl.register
221         alternative {
222             message strict requested obstacleControl -> car.enteringAllowed
223         } or {
224             message strict requested obstacleControl -> car.enteringDisallowed
225         }
226     }
227
228     specification scenario ControlStationAllowsOrDisallowsCarOnBlockedLaneToEnter
229     with dynamic bindings [
230         bind obstacle to car.approachingObstacle
231         bind obstacleControl to obstacle.controlledBy
232         bind dashboard to car.dashboard
233     ] {
234         message env -> car.setApproachingObstacle(*)
235         message strict requested car -> obstacleControl.register
236         alternative if [ obstacleControl.carsOnNarrowPassageLaneAllowedToPass.isEmpty()
237             ] {
238             message strict requested obstacleControl -> car.enteringAllowed
```

A. Anhang

```
238     message strict car -> dashboard.showGo
239   } or if [ ! obstacleControl.carsOnNarrowPassagelaneAllowedToPass.isEmpty() ] {
240     message strict requested obstacleControl -> car.enteringDisallowed
241     message strict car -> dashboard.showStop
242   }
243 }
244
245 specification scenario CarUnregistersAfterLeavingNarrowPassage
246 with dynamic bindings [
247   bind obstacle to car.approachingObstacle
248   bind obstacleControl to obstacle.controlledBy
249 ] {
250   message env -> car.hasLeftNarrowPassage
251   message strict requested car -> obstacleControl.unregister
252 }
253
254 /*
255  * The obstacle control tells the next car that it may now enter the
256  * narrow passage
257  */
258 specification scenario ControlStationTellsNextRegisteredCarToEnter
259 with dynamic bindings [
260   bind nextCar to obstacleControl.registeredCarsWaiting.first()
261 ] {
262   message car -> obstacleControl.unregister
263   interrupt if [ nextCar == null ]
264   message strict requested obstacleControl -> nextCar.enteringAllowed
265 }
266
267 /*
268  * A car that has been told to wait may eventually continue driving
269  * and pass the obstacle
270  */
271 specification scenario CarEventuallyAllowedToPassObstacle {
272   message obstacleControl -> car.enteringDisallowed()
273   message strict obstacleControl -> car.enteringAllowed()
274 }
275
276 /*
277  * A car that has been told that it may pass the obstacle will enter
278  * the narrow passage
279  */
280 assumption scenario CarDrivesIntoNarrowPassageIfItMayPassObstacle_CarOnRightLane
281 with dynamic bindings [
282   bind currentArea to car.inArea
283   bind nextArea to currentArea.next
284   bind env to car.environment
285 ] {
286   message obstacleControl -> car.enteringAllowed
287   interrupt if [ car.onLane != car.drivingInDirectionOfLane ]
288   alternative if [ nextArea.obstacle == null ] {
289     message strict requested env -> car.movedToNextArea
290   } or if [ nextArea.obstacle != null ] {
291     message strict requested env -> car.changedToOppositeArea
292     message strict requested env -> car.movedToNextAreaOnOvertakingLane
293   }
294 }
295
296 assumption scenario CarDrivesIntoNarrowPassageIfItMayPassObstacle_CarOnLeftLane
297 with dynamic bindings [
298   bind currentArea to car.inArea
299   bind nextArea to currentArea.previous
300   bind env to car.environment
301 ] {
302   message obstacleControl -> car.enteringAllowed
303   interrupt if [ car.onLane == car.drivingInDirectionOfLane ]
304   alternative if [ nextArea.obstacle == null ] {
305     message strict requested env -> car.movedToNextAreaOnOvertakingLane
306   } or if [ nextArea.obstacle != null ] {
307     message strict requested env -> car.changedToOppositeArea
308     message strict requested env -> car.movedToNextArea
```

A.6. Car-To-X Klassendiagramm

```
309     }  
310   }  
311  
312   }  
313  
314 }
```

Code 65: Vollständige SML-Spezifikation für das Car-To-X-System.

A.6. Car-To-X Klassendiagramm

Abbildung A.2 zeigt das Klassendiagramm zum Car-To-X System.

A.7. Transformationsregeln-Henshin

Abbildung A.3 zeigt die Henshin-Regeln, die für die Car-To-X Spezifikation verwendet werden.

A. Anhang

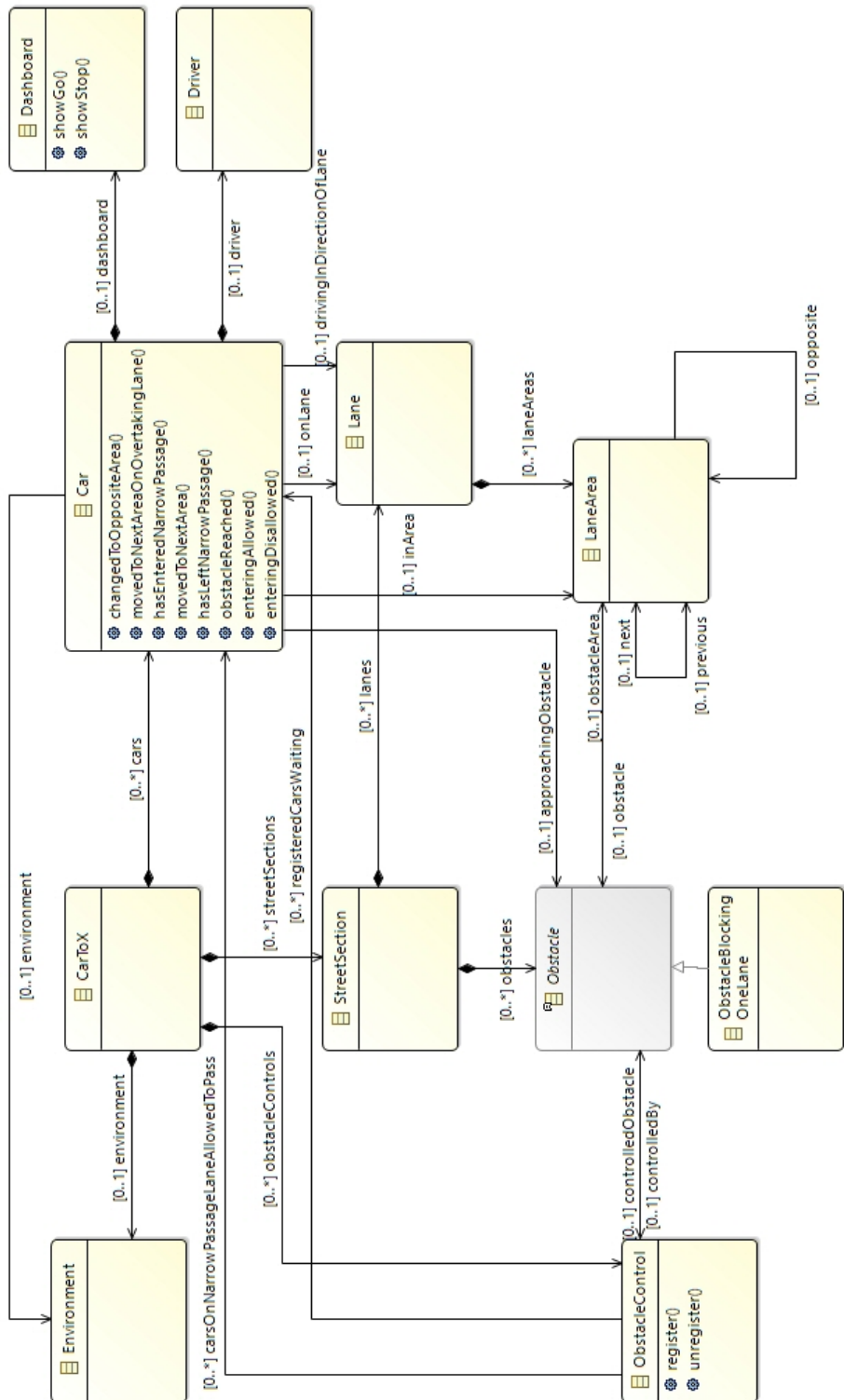


Abbildung A.2.: Klassendiagramm des Car-To-X Beispiels.

A.7. Transformationsregeln-Henshin

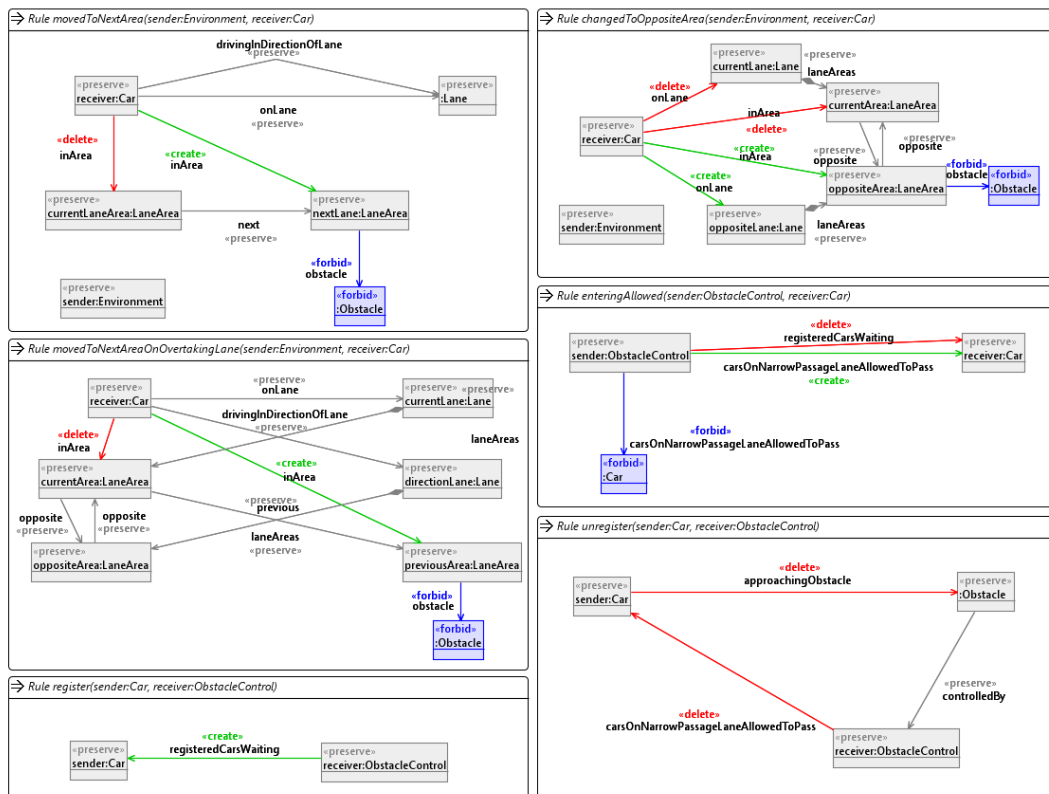


Abbildung A.3.: Henshin-Regeln für die Car-To-X Spezifikation.

A. Anhang

Literaturverzeichnis

- [1] W. Damm and D. Harel. Lscs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.
- [2] J. Greenyer, C. Brenner, and V. P. La Manna. The scenariotools play-out of modal sequence diagram specifications with environment assumptions. *Electronic Communications of the EASST*, 58, 2013.
- [3] J. Greenyer, D. Gritzner, T. Gutjahr, T. Duentel, S. Dulle, F. Deppe, N. Glade, M. Hilbich, F. Koenig, J. Luennemann, et al. Scenarios@ run. time-distributed execution of specifications on iot-connected robots. In *Proceedings of the 10th International Workshop on Models@ Run. Time (MRT 2015), co-located with MODELS*, volume 2015, 2015.
- [4] J. Greenyer, D. Gritzner, G. Katz, A. Marron, N. Glade, T. Gutjahr, and F. König. Distributed execution of scenario-based specifications of structurally dynamic cyber-physical systems. *Procedia Technology*, 26:552–559, 2016.
- [5] D. Harel. Can programming be liberated, period? *Computer*, 41(1):28–37, 2008.
- [6] D. Harel and S. Maoz. Assert and negate revisited: Modal semantics for uml sequence diagrams. *Software & Systems Modeling*, 7(2):237–252, 2008.
- [7] D. Harel, S. Maoz, S. Szekely, and D. Barkan. Playgo: Towards a comprehensive tool for scenario based programming. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ASE '10*, pages 359–360, New York, NY, USA, 2010. ACM.
- [8] D. Harel and R. Marelly. *Come, let's play: scenario-based programming using LSCs and the play-engine*, volume 1. Springer Science & Business Media, 2003.
- [9] D. Harel, A. Marron, and G. Weiss. Behavioral programming. *Commun. ACM*, 55(7):90–100, July 2012.

Literaturverzeichnis

- [10] F. W. H. König. Mspec eine technik zur textuellen modellierung und simulation multimodaler szenariobasierter spezifikationen.
- [11] D. Kundu, D. Samanta, and R. Mall. Automatic code generation from unified modelling language sequence diagrams. *IET Software*, 7(1):12–28, 2013.
- [12] S. Maoz and D. Harel. From multi-modal scenarios to code: Compiling lscs into aspectj. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '06/FSE-14*, pages 219–230, New York, NY, USA, 2006. ACM.