Bewertender Vergleich und Erweiterung unterschiedlicher UML-Simulatoren zur Bestimmung der Modellüberdeckung

Diplomarbeit im Fach Informatik

vorgelegt von

Dominik Schindler

geb. 30.06.1980 in Oberviechtach

angefertigt am

Institut für Informatik
Lehrstuhl für Software Engineering (Informatik 11)
Friedrich-Alexander-Universität Erlangen-Nürnberg
(Prof. Dr. Francesca Saglietti)

Prüfer: Prof. Dr. Francesca Saglietti

Betreuer: Dipl.-Inf. Claudia Schieber

Beginn der Arbeit: 10.07.2006
Abgabe der Arbeit: 10.01.2007

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 10.01.2007

Dominik Schindler

Zusammenfassung

Moderne Softwaresysteme sind heutzutage sehr komplex. Bedingt durch diese Komplexität steigt auch die Wahrscheinlichkeit von Restfehlern, die insbesondere bei hochzuverlässiger und sicherheitskritischer Software katastrophale Folgen haben können.

Das UnITeD Projekt versucht, die hohen Verifikations- und Validierungskosten sicherheitskritischer und hochzuverlässiger Software durch Automatisierung signifikant zu reduzieren und gleichzeitig die Erkennung von Restfehlern zu optimierten. Dazu verwendet es evolutionäre Verfahren um aus UML-Diagrammen automatisch Testdaten zu generieren. Das Projekt gliedert sich in mehrere Aufgabenpakete, wovon eines die automatische Erfassung des Testablaufs ist.

In diesem Aufgabenteil des Projekts sollen die Testdaten bezüglich der von ihnen erfüllten Testkriterien untersucht werden, damit die Optimierung der Daten präzise gesteuert werden kann. Dazu wird ein Simulator benötigt, der ein UML Modell automatisch simulieren kann und die bei der Simulation überdeckten Elemente zurückliefert.

Die Aufgabe dieser Arbeit ist, einen für das Projekt geeigneten Simulator auszuwählen und dafür eine Schnittstelle zu implementieren. Leider zeigt der Vergleich zweier Simulatoren, dass die bestehenden Simulationswerkzeuge die Anforderungen nur ungenügend erfüllen. Aus diesem Grund wurde im Rahmen der Arbeit ein Simulator neu entwickelt, der alle Anforderungen erfüllt.

Die vorliegende Arbeit gliedert sich wie folgt: Kapitel 2 beschreibt nochmal die Aufgabenstellung und zeigt einen Beispiel-Zustandsautomaten, der im Verlauf der Arbeit verwendet wird, um verschiedene Sachverhalte zu beschreiben. In Kapitel 3 werden die Anforderungen an einen Simulator für das Projekt erörtert und die Eigenschaften zweier kommerzieller Simulatoren verglichen. Kapitel 4 beschäftigt sich mit der Simulierbarkeit von UML Verhaltensdiagrammen und insbesondere mit den Einschränkungen, die die Elemente eines Zustandsautomaten erfahren müssen, damit eine automatische Simulation möglich ist. Den Hauptteil bildet Kapitel 5, in dem der neu entwickelten Simulator ausführlich beschrieben wird. Das Kapitel 6 schließt durch einen Ausblick auf mögliche Verbesserungen und Erweiterungen des Simulators die Arbeit ab.

Abstract

Modern software systems are very complex. As a result of the complexity the probability of remaining failures in software is also rising. These remaining failures may have disastrous consequences in safety critical and high reliable software.

The UnITeD Project tries to significantly reduce the high verification and validations costs of safety critical and high reliable software and is attempting to optimize the detection of remaining failures by automation. For this purpose, it uses evolutionary methods to automatically generate test data from a UML model. The project is divided into several activities, whereof one is the gathering of the test flow.

In this part of the project, the test data is to be analyzed regarding to the test criterias, which they achieve. This information is then used to precisely control the optimization of the test data. Therefore a simulator, which could automatically simulate the UML model and which returns the covered elements after the simulation, is needed by the project.

The main task of this paper is finding an appropriate simulator for the project, and afterwards implementing an interface for this simulator. Unfortunately, the comparison of two common simulators reveals, that the existing simulators achieve the needed requirements inadequately. Therefore, a new simulator has been developed in the context of this paper, which achieves all the requirements.

This paper is divided into six chapters: Chapter 2 describes the problem and shows a sample state machine, which is used along the paper to describe several circumstances graphically. Chapter 3 illustrates the requirements, which a simulator must achieve, and compares the properties of two commercial simulator. Chapter 4 examines the simulateability of the different UML diagrams, and particularly describes the constraints of the state machine elements. The main part is chapter 5, where the implemented simulator is described. Chapter 6 concludes the paper with an outlook of possible improvements and extensions of the simulator.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Moderne Softwaresysteme sind heutzutage sehr komplex. Bedingt durch diese Komplexität steigt auch die Wahrscheinlichkeit von Restfehlern, die insbesondere bei hochzuverlässiger und sicherheitskritischer Software katastrophale Folgen haben können.

Ein bekanntes Beispiel eines Software bedingten Unfalls mit Todesfolge ist der Therac-25 Linearbeschleuniger, der zwischen 1982 und 1985 in Kliniken in den USA und in Kanada zur Strahlentherapie verwendet wurde. Durch einen Softwarefehler aufgrund mangelnder Qualitätssicherung war ein schwerer Funktionsfehler möglich, der für den Tod von drei Patienten und weiteren Schwerverletzten verantwortlich war.

Wie das Beispiel zeigt, ist es besonders bei sicherheitskritischer und hochzuverlässiger Software wichtig, ausreichend und gründlich zu testen. Deshalb besitzt gerade diese Art von Software sehr hohe Verifikations- und Validierungskosten.

Das UnITeD Projekt versucht, diese Kosten durch Automatisierung signifikant zu reduzieren und gleichzeitig die Erkennung von Restfehlern zu optimierten. Dazu verwendet es evolutionäre Verfahren um aus UML-Diagrammen automatisch Testdaten zu generieren. Das Projekt gliedert sich in mehrere Aufgabenpakete, wovon eines die automatische Erfassung des Testablaufs ist.

In diesem Aufgabenteil des Projekts sollen die Testdaten bezüglich der von ihnen erfüllten Testkriterien untersucht werden, damit die Optimierung der Daten präzise gesteuert werden kann. Dazu wird ein Simulator benötigt, der ein UML Modell automatisch simulieren kann und die bei der Simulation überdeckten Elemente zurück liefert. Die Ausgabe des Simulators – die Modellüberdeckung – dient schließlich als Grundlage für die jeweilige Fitnessfunktion. Leider zeigt der Vergleich zweier Simulatoren, dass die bestehenden Simulationswerkzeuge die Anforderungen nur ungenügend erfüllen, die für das Projekt nötig sind.

Aus diesem Grund wurde im Rahmen dieser Arbeit ein Simulator neu entwickelt, der alle Anforderungen erfüllt. Obwohl eine Neuentwicklung einen nicht unerheblichen Aufwand bedeutet, weist eine Eigenentwicklung einige gravierende Vorteile gegenüber den kommerziellen Simulatoren auf. So kann der Simulator beliebig erweitert, leicht angepasst, getestet und gewartet werden.

Die Eigenschaften des Simulators, die für die Simulation nötigen Einschränkungen der Elemente und die möglichen Erweiterungen und Verbesserungen des Simulators sind Gegenstand des zweiten Teils dieser Arbeit.

Kapitel 2

Aufgabenstellung

Der UML-Simulator soll im Zusammenhang mit dem Projekt UnITeD (Unterstützung Inkrementeller TestDaten) verwendet werden, das versucht, den Test hochzuverlässiger und sicherheitskritischer Software zu automatisieren, um die Testkosten zu reduzieren und die Erkennung von Restfehlern in komplexer Software zu optimieren. Dazu verwendet das Projekt ein Verfahren, das durch Simulation des UML Modells und der anschließenden Bewertung der Simulationsergebnisse die Testdaten ermittelt.

Die Aufgabe dieser Arbeit ist, einen für das Projekt geeigneten Simulator auszuwählen und dafür eine Schnittstelle zu implementieren. Dazu sind zunächst Eigenschaften zu identifizieren, die ein Simulationswerkzeug bei der automatischen Simulation von UML-Modellen unter bestimmten Eingabedaten zur Messung der Modellüberdeckung benötigten. Anschließend sollen verschiedene existierende Werkzeuge anhand dieser Eigenschaften verglichen, und deren Tauglichkeit bewertet werden. Darauf aufbauend soll für ausgewählte Modelle eine Schnittstelle definiert werden, die die Anbindung eines solchen Modellsimulators an das bestehende Projekt ermöglicht.

Im Laufe der Arbeit wurde festgestellt, dass die bestehenden Simulationswerkzeuge die Anforderungen nur ungenügend erfüllen. Deshalb wurde entschieden, einen Simulator selbst zu entwickeln. Vorerst unterstützt der Simulator nur Zustandsautomaten, später kann – aufbauend auf den bestehenden Quelltext – die Unterstützung für weitere dynamische Diagramme leicht hinzugefügt werden (siehe Kapitel 6.1). Deshalb bezieht sich diese Arbeit größtenteils auf UML Zustandsautomaten.

Ein weiterer Vorteil einer Eigenentwicklung ist die Möglichkeit, den Simulator einfach anzupassen (z.B. ein anderes Verhalten), zu erweitern (z.B. um neue Diagramme), zu testen (z.B. durch JUnit Testfälle) und zu warten (Fehlerbereinigung). Außerdem kann die Entwicklung in Hinblick auf das vom bestehenden Projekt verwendete Datenmodell erfolgen, so dass keine weitere Schnittstelle benötigt wird.

2.1 Beispiel

Die Abbildung 2.1 zeigt als Beispiel einen Zustandsautomaten, der das Verhalten eines Autoradios modelliert. Das Beispiel wird auch in den späteren Kapiteln für die Beschreibung der Simulation mit dem neu entwickelten Simulator verwendet.

Da ein Zustandsautomat gemäß UML Spezifikation nicht alleine sondern nur zusammen mit einem Classifier existieren kann, ist in der Abbildung 2.2 der dazugehörende Classifier (eine Klasse) dargestellt. Die in der Klasse deklarierten Operationen können für die CallTrigger im Zustandsautomaten verwendet werden, und die deklarierten Variablen können in der Spezifikation der Guard-Bedingungen benutzt werden. Der CallTrigger einer Transition wird aktiviert, wenn die zugeordnete Operation aufgerufen wird.

Der Zustandsautomat verhält sich wie folgt: Zu Beginn befindet sich der Zustandsautomat im Startzustand (der ausgefüllte Kreis). Da die Transition zum *Aus*-Zustand keinen Trigger und keine Guard-Bedingung besitzt, wird die Transition umgehend durchlaufen und der *Aus*-Zustand betreten.

Von diesem Zustand aus existiert eine Transition zum *Historien*-Zustand, so dass der *Aus*-Zustand wieder verlassen und der *Historien*-Zustand betreten wird. Dieser befindet sich im zusammengesetzten Zustand *Betriebsmodi Radio*, so dass dieser ebenfalls durch einen Explicit Entry betreten wird. Ein Explicit Entry eines zusammengesetzten Zustands bedeutet, dass dieser indirekt durch einen in ihm enthaltenen Unterzustand betreten wird.

Da der *Historien*-Zustand das erste mal betreten wird, wird die Transition zum *Radiobetrieb*-Zustand durchlaufen und der Zustand betreten. Dort verweilt Zustandsautomat so lange, bis einer der vier Trigger ausgelöst wird.

Um eine der vier ausgehenden Transitionen zu durchlaufen, muss die mit dem Trigger assoziierte Operation aufgerufen werden und die Bedingung eines optional vorhandenen Guards wahr sein. Das heißt, dass sich der Zustandsautomat abhängig von der aufgerufenen Operation – *CD_eingelegt()* oder *Kassette_eingelegt()* – anschließend im *CD-Wechsler-Betrieb*- oder *Kassettenbetrieb*-Zustand befindet. Die beiden Operationen setzen außerdem die Variablen *CD_drin* bzw. *Kassette_drin* auf den Wert, der beim Aufruf der Operation als Parameter übergeben wurde, so dass später überprüft werden kann, ob bereits eine CD oder Kassette eingelegt wurde.

Im *CD-Wechsler-Betrieb*- oder *Kassettenbetrieb*-Zustand kann durch den Aufruf der *Radio_manuell()* Operation der Zustand verlassen und der *Radiobetrieb* Zustand wieder betreten werden. Wird anschließend die Operation *Kassette_manuell()* aufgerufen, wird die vom *Radiobetrieb*-Zustand ausgehende Transition nicht durchlaufen, weil die Auswertung der Guard-Bedingung falsch ergibt, da noch keine Kassette eingelegt wurde. Wird stattdessen die *CD_manuell()* Operation aufgerufen, wird – da bereits eine CD eingelegt wurde und die Guard-Bedingung deswegen wahr ist – die ausgehende Transition durchlaufen.

Während sich der Zustandsautomat in einem Unterzustand des zusammengesetzten Zustands befindet, kann dieser durch den Aufruf der Operation *ausschalten()* verlassen werden. Durch das Auslösen des Triggers wird die ausgehende Transition durchlaufen und der *Aus* Zustand betreten.

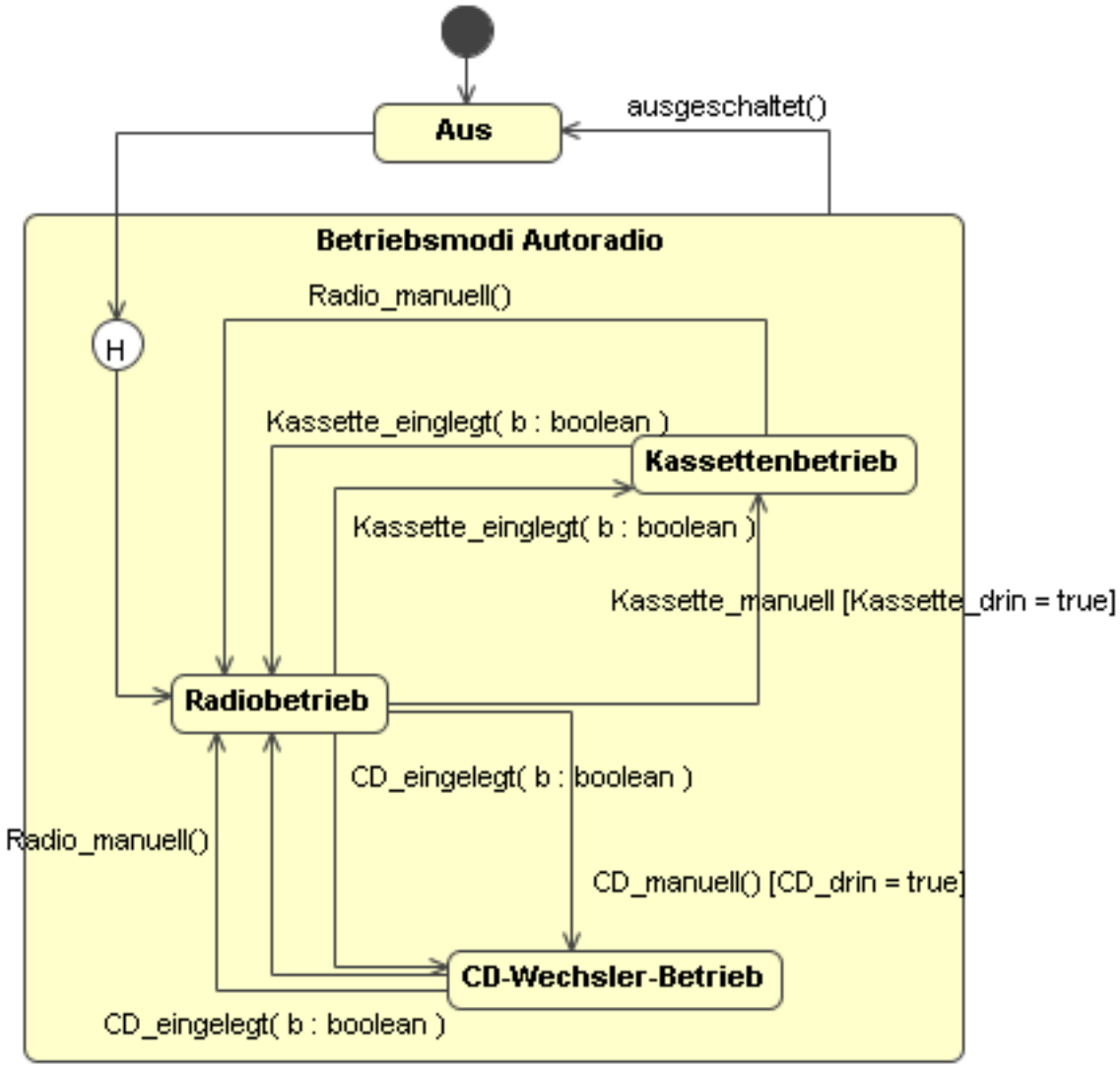

Abbildung 2.1: Beispiel eines Zustandsautomaten

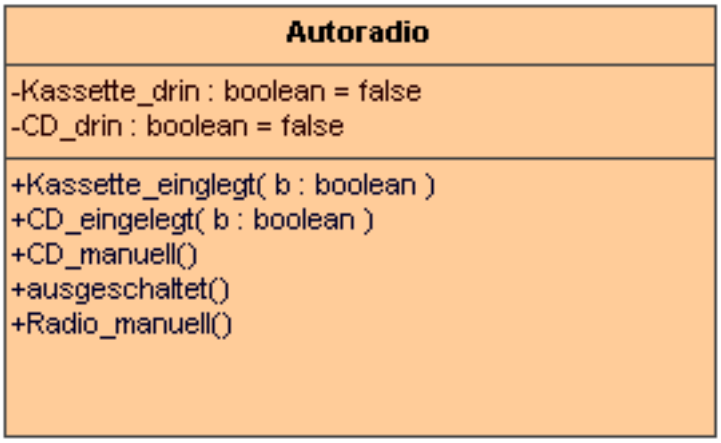

Abbildung 2.2: Klasse des Beispiel-Zustandsautomaten

Wird anschließend der zusammengesetzten Zustand wieder betreten, wird durch den Historien-Zustand der Zustand wieder aktiv, der beim Verlassen des zusammengesetzte Zustands zuletzt aktiv war. Das heißt, der Historien-Zustand „merkt“ sich den zuletzt aktiven Zustand.

Kapitel 3

Anforderungskatalog

Die auf dem Markt befindlichen Simulatoren weisen zum Teil sehr unterschiedliche Eigenschaften auf. Damit diese Simulatoren trotzdem auf ihre Tauglichkeit verglichen werden können, werden im Folgenden die Anforderungen spezifiziert, die erfüllt werden müssen bzw. erfüllt werden sollten. Simulatoren, die mindestens eine der Pflichtanforderungen nicht erfüllen, sind als Simulator für das bestehende Projekt ungeeignet und je mehr optionale Anforderungen ein Simulator erfüllt, desto besser. Leider zeigt der Vergleich zweier kommerzieller Produkte, dass die bestehenden Simulatoren nur ungenügend die Anforderungen erfüllen.

Deshalb wurde ein Simulator neu entwickelt, der im Endstadium alle genannten Anforderungen erfüllt. Welche Anforderungen der Simulator aktuell erfüllt, zeigt die dritte Spalte in Tabelle 3.2.

Die folgende Tabelle zeigt eine Übersicht über die Pflichtanforderungen und optionalen Anforderungen. In den entsprechenden Abschnitten wird auf die Erfüllung der einzelnen Anforderungen näher eingegangen.

Pflichtanforderungen	**Optionale Anforderungen**
Automatische Simulation des UML Modells	Unterstützung mehrerer dynamischer UML Modelle
Automatische Auswertung der Bedingungen und automatisches Ausführen des Verhaltens	Erkennung von Deadlocks
Standardisierte Formate	Geringe Kosten und Komplexität
Schnittstelle zum bestehenden Projekt	Plattformunabhängigkeit
Unterstützung der UML 2.0	Effizienz
Unterstützung gängiger Elemente	

Tabelle 3.1: Anforderungen an einen Simulator

3.1 Pflichtanforderungen

In diesem Abschnitt werden die Anforderungen beschrieben, die ein Simulator erfüllen **muss**, damit er für das UnITeD Projekt verwendet werden kann. Sofern ein Simulator eine der genannten Anforderungen nicht erfüllt, kann dieser nicht für das Projekt verwendet werden.

3.1.1 Automatische Simulation des UML Modells

Ein Teilprojekt des UnITeD Projekts versucht, die Modellüberdeckung eines UML-Modells zu bestimmen. Dazu wird für die dynamischen Modelle ein Simulator benötigt, der ausgehend von der Eingabe automatisch die überdeckten Elemente zurückliefert. Die Eingabe besteht dabei aus dem zu simulierenden Modell, eine Liste von Operationen und Signalen, die die Trigger der Transitionen aktivieren, einem Abbruchkriterium und evtl. einer anfänglichen Belegung der verwendeten Variablen. Als Ausgabe soll der Simulator die bei der Simulation überdeckten Elemente (z.B. besuchten Zustände und durchlaufenen Transitionen) zurückliefern. Dabei soll die Simulation so lange automatisch ablaufen, bis das Abbruchkriterium erfüllt ist.

Mögliche Abbruchkriterien sind zum Beispiel das Fehlen weiterer Ereignisse und das Auftreten eines totalen oder partiellen Deadlocks.

Ein totaler Deadlock liegt vor, wenn ausgehend von allen geraden aktiven Zuständen keine ausgehende Transition vorhanden ist, die mit den gerade aktiven Ereignissen durchlaufen werden kann. Der Zustandsautomat macht bei einem totalen Deadlock keinen Fortschritt.

Ein partieller Deadlock tritt im Gegensatz dazu nur zwischen einer bestimmten Menge von Transitionen auf, wenn Transitionen auf das Ereignis warten, dass der jeweils andere erzeugt. Bei einem partiellen Deadlock kann der Zustandsautomat Fortschritte machen, da nur eine begrenzte Menge der Transitionen vom Deadlock betroffen ist.

3.1.2 Automatische Auswertung der Bedingungen und automatisches Ausführen des Verhaltens

Eine Voraussetzung für die automatische Simulation ist die Fähigkeit des Simulators, automatisch Ausdrücke – wie sie beispielsweise in den Bedingungen der Guards vorkommen – auswerten zu können. Natürlich kann diese automatische Auswertung nur dann erfolgen, wenn die Bedingungen so spezifiziert wurden, dass sie vom Computer berechnet werden können. Aus diesem Grund scheidet die natürliche Sprache zur Spezifikation aus.

Eine Möglichkeit, die ein Simulator zur Spezifikation anbieten könnte, ist, die Bedingungen in einer formalen Sprache, ähnlich einer Programmiersprache, zu spezifizieren. Dadurch kann der Simulator mittels eines Ausdruck Parsers den Wert (wahr oder falsch) während der Simulation bestimmen und entsprechend reagieren.

Die natürliche Sprache ist für die Spezifikation des Verhaltens – zum Beispiel das einer Operation – ebenso ungeeignet, wenn die Abarbeitung des Verhaltens einen Einfluss auf die Bedin-

gungen der Guards haben soll. Deshalb muss das Verhalten von Operationen, Transitionen und Zuständen für die automatische Simulation modelliert werden können.

Die Spezifikation des Verhaltens kann zum Beispiel in Form einer Programmiersprache erfolgen. Das ARTiSAN Studio verwendet beispielsweise zur Spezifikation des Verhaltens C bzw. C++ Code. Da zur Simulation eines Zustandsautomaten das Modell nach C bzw. C++ Quelltext exportiert wird, kann der verhaltensspezifizierende Code recht einfach beim Export an den entsprechenden Stellen eingefügt werden. Dadurch wird während der Simulation kein Parser für die Auswertung der Bedingungen und das Ausführen des Verhaltens benötigt.

Weitere Einschränkungen, die die Elemente eines Zustandsautomaten betreffen, beschreibt Kapitel 4.3 genauer.

3.1.3 Standardisierte Formate

Damit ein Simulator an das bestehende Projekt durch eine Schnittstelle angebunden werden kann, müssen die vom Simulator verwendeten Formate bekannt sein. Idealerweise verwendet der Simulator standardisierte, bzw. zumindest bekannte Formate für die Ein- und Ausgabe. Proprietäre Formate scheiden aus, weil deren Spezifikation meistens nicht offen gelegt wurde, sodass mit diesen Formaten nur sehr schwer eine Schnittstelle implementiert werden kann.

Das ideale Eingabeformat für den Simulator ist XMI (XML Metadata Interchange), das von der OMG speziell für den Austausch von UML Modellen spezifiziert wurde. Weil die Spezifikation des Formats offen gelegt ist, wird dieses Format bereits von sehr vielen Werkzeugen unterstützt. Dadurch ist die Implementierung einer Schnittstelle mit diesem Format sehr einfach möglich.

Bedingt durch die unvollständige Spezifikation der XMI Version 1 hatten die Hersteller die Möglichkeit selbst zu entscheiden, wie bestimmte Teile der Spezifikation implementiert werden, so dass jeder Hersteller diese Teile anders implementierte. Deshalb muss ein Simulator die XMI Version 2 unterstützten, die diese Freiheiten nicht mehr besitzt, so dass theoretisch jeder Simulator die XMI-Dateien des anderen einlesen kann.

Als Ausgabeformat kann beispielsweise das universell einsetzbare – und ebenfalls von der OMG spezifizierte – XML Format verwendet werden, dessen Spezifikation ebenfalls offen gelegt ist und deshalb von fast jeder Programmiersprache unterstützt wird. Auch jedes andere Ausgabeformat kann verwendet werden, sofern es eine einheitliche Struktur aufweist, die automatisch eingelesen und verarbeitet werden kann, und dokumentiert ist.

Besonders bei der Ausgabe des Simulationsergebnisses ist es für das UnITeD Projekt wichtig, dass die bei der Simulation überdecken Elemente, dass heißt, die besuchen Zustände und durchlaufenen Transitionen, vollständig und fehlerfrei zurückgegeben werden.

3.1.4 Schnittstelle zum bestehenden Projekt

Unter Umständen ist es nicht ausreichend, den Simulator nur über das Ein- und Ausgabeformat anzubinden. So kann es beispielsweise nötig sein, in den Ablauf der Simulation eines Zustandsautomaten einzugreifen, um bestimmte Ereignisse zu bestimmten Zeiten zu injizieren. Das bedeutet, der Simulator muss den – möglichst direkten – Zugriff auf die Simulation ermöglichen, zum Beispiel durch eine API oder durch eine Skriptsprache. Außerdem sollte die Spezifikation der API bzw. Skriptsprache verfügbar und vollständig sein.

Bei der Neuentwicklung eines Simulators ist die Definition einer Schnittstelle überflüssig, weil das im Projekt verwendete Datenmodell verwendet werden kann. Durch die direkte Anbindung ist außerdem keine Transformation der einzelnen Datenmodelle (z.B. Datenmodell UnITeD → XMI → Datenmodell Simulator) notwendig, was wiederum Ausführungszeit spart und Fehler vermeiden kann.

3.1.5 Unterstützung der UML 2

Die Spezifikation der UML 2 ist bereits im April 2005 veröffentlicht worden und mittlerweile Standard, so dass die Simulatoren diese unterstützen müssen. Die UML 2 wurde gegenüber der UML 1.x in vielen Punkten wesentlich verbessert und erweitert. So wurden bei den Zustandsautomaten die Ein- und Austrittspunkte, die das beschreiben komplexer Zustandsautomaten erheblich erleichtern, sowie die Terminatoren eingeführt. Weiterhin wurde die Semantik einiger Diagramme (z.B. Aktivitätsdiagramm) grundlegend geändert und neue Diagramme hinzugefügt. Ein Simulator, der die neuen Elemente und die geänderte Semantik der UML 2 nicht unterstützt, kann somit nicht für das Projekt verwendet werden.

3.1.6 Unterstützung gängiger Elemente

Damit der Simulator auch sinnvoll eingesetzt werden kann, muss dieser mindestens die gängigen Elemente eines Diagramms unterstützen. Gängige Elemente eines Zustandsautomaten sind beispielsweise:

- Einfacher und zusammengesetzter Zustand
- Gängige Pseudozustände
  - Start- und Endzustand
  - Entscheidung (bedingte Verzweigung)
  - Gabelung und Vereinigung (Parallelität)
- Transitionen
  - Guards

  - Gängige Trigger
    - CallTrigger
    - SignalTrigger
    - AnyTrigger

- Verhalten
  - Operation
  - Effekt
  - Eintritts-, Zustands- und Austrittsverhalten

Gängige Elemente eines Aktivitätsdiagramms sind zum Beispiel:

- Aktions- und Objektknoten
- Kanten (bedingt und gewichtet)
- Start- und Endknoten
- Verzweigungs- und Verbindungsknoten (bedingte Verzweigung)
- Parallelisierungs- und Synchronisationsknoten (Parallelität)

3.2 Optionale Anforderungen

Die in diesem Abschnitt beschriebenen Anforderungen sind optional, dass heißt, dass ein Simulator sie nicht unbedingt erfüllen muss. Trotzdem ist es von Vorteil, wenn ein Simulator möglichst viele der genannten Anforderungen erfüllt und dadurch die Verwendung wesentlich vereinfacht.

3.2.1 Unterstützung mehrerer dynamischer Modelle

In der UML existieren mehrere dynamische Modelle – in der UML als Verhaltensdiagramme bezeichnet –, die simuliert werden können (siehe Kapitel 4.2). Deshalb wäre es praktisch, wenn der Simulator die automatische Simulation mehrerer dynamischer Modelle ermöglichen würde. Dadurch müsste die Schnittstelle nur für einen Simulator und nicht für mehrere unterschiedliche Simulatoren entwickelt werden, wenn auch andere Modelle simuliert werden sollen.

Bei der Neuentwicklung eines Simulators sollte es zudem einfach möglich sein, nachträglich die Unterstützung für andere dynamische Modelle (z.B. Aktivitätsdiagramme) hinzuzufügen. Das bedeutet, dass der Kern des Simulators so allgemein wie möglich gehalten werden sollte, sodass die Simulation eines dynamischen Modells nur durch die für ein Diagramm spezialisierten Komponenten des Simulators erfolgt.

3.2.2 Erkennung von Deadlocks

Ein wünschenswertes und sehr sinnvolles Feature ist das Erkennen von Deadlocks. Dabei können in dynamischen Modellen zwei verschiedene Arten von Deadlocks auftreten: totale und partielle.

Bei einem totalen Deadlock macht die Simulation des Zustandsautomaten keinen Fortschritt. Dass heißt, er befindet sich immer in den gleichen Zuständen, die er nicht verlassen kann. Sofern alle Zustände keine Endzustände sind, die ja keine ausgehende Transition besitzen dürfen, befindet sich der Zustandsautomat in einem totalen Deadlock. Ein totaler Deadlock kann auftreten, wenn keine Transition mehr durchlaufen werden kann, weil kein Ereignis für einen Trigger vorhanden ist, oder wenn keine Bedingung eines Guards wahr ist. Eine solche Situation kann auftreten, wenn die Transitionen des zu simulierende Zustandsautomaten Trigger besitzen und zu Beginn der Simulation die dafür notwendigen Ereignisse nicht mit übergeben wurden. Ein totaler Deadlock kann im Gegensatz zu einem partiellen Deadlock sehr einfach erkannt werden, indem nach jedem Simulationsschritt überprüft wird, ob mindestens eine Transition durchlaufen wurde.

Ein partieller Deadlock betrifft nicht den ganzen Zustandsautomaten, sondern nur einen bestimmten Teil davon. So kann ein partieller Deadlock auftreten, wenn zwei Transitionen auf ein Signal warten, das der jeweils andere erzeugt. Auch wenn die Guard-Bedingung zweier Transitionen nur durch das Verhalten der jeweils anderen Transition beeinflusst wird, tritt ein partieller Deadlock auf, da das Verhalten einer Transition erst ausgeführt wird, nachdem diese durchlaufen wurde. Partielle Deadlocks weisen normalerweise auf eine fehlerhaftes Modell hin, sodass eine frühzeitige Erkennung vorteilhaft ist.

Das Erkennen eines partiellen Deadlocks ist relativ schwierig, da nur ein Teil des Zustandsautomaten betroffen ist, so dass dieser weiteren Fortschritt machen kann. Ein aus den Betriebssystemen bekanntes Verfahren zur Erkennung von Deadlocks ist das Suchen nach einen Zyklus im Wartegraphen. Das Verfahren könnte auch in etwas abgewandelter Form zur Erkennung partieller Deadlocks in einem Zustandsautomaten verwendet werden.

Bei einem Deadlock sollte der Simulator die Simulation anhalten und den Grund für den Abbruch mitteilen, so dass das UnITeD Werkzeug entscheiden kann, ob die Simulation mit anderen Eingabedaten wieder aufgenommen oder abgebrochen werden soll.

3.2.3 Geringe Kosten und Komplexität

Kommerzielle Produkte im UML Sektor sind verhältnismäßig teuer, besonders wenn sie spezielle Funktionalität wie die Simulation eines dynamischen UML Modells bieten. Außerdem sind die kommerziellen Produkte keine einfachen Werkzeuge mehr sondern überwiegend ganze Programmsammlungen, die fast alle in der UML spezifizieren Diagramme und Elemente unterstützen. Obwohl eine durchgängige Unterstützung durch ein Werkzeug vorteilhaft ist, hat die gebotene Funktionsvielfalt natürlich seinen Preis. Auch die Bedienbarkeit leidet bei den meisten kommerziellen Produkten an deren Komplexität.

3.2.4 Plattformunabhängigkeit

Der Simulator sollte für verschiedene Systeme verfügbar sein, so dass er universell eingesetzt werden kann und nicht auf eine bestimme Plattform beschränkt ist.

Bei der Neuentwicklung eines Simulators hat man die Wahl zwischen verschiedenen plattformunabhängigen Programmiersprachen. So kann bereits bei der Entwicklung auf Plattformunabhängigkeit geachtet werden – zum Beispiel durch das Verwenden der Programmiersprache Java. Für diese gibt es eine Vielzahl von Interpretern für unterschiedliche Betriebssysteme.

3.2.5 Effizienz

Das UnITeD Projekt verwendet ein Verfahren, das sehr hohe Ansprüche an die Effizienz des Simulators stellt. Bei diesem Verfahren wird der Simulator sehr häufig und mit unterschiedlichen Eingaben parallel ausgeführt, so dass dieser mit den vorhandenen Ressourcen schonend umgehen sollte. Da vor allem in kommerziellen Produkten die Simulatoren nicht eigenständig sind, sondern nur ein Teil einer komplexen Werkzeugsammlung, schneiden diese bei der Betrachtung der Effizienz meistens schlechter ab als eigenständige Simulatoren, die auf das Simulieren von Modellen spezialisiert sind.

Im Gegensatz zu den meisten kommerziellen Simulatoren ist es bei einer Eigenentwicklung relativ einfach möglich, bei der Entwicklung auf effiziente Programmierung zu achten oder bereits vorhandene Funktionalität zu optimieren.

3.3 Vergleich der Simulatoren

Um die Leistungsfähigkeit der auf dem Markt befindlichen Simulatoren zu überprüfen, wurden zwei bekannte kommerzielle Produkte herangezogen und deren Eigenschaften mit den Anforderungen verglichen. Bei den beiden Produkten handelt es sich um das ARTiSAN Studio von ARTiSAN und das TAU G2 von Telelogic. In der Tabelle 3.2 sind zum direkten Vergleich die Eigenschaften der beiden kommerziellen Simulatoren und des selbst entwickelten Simulators angegeben. Zusätzlich zu den beiden Produkten wurde noch der kostenlose und quelloffene Simulator JavaFSM der Universität Hamburg untersucht. Dieser unterstützt aber nur die Simulation von Zustandsautomaten, so dass dieser nicht für das Projekt geeignet ist.

3.3.1 ARTiSAN Studio

Die Produkt ARTiSAN Studio der Firma ARTiSAN unterstützt den UML 2.0 und SysML-Standard, so dass es für das Software- und Systems-Engineering verwendet werden kann. Laut ARTiSAN ist es das marktführende Werkzeug für das auf Standards basierte Systems-Engineering mit einer beispiellosen Unterstützung für OMG UML und OMG SysML. Zusätzlich zur UML-

und SysML-Modellierung bietet das ARTiSAN Studio einen Simulator zur Simulation von Zustandsautomaten und Sequenzdiagrammen.

Im grafischen Editor des ARTiSAN Studios kann der zu simulierende Zustandsautomat und die dazu gehörende Klasse komfortabel durch Einfügen der Elemente und Anpassung der Eigenschaften modelliert werden. Die in der Klasse definierten Variablen und Operationen können anschließend in den Guard-Bedingungen und in den Verhaltens-Spezifikationen verwendet werden. Zur Spezifikation der Guard-Bedingungen und der Operationen verwendet das ARTiSAN Studio C bzw. C++ Quelltext, so dass die Bedingungen automatisch ausgewertet und das Verhalten einer Operation automatisch ausgeführt werden kann.

Das ARTiSAN Studio erlaubt bei der Spezifikation sogar die Verwendung der *return*-Anweisung, so dass einer Operation nicht nur Werte in Form von Parametern übergeben werden können, sondern auch Werte nach der Abarbeitung des Verhaltens zurück liefern kann. Dadurch ist es beispielsweise möglich, eine Operation innerhalb einer Guard-Bedingung aufzurufen und den Rückgabewert anschließend in der Bedingung zu verwenden.

Das ARTiSAN Studio unterstützt bei der Modellierung und Simulation von Zustandsautomaten die folgenden Trigger:

- CallTrigger
- SignalTrigger
- TimeTrigger
- ChangeTrigger
- EntryTrigger und ExitTrigger – Code, der beim Betreten bzw. Verlassen eines Zustands ausgeführt werden soll
- CreateTrigger und DestroyTrigger – Code, der bei der Erzeugung bzw. Zerstörung der Klasse ausgeführt werden soll

Damit der Zustandsautomat simuliert werden kann, muss eine so genannte Test Harness erstellt werden, die für die Simulation durch den ARTiSAN Simulator benötigt wird. Dazu wird der Zustandsautomat als C bzw. C++ Quelltext exportiert und daraus unter Zuhilfenahme des Visual Studio Compilers von Microsoft eine dynamische Bibliothek erzeugt (DLL). Beim Export des Modells werden die Stubs mit den in den Guards und Operationen angegeben Quellcode ergänzt.

Anschließend kann der Zustandsautomat durch den Simulator simuliert werden. Dazu bietet das ARTiSAN Studio die Wahl zwischen der schrittweisen manuellen und automatischen Simulation an. Während der Simulation können Ereignisse injiziert und Operationen aufgerufen werden, denen aktuelle Parameter mit übergeben werden können.

Der Verlauf der Simulation wird in einem Bereich des Simulators grafisch angezeigt und kann nach dem Beenden in einer Datei gespeichert werden. In Tabelle 3.2 werden die Eigenschaften des ARTiSAN Studios mit den anderen Simulatoren verglichen.

Weiterhin können im ARTiSAN Studio durch die Integration von Editor und Simulator sehr viele Aktionen durch intuitives Drag & Drop erfolgen, so dass trotz der Komplexität ein gewisses Maß an Bedienkomfort vorhanden ist. Das ARTiSAN Studio bietet außerdem eine sehr gute Unterstützung der Teamarbeit durch ein gemeinsames Modell-Repository und die Unterstützung für das simultane Round-Trip Engineering, eine sehr gute Integration in die Visual Studio Entwicklungsumgebung von Microsoft, Erweiterungen für die Modellierung von Realzeitsystemen und die Simulation einer Benutzerschnittstelle mit dem Zusatzprodukt Altia FacePlate.

Eigenschaft	**ARTiSAN Studio**	**TAU G2**	**Eigenentwicklung**
Automatische Simulation des Modells	✓	✓	✓
Automatische Auswertung der Bedingungen und automatisches Ausführen des Verhaltens	✓	✓	✓
Standardisierte Formate	✓ (XMI)	✓ (XMI)	✓ (UML 2 EMF)
Schnittstelle zu bestehendem Projekt	✓ (VBScript)	✓ (Plugin)	✓ (UML 2 EMF)
Unterstützung der UML 2	✓ (2.0)	✓ (2.0)	✓ (2.1)
Unterstützung aller gängigen Elemente eines Diagramms	✓	× (keine Parallelität, Hierarchie und Operationen)	✓
Anzahl unterstützter dynamischer UML Modelle	2	2	1 (aktuell)
Erkennung von Deadlocks	✓ (total)	✓ (partiell und total)	✓ (total)
Geringe Kosten und Komplexität	×	×	✓
Plattformunabhängigkeit	× (nur Windows)	✓ (Windows, Solaris, Linux)	✓ (Java)
Effizient	×	×	✓ (leicht optimierbar)

Tabelle 3.2: Eigenschaften der Simulatoren

3.3.2 TAU G2

Das Produkt TAU G2 bietet Unterstützung für das Protocol- und Systems Engineering. Laut Telelogic, dem Unternehmen, das TAU G2 entwickelt hat, bietet es „[...] anspruchsvolle Funk-

tionalitäten für jede Phase des Lebenszyklus [...], vom Systems Engineering bis hin zur Testgenerierung.“ [5]. Zusätzlich bietet TAU G2 einen integrierten Simulator für Protokollzustandsautomaten und Sequenzdiagramme.

In TAU G2 können die statischen und dynamischen Modelle durch den komfortablen grafischen Editor modelliert werden. Zur Modellierung eines Protokollzustandsautomaten unterstützt das TAU G2 zwei Ansichten: die zustandsorientierte Sicht, die eine sehr gute Übersicht über das ganze System bietet, Details aber ausblendet – ähnlich dem Kommunikationsdiagramm –, und eine transitionsortientierte Sicht, die die Zustände und Transitionen detailliert in einer SDL-ähnlichen Form darstellt.

Bei der Modellierung eines Protokollzustandsautomaten werden Variablen – im Gegensatz zu anderen Modellierungswerkzeugen – nicht in der Klasse als Attribut, sondern – wie in der SDL üblich – durch einen Kommentar im Diagramm deklariert. Die Variablen können anschließend in den Guard-Bedingungen verwendet, oder den Signalen als Payload mitgegeben werden, wobei bereits während der Eingabe die Syntax und die Typkompatibilität überprüft wird. Des Weiteren dürfen die Bedingungen der Guards nur aus einfachen Ausdrücken ohne Seiteneffekte bestehen, die entweder den Wert wahr oder falsch zurück liefern.

Da in TAU G2 nur Protokollzustandsautomaten modelliert und simuliert werden können, werden zur Interaktion nur das Senden und Empfangen von Signalen durch SignalTrigger unterstützt. Beim Hinzufügen eines Signals in die Signalwarteschlange können dem Signal aktuelle Parameter mit übergeben werden, die vom Empfänger weiter verwendet werden können – zum Beispiel innerhalb einer Guard-Bedingung. Für die Modellierung von Timeouts, die für das Protocol Engineering von besonderer Bedeutung sind, bietet das TAU G2 außerdem die Unterstützung der TimeTrigger.

Damit ein Protokollzustandsautomat simuliert werden kann, wird eine so genannte Configuration erstellt. Dazu benötigt das TAU G2 einen C++ Compiler – entweder den Microsoft Visual C++ Compiler oder den GCC – um den Quelltext zu übersetzen, der aus dem Zustandsautomaten erzeugt wird. Anschließend kann die erstellte Configuration durch den Benutzer schrittweise oder bis zu vorher festgelegten Breakpoints automatisch simuliert werden.

Der Simulationsfortschritt wird am Bildschirm ausgegeben; es kann aber eingestellt werden, dass der Ablauf zusätzlich textuell in einer Datei gespeichert werden soll. Weiterhin kann beim TAU G2 während der Simulation das automatisch erstellte Sequenzdiagramm angezeigt werden, dass die zeitliche Reihenfolge der ausgetauschten Nachrichten und die Interaktionspartner übersichtlich darstellt. Außerdem kann das TAU G2 nach der Simulation weitere Informationen über die überdeckten Elemente ausgeben.

Weitere nützliche Features sind das Round-Trip Engineering, und die automatische Ausführung und das Verwalten von Testfällen mittels der TTCN-3 (Testing and Test Control Notation). TAU G2 bietet außerdem eine sehr gute Integration in das Microsoft Visual Studio .NET 2003 und Eclipse. Außerdem ist das Produkt für unterschiedliche Betriebssysteme verfügbar.

Da das TAU G2 nur die Modellierung und Simulation von Protokollzustandsautomaten unterstützt, existieren in TAU G2 keine Elemente, um parallele Abläufe oder Zustands-Hierarchien zu modellieren. Eine Pflichtanforderung ist aber die Unterstützung solcher Elemente, so dass das

TAU G2 nicht als Simulator für das Projekt verwendet werden kann.

In Tabelle 3.2 werden die Eigenschaften des Produkts mit den anderen Simulatoren verglichen.

Anmerkung zur Vergleichstabelle TAU G2 kann nur durch ein Add-in XMI Dateien einlesen.

3.3.3 Eigenentwicklung

Der im Rahmen dieser Arbeit in Java entwickelte Simulator besitzt selbst keinen Editor um die Zustandsautomaten modellieren zu können, sondern benutzt als Eingabe ein Modell des EMF UML2 Frameworks. Er unterstützt sowohl die manuelle schrittweise als auch die automatische Simulation, bis ein vorher festgelegtes Ereignis eintritt. Aktuell wird nur die Simulation eines Zustandsautomaten unterstützt, die Unterstützung für weitere Diagramme kann aber einfach hinzugefügt werden. Weiterhin werden bei der Simulation aktuell CallTrigger, SignalTrigger und AnyTrigger unterstützt.

Die verwendeten Variablen müssen in der Klasse deklariert worden sein, für die der Zustandsautomat das Verhalten darstellt, damit diese Variablen anschließend in den Bedingungen der Guards verwendet werden können. Außerdem können diese Variablen durch den Aufruf einer Operation geändert werden. Dazu werden mathematische Ausdrücke verwendet, die bei der Modellierung als Einschränkung an die Operation angehängt werden, und die beim Aufruf einer Operation durch einen Ausdruck Parser ausgewertet werden. Beim Aufruf können einer Operation außerdem aktuelle Parameter mit übergeben werden.

Für die Simulation eines Zustandsautomaten benötigt der Simulator keine externen Werkzeuge, wie das beim ARTiSAN Studio und dem TAU G2 der Fall ist. Außerdem kann die Simulation automatisch bis zu einem vorher festgelegten Ereignis, oder schrittweise manuell durch den Benutzer erfolgen.

Nach der Simulation können die besuchten Zustände und durchlaufenen Transitionen durch die entsprechenden Methoden abgefragt werden. In Tabelle 3.2 werden die Eigenschaften der Eigenentwicklung mit den anderen Simulatoren verglichen.

Eine genaue Beschreibung des Simulators erfolgt im Kapitel 5.

3.3.4 JavaFSM

JavaFSM ist ein Java-Programm, das im Rahmen einer Studienarbeit an der Universität Hamburg entwickelt wurde und zur Animation von Mealy- und Moore-Automaten dient. Weil es keine UML Zustandsautomaten unterstützt – und somit nicht für das Projekt relevant ist – wird es hier nur ansatzweise beschrieben. Für weitere Informationen verweise ich auf [10].

Der zugrunde liegende Automat kann mit Hilfe eines Editors entworfen werden, indem Zustände hinzugefügt und durch Transitionen verbunden werden. Den Zuständen können Ausgabe-

werte zugeordnet werden, die beim Betreten eines Zustand durch einen Ausgabekanal ausgegeben werden. Weiterhin kann für die Transitionen eine Übergangsbedingung festgelegt werden, die wahr sein muss, damit die Transition durchlaufen wird. Eine solche Bedingung besteht aus einer Verknüpfung der Eingabekanäle durch die logischen Operatoren NICHT, UND und ODER.

Die Simulation erfolgt schrittweise durch den Benutzer und wird grafisch auf dem Bildschirm dargestellt. Der Benutzer kann vor einem Schritt die Eingabedaten ändern um Zustandsübergänge zu ermöglichen. Nach einem Schritt kann das Impulsdiagramm angezeigt werden, dass die Eingabe, Ausgabe und den aktuellen Zustand übersichtlich und gut nachvollziehbar in einem Diagramm darstellt.

Weitere nützliche Features sind der Export des Zustandsautomaten nach VHDL und KISS, sowie die Möglichkeit, die Erreichbarkeit vom Simulator automatisch berechnen zu lassen.

Kapitel 4

Simulation von Modellen

In der UML 2 wurden zwei Arten von Diagramme spezifiziert: statische und dynamische. Statische Diagramme werden in der UML 2 als Strukturdiagramme bezeichnet und modellieren die Statik des Systems. Zu dieser Diagrammart gehört das Klassendiagramm, das Paketdiagramm, das Objektdiagramm, das Kompositionsstrukturdiagramm, das Komponentendiagramm und das Verteilungsdiagramm.

Die dynamischen Diagramme heißen in der UML 2 Verhaltensdiagramme und beschreiben das dynamische Verhalten eines Modellelements. Zu den Verhaltensdiagrammen zählt das Use-Case-Diagramm, das Aktivitätsdiagramm, der Zustandsautomat, das Sequenzdiagramm, das Kommunikationsdiagramm, das Timingdiagramm und das Interaktionsübersichtsdiagramm. Aufgrund der Dynamik der Verhaltensdiagramme eignen sie sich bestens für eine Simulation.

In diesem Kapitel werden zuerst die Eigenschaften der verschiedenen Verhaltensdiagramme kurz beschrieben (vgl. [2]) und auf Probleme bei der automatischen Simulation kurz eingegangen. Anschließend werden die Einschränkungen dargestellt, die die Elemente eines Zustandsautomaten bei der automatischen Simulation erfahren.

4.1 Vor und Nachteile einer Simulation

Unabhängig vom UnITeD Projekt hat die Simulation eines UML-Modells weitere Vorteile:

Ein wesentlicher Vorteil ist die frühzeitige Erkennung von Fehlern. Da die Kosten für die Behebung von Fehlern relativ zur Entstehungsphase sind ist es von Vorteil, das UML-Modell so früh wie möglich zu simulieren. Leider kann vor allem in den frühen Entwicklungsphasen weniger simuliert werden, da weniger Informationen zur Verfügung stehen als in späteren Entwicklungsphasen. Schlimmstenfalls sind zu wenig Informationen vorhanden, so dass eine Simulation unmöglich ist.

Ein weiterer Vorteil ist, dass die automatische Simulation eines Modells wesentlich genauer und nicht so fehleranfällig ist wie das manuelle Überprüfen durch den Entwickler.

Natürlich bringt die automatische Simulation von UML-Modellen auch Nachteile mit sich. So muss für eine automatische Simulation die Mächtigkeit der UML 2.0 eingeschränkt werden. In Kapitel 4.3 werden die Einschränkungen beschrieben, die bei der Simulation eines Zustandsautomaten gemacht werden müssen. Diese Einschränken erleichtern die Simulation bzw. machen diese erst möglich. So kann zum Beispiel das Verhalten nicht simuliert werden, wenn es in natürlicher Sprache spezifiziert wurde. Die gleiche Einschränkung trifft auch auf die Bedingungen und weitere Elemente zu.

Ein weiterer Nachteil einer automatischen Simulation ist, dass nur Modelle der unterstützten UML-Version simuliert werden können. Obwohl die UML 2 bereits seit April 2005 von der OMG offiziell verabschiedet wurde, existiert noch eine Vielzahl von Werkzeugen, die nur die veraltete UML Version 1.x unterstützen (vgl. [7]). Da langfristig die UML 2 ältere Version verdrängen wird, wurde die Unterstützung für die UML 2 als Pflichtanforderung für den Vergleich ebenfalls mit aufgenommen.

Des Weiteren erfordern vor allem kommerzielle Werkzeuge eine gewisse Einarbeitungszeit. Die Werkzeuge zur UML Modellierung sind mittlerweile so mächtig und komplex, so dass damit nur nach einer gewissen Einarbeitungszeit produktiv gearbeitet werden kann.

Der letzte Nachteil einer automatischen Simulation ist, dass dabei kein Nichtdeterminismus möglich ist, den die UML Spezifikation grundsätzlich erlaubt. Echter Nichtdeterminismus tritt beispielsweise bei einem Zustandsautomaten auf, wenn zwei ausgehende Transitionen dieselbe Bedingung als Guard besitzen. Der Simulator müsste dann nichtdeterministisch entscheiden, welche der beiden Transitionen er wählt. Da Computer aber deterministisch arbeiten, ist echter Nichtdeterminismus unmöglich.

Obwohl in dieser Aufzählung die Anzahl der Nachteile überwiegen bedeutet das nicht, dass eine automatische Simulation eines Modell grundsätzlich schlecht ist. Vielmehr sind einige der hier genannten Nachteile nicht so gravierend wenn man bedenkt, welche Vorteile die Simulation eines sehr großen Modells – im Vergleich zum dafür nötigen Aufwand – mit sich bringt.

4.2 UML 2 Verhaltensdiagramme

4.2.1 Use-Case-Diagramm

Das Use-Case-Diagramm (deutsch: Anwendungsfall-Diagramm) wird verwendet um darzustellen, was das System für seine Umwelt leistet. Dazu stellt das Diagramm die Außenansicht auf das System auf sehr hohem Abstraktionsniveau durch Use-Cases und deren Beziehungen zu Nutzern – in UML Akteure genannt – und anderen Use-Cases dar. Ein Nutzer muss dabei nicht eine Person, sondern kann auch ein anderes System sein.

Die Simulation eines Use-Case Diagramms ist nicht sinnvoll, da es nur die Funktionalität des Systems beschreibt und deshalb eher statischer Natur ist. Sinnvoll dagegen ist die Simulation der dynamischen Modelle, die das Verhalten eines Use-Cases beschreiben, beispielsweise ein Zustandsautomat oder ein Aktivitätsdiagramm.

Die Abbildung 4.1 zeigt das nach außen sichtbare Verhalten eines Online-Banking Systems mittels eines Use-Case Diagramms.

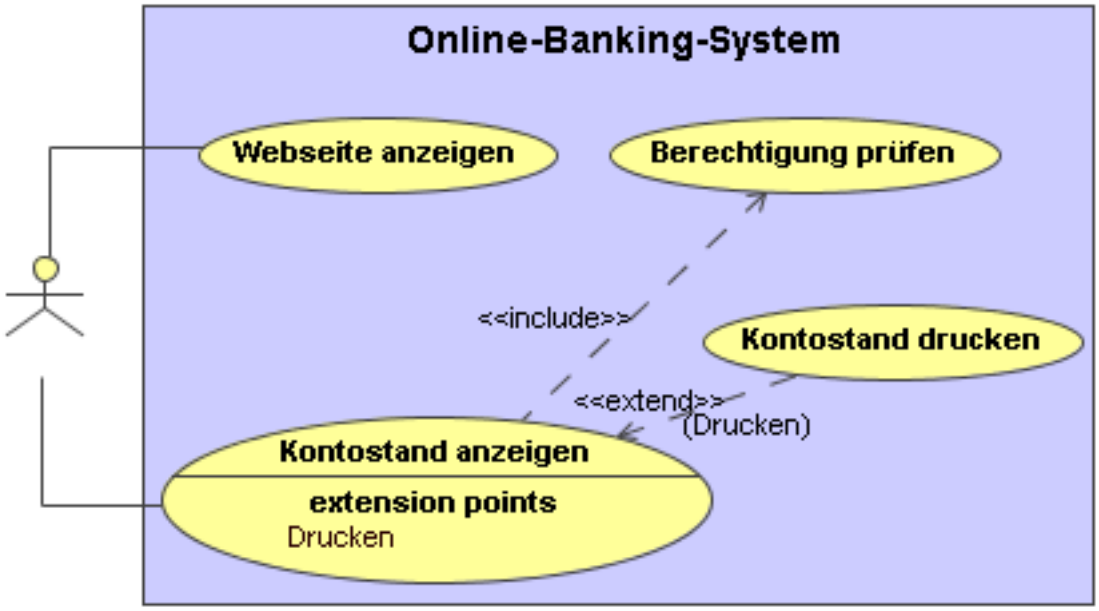

Abbildung 4.1: Beispiel für ein Use-Case-Diagramm, Quelle: [2]

4.2.2 Aktivitätsdiagramm

Durch ein Aktivitätsdiagramm werden in der UML Abläufe modelliert. Ein Ablauf kann zum Beispiel die Abarbeitung einer Operation oder das Verhalten eines Use-Cases sein. Auch ein kompletter Geschäftsvorfall kann mit dieser Diagrammart visualisiert werden. Dazu existieren in den Aktivitätsdiagrammen Elemente, mit denen der Ablauf durch Bedingungen gezielt gesteuert und Nebenläufigkeit modelliert werden kann. Allgemein zeigen Aktivitätsdiagramme wie das System ein bestimmtes Verhalten realisiert. Abbildung 4.2 zeigt beispielsweise den Ablauf, wie Spaghetti richtig gekocht werden.

Aktivitätsdiagramme sind in der UML 2 keine Sonderform der Zustandsautomaten mehr – wie es in der UML 1.x der Fall war –, sondern basieren jetzt auf den erweiterten Petri-Netzen, inklusive dem damit verbundenen Token-Konzept. Deshalb können Aktivitätsdiagramme sehr leicht simuliert werden. Für die Simulation von (erweiterten) Petri-Netzen existiert bereits eine Vielzahl von zum Teil kostenlosen und quelloffenen Simulatoren.

In Kapitel 6.1 wird beschrieben, wie der neu entwickelte Simulator angepasst werden muss bzw. welche Teile wieder verwendet werden können, damit er ebenfalls Aktivitätsdiagramme simulieren kann.

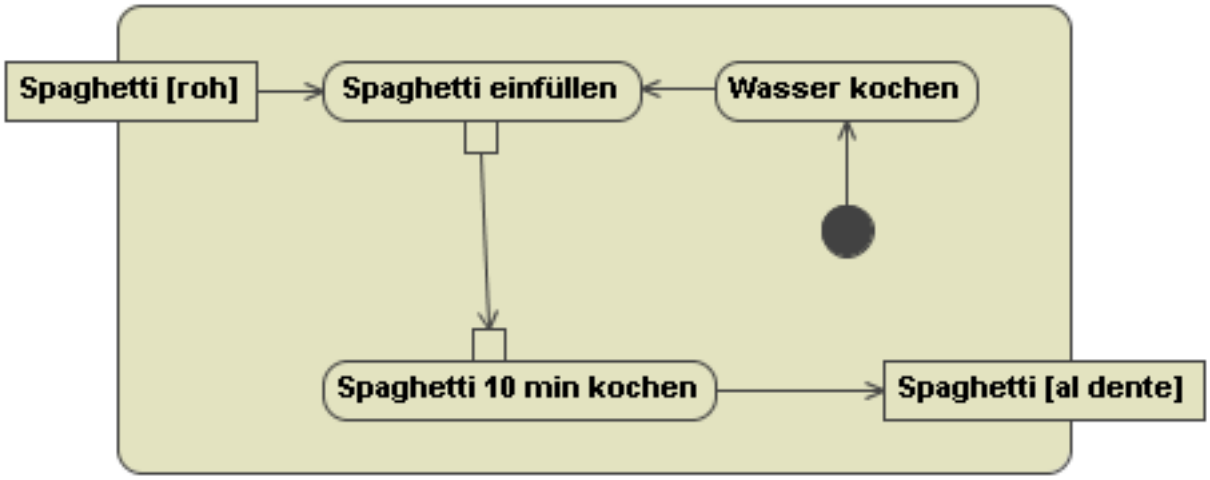

Abbildung 4.2: Beispiel für ein Aktivitätsdiagramm, Quelle: Wikipedia

4.2.3 Zustandsautomat

Mit einem Zustandsautomaten bietet die UML eine weitere Möglichkeit, das Verhalten von Modellelementen zu modellieren. Modelliert wird das Verhalten durch Zustände, die ein Modellelement einnehmen kann, und Übergänge zwischen diesen Zuständen. Ein solcher Zustandsübergang wird dabei durch externe oder interne Ereignisse ausgelöst und kann zusätzlich mit einer Bedingung versehen werden, die erfüllt sein muss, damit der Zustandsübergang auch erfolgt. Allgemein zeigt ein Zustandsautomat wie das System in einem bestimmten Zustand auf gewisse Ereignisse reagiert. Angelehnt ist die Semantik der Zustandsautomaten an endliche Automaten.

In Kapitel 4.3 werden die Einschränkungen, die die Elemente eines Zustandsautomaten für die automatische Simulation erfahren müssen, näher beschrieben. In Kapitel 5 wird schließlich der selbst entwickelte Simulator vorgestellt, der eine automatische Simulation eines Zustandsautomaten ermöglicht.

4.2.4 Interaktionsdiagramme

Interaktionsdiagramme zeigen das Zusammenspiel von mehreren Kommunikationspartnern, wobei jedes Interaktionsdiagramm einen anderen Einsatzschwerpunkt hat. Grundkonzepte der Interaktionsdiagramme sind die Interaktionen (der Austauschen der Nachrichten), Lebenslinien, Nachrichten und die miteinander kommunizierenden Partner, wobei bei einigen Diagrammen einige der Konzepte nicht existieren bzw. deren Anwendung keinen Sinn machen.

Wie die beiden in Kapitel 3.3 vorgestellten Werkzeuge demonstrieren, ist die Simulation der Sequenzdiagramme – und damit auch der anderen Interaktionsdiagramme – problemlos möglich.

Sequenzdiagramm

Ein Sequenzdiagramm zeigt den Austausch von Nachrichten zwischen Kommunikationspartnern innerhalb eines Systems oder über Systemgrenzen hinweg. Dabei wird im Gegensatz zum Kommunikationsdiagramm die zeitliche Reihenfolge der Nachrichten in den Vordergrund gestellt. Die Sequenzdiagramms sind leicht abgewandelte Message Sequence Charts, die sehr häufig in der Telekommunikation zur Modellierung von Protokollen verwendet werden. Abbildung 4.3 zeigt beispielsweise den Nachrichtenaustausch beim 3-Wege Handshake des TCP-Protokolls anhand eines Sequenzdiagramms.

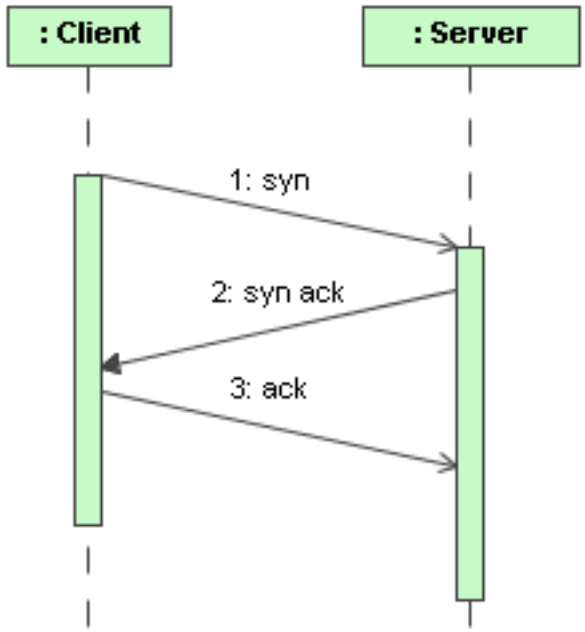

Abbildung 4.3: Beispiel für ein Sequenzdiagramm, Quelle: Wikipedia

Kommunikationsdiagramm

Das Kommunikationsdiagramm ähnelt sehr dem Sequenzdiagramm, nur liegt der Fokus auf den interagierenden Partnern und nicht auf der zeitliche Reihenfolge der Nachrichten. Verwendet wird das Diagramm um zu zeigen, wie die Kommunikationspartner kooperieren, um eine Aufgabe gemeinsam zu lösen.

Timingdiagramm

Mit einem Timing-Diagramm lässt sich das zeitliche Verhalten von Modellelementen in einem System darstellen. Dieses Diagramm wird schon seit langem in der Elektrotechnik verwendet, um das zeitliche Verhalten digitaler Schaltungen zu visualisieren.

Interaktionsübersichtsdiagramm

Ein Interaktionsübersichtsdiagramm zeigt, in welcher Reihenfolge und unter welchen Bedingungen Interaktionen stattfinden. Ein solches Diagramm ähnelt sehr einem Aktivitätsdiagramm, nur werden anstatt Aktivitäten und Objektknoten die Interaktionen bzw. Interaktionsreferenzen verwendet.

4.3 Elemente von Zustandsautomaten und deren Einschränkungen

Ein Zustandsautomat spezifiziert das Verhalten durch Zustände, die ein Modellelement (ein Classifier) einnehmen kann, und Übergänge zwischen den Zuständen. Die UML Zustandsautomaten ähneln sehr stark den erweiterten endlichen Automaten, es wurden aber zusätzlich weitere Elemente hinzugefügt, um komplexe Systeme übersichtlich und lückenlos modellieren zu können. So ist es im Gegensatz zu den endlichen Automaten möglich, das Verhalten eines Zustandsautomaten in immer kleinere und einfachere Teile zu zerlegen und so eine Hierarchie aufzubauen. Diese Hierarchie erleichtert sowohl die Entwicklung durch kleinere abgeschlossene Code-Einheiten, als auch den Test durch kleinere zu testende Einheiten. Außerdem ist es bei UML Zustandsautomaten möglich, parallele Abläufe zu modellieren, wodurch verteilte Systeme relativ einfach beschrieben werden können.

Damit das Verhalten eines Modellelements durch einen Zustandsautomaten beschrieben werden kann, müssen gemäß ([2], S. 336) folgende vereinfachende Annahmen gelten:

- Der Zustandsautomat befindet sich zu einem bestimmten Zeitpunkt in genau einem Zustand.
- Der Übergang von einem Zustand in den nächsten erfolgt ohne zeitliche Verzögerung.

Im Folgenden werden die einzelnen Elemente eines Zustandsautomaten, und die gegebenenfalls auftreten Einschränkungen dieser Elemente bei einer automatischen Simulation, näher beschrieben (vgl. [2]).

4.3.1 Start- und Endzustand

Der Startzustand ist der Zustand, der beim Betreten eines Zustandsautomaten als erstes aktiv ist. Ausgehend von diesem Zustand wird durch höchstens eine ausgehende ungetriggerte Transition der erste „echte“ Zustand des Zustandsautomaten erreicht. Weiterhin darf ein Startzustand gemäß UML Spezifikation keine eingehenden Transitionen besitzen und nur einmal pro Region vorhanden sein. Damit ein Startzustand automatisch simuliert werden kann, muss die Verwendung wie folgt eingeschränkt werden.

Der Startzustand muss konform zur UML Spezifikation – wie im vorherigen Absatz beschrieben – verwendet werden. Weiterhin darf die einzige ausgehende Transition keine Trigger und keine Guard-Bedingung besitzen, so dass sichergestellt ist, dass der Zustandsautomat sich umgehend nach dem Eintritt in eine Region in einem „echten“ Zustand befindet. Die ausgehende Transition darf aber ein Verhalten in Form eines Effekts besitzen, das beim Durchlaufen der Transition ausgeführt wird. Dieses Verhalten kann beispielsweise dazu verwendet werden, die in den Bedingungen verwendeten Variablen zu initialisieren, so dass die Bedingungen einen sinnvollen Startwert haben.

Der Endzustand ist der Zustand, in dem kein weiteres Verhalten mehr in der Region ausgeführt wird. Deshalb darf ein Endzustand keine ausgehenden Transitionen besitzen. Weiterhin gilt, dass eine Region mehrere Endzustände besitzen darf.

Bei der Simulation ist darauf zu achten, dass das Betreten eines Endzustands innerhalb eines zusammengesetzten Zustands zum Verlassen des zusammengesetzten Zustands über eine ausgehende ungetriggerte Transition führt. Bei orthogonal zusammengesetzten Zuständen müssen außerdem alle Endzustände der Regionen aktiv sein, damit dieser durch die ausgehende ungetriggerte Transition verlassen werden darf (UND-Verknüpfung).

Ein Terminator ist dem Endzustand sehr ähnlich, nur endet beim Betreten eines Terminators die Lebensdauer des beschriebenen Modellelements, so dass kein weiteres Verhalten mehr ausgeführt wird und die Simulation umgehend beendet werden kann.

4.3.2 Einfacher Zustand

Ein Zustand modelliert eine Situation, in deren Verlauf eine invariante Bedingung gilt, und die durch Transitionen betreten und verlassen werden kann. In der UML existieren neben den einfachen Zuständen auch zusammengesetzte Zustände, die mehrere Regionen besitzen können. Diese Zustände ermöglich die Modellierung einer Hierarchie und von Parallelität.

Einfache und zusammengesetzte Zustände können ein Verhalten besitzen, dass jeweils beim Betreten, während des Aufenthalts und beim Verlassen des Zustand ausgeführt wird. Die UML bezeichnet diese Verhalten als Eintritts-, Zustands- und Austrittsverhalten. Dabei gilt bei der Ausführung die folgende zeitliche Reihenfolge: Eintrittsverhalten → Zustandsverhalten → Austrittsverhalten.

Das Eintrittsverhalten wird beim Betreten des Zustands ausgeführt. Wenn das Verhalten vollständig abgearbeitet wurde, wird umgehend das Zustandsverhalten ausgeführt. Das Verlassen eines Zustands führt schließlich zum Ausführen des Austrittsverhaltens.

In der UML existieren drei Möglichkeiten, diese Verhalten zu spezifizieren: durch eine Operation, durch ein Aktivitätsdiagramm, oder durch einen weiteren Zustandsautomaten. Die Einschränkungen, die bei der Simulation des Verhaltens eines Zustands gelten, werden in Abschnitt 4.3.8 näher beschrieben.

4.3.3 Zusammengesetzte Zustände

Zusammengesetzte Zustände besitzen eine oder mehrere Regionen, die wiederum weitere Unterzustände beinhalten können. Besitzt ein zusammengesetzter Zustand nur eine Region, wird dieser als einfacher zusammengesetzter Zustand bezeichnet. Diese Zustände werden zur Modellierung von Zustands-Hierarchien verwendet. Ein Zustand mit mehreren Regionen wird als orthogonal zusammengesetzter Zustand bezeichnet. Ein solcher Zustand erlaubt es, Parallelität zu modellieren. Ansonsten gilt für einen zusammengesetzten Zustand das bereits bei einem einfachen Zustand Genannte.

Da zusammengesetzte Zustände weitere Unterzustände besitzen, können diese auf mehrere Arten betreten und verlassen werden. Dabei muss zwischen einem einfachen zusammengesetzten Zustand und einem zusammengesetzten orthogonalen Zuständen unterschieden werden.

Einfacher zusammengesetzter Zustand

Ein zusammengesetzter Zustand kann auf zwei Arten betreten werden: durch den sog. Default Entry und einem Explicit Entry.

Bei einem Default Entry endet die Transition am Rand des zusammengesetzten Zustands, so dass dessen Eintrittsverhalten ausgeführt wird und anschließend der Startzustand aktiv ist. Bei einem Default Entry ist ein Startzustand notwendig, damit kein Semantic Variation Point auftreten kann. Ein Semantic Variation Point deutet entweder auf einen Fehler in der Modellierung, oder auf einen zusammengesetzten Zustand hin, der zwar betreten wird, dessen Unterzustände aber nicht betreten werden. Eine solche Situation sollte aufgrund von Mehrdeutigkeiten vermieden werden, indem in jedem zusammengesetzten Zustand ein Startzustand eingefügt wird (vgl. [2], Seite 365).

Ein Explicit Entry erfolgt, wenn die Transition an einem Unterzustand eines zusammengesetzten Zustands endet. Beim Betreten des Unterzustands wird indirekt der zusammengesetzte Zustand betreten, so dass ebenfalls das Eintrittsverhalten dieses Zustands ausgeführt wird.

Zum Verlassen eines zusammengesetzten Zustands existieren drei Möglichkeiten, wobei immer das Austrittsverhalten einmal ausgeführt wird. Die erste ist das Verlassen durch Erreichen des Endzustands. Wird ein Endzustand betreten, führt das zum Verlassen des zusammengesetzten Zustands über die ausgehende ungetriggerte Transition.

Die zweite Möglichkeit ist ein Trigger für den zusammengesetzten Zustand. Wenn ein Ereignis den Trigger aktiviert, kann der zusammengesetzte Zustand von jedem Unterzustand aus über die Transition verlassen werden.

Die dritte und schließlich letzte Möglichkeit ist das Verlassen durch einen Trigger für einen Unterzustand, dessen Transition auf einen Zustand außerhalb des zusammengesetzten Zustands zeigt. Abbildung 4.4 zeigt die drei Möglichkeiten, wie ein zusammengesetzter Zustand verlassen werden kann.

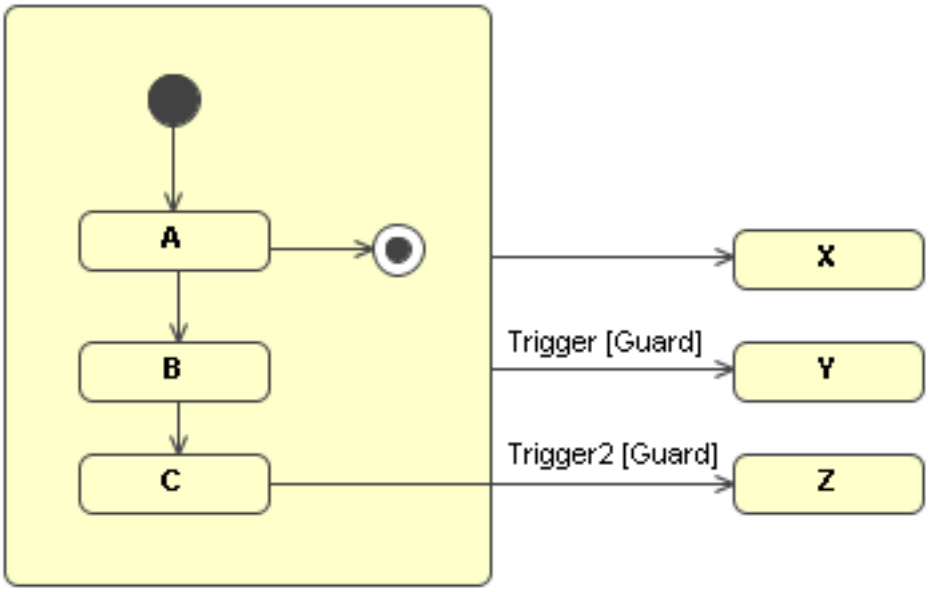

Abbildung 4.4: Verlassen eines einfach zusammengesetzten Zustands

Zusammengesetzter orthogonaler Zustand

Ein zusammengesetzter orthogonaler Zustand kann ebenfalls auf unterschiedliche Arten betreten und verlassen werden, wobei das Eintritts- und Austrittsverhalten abhängig von der Art des Ein- bzw. Austritts ein oder mehrmals ausgeführt wird.

Ein Default Entry erfolgt – genauso wie der Default Entry eines einfachen zusammengesetzten Zustands – durch eine Transition, die am Rand des Zustands endet. Anschließend ist der Startzustand der jeweiligen Regionen aktiv. Das Eintrittsverhalten des zusammengesetzten orthogonalen Zustands wird einmal ausgeführt.

Ein Explicit Entry erfolgt durch eine Gabelung, die eine Transition auf mehrere Transitionen aufspaltet, die dann auf einen Unterzustand in den orthogonalen Regionen zeigen. Das Eintrittsverhalten wird aufgrund mehrerer eingehender Transitionen mehrfach ausgeführt.

Die mehrfache Abarbeitung der Eintrittsverhalten führt dazu, dass die Reihenfolge koordiniert werden muss. Die Abbildung 4.7 zeigt ein Beispiel, bei dem zwei Transitionen von einer Gabelung zu zwei Unterzuständen führen.

In der aktuellen Version der UML 2 Spezifikation wird nicht spezifiziert, in welcher Reihenfolge die einzelnen Verhalten ausgeführt werden. Deshalb existieren für das obige Beispiel zwei mögliche Abläufe, die gleichermaßen auftreten können. Beim ersten Ablauf werden die Eintrittsverhalten sequenziell (nacheinander) ausgeführt, d.h., zuerst *entry_complex*, dann *entry_state1*, anschließend nochmals *entry_complex* und zum Schluss *entry_state2*.

Beim zweiten möglichen Ablauf wird zuerst das Eintrittsverhalten des zusammengesetzten orthogonalen Zustands zweimal ausgeführt und anschließend erst die Eintrittsverhalten der bei-

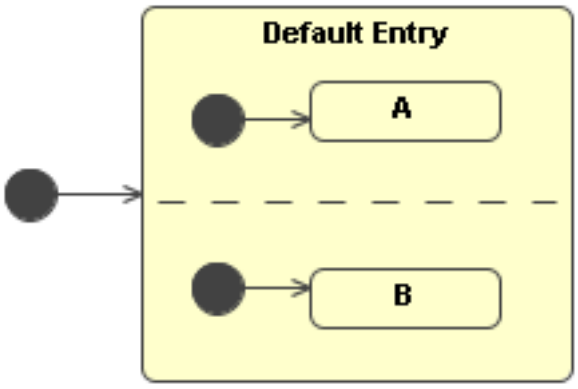

Abbildung 4.5: Default Entry eines zusammengesetzten orthogonalen Zustands

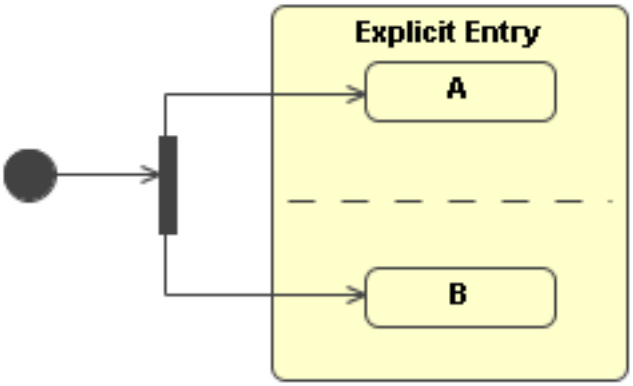

Abbildung 4.6: Explicit Entry eines zusammengesetzten orthogonalen Zustands

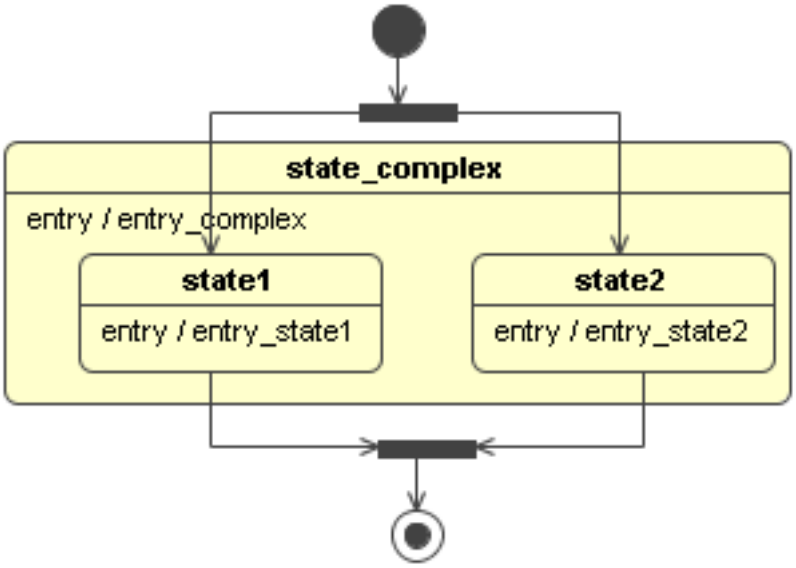

Abbildung 4.7: Zustandsautomat mit konkurrierendem Eintrittsverhalten

den anderen Zustände. Ein möglicher Ablauf für das Beispiel in Abbildung 4.7 ist *entry_complex*, *entry_complex*, *entry_state1*, *entry_state2*. Die Eintrittsverhalten der Unterzustände können natürlich auch gleichzeitig oder in umgekehrter Reihenfolge aufgerufen werden. Die konkrete Reihenfolge hängt von der Zeit ab, die für die Abarbeitung des Verhaltens benötigt wird.

Da zum Zeitpunkt der Modellierung die genauer Daue eines Verhaltens nicht bekannt ist – diese ist vielmehr abhängig von der späteren Implementierung –, muss eine bestimmte Reihenfolge unabhängig von der Dauer eines Verhaltens spezifizieren werden.

Eine Möglichkeit ist die Reihenfolge durch eine Timing-Einschränkung anzugeben, die als Kommentar an den betroffenen Elementen angehängt wird. Dadurch wird sichergestellt, dass die Verhalten in der angegeben Reihenfolge ausgeführt werden. Zur Dokumentation des Modells ist dieses Vorgehen sehr leicht möglich, leider tritt bei der automatischen Simulation dadurch ein Problem: Damit die Einschränkung durch den Simulator automatisch verarbeitet werden kann, muss diese für den Rechner interpretierbar spezifiziert werden, wobei die natürliche Sprache als Spezifikationssprache ungeeignet ist.

Der selbst entwickelte Simulator verwendet eine weitere Möglichkeit, um eine bestimmte Reihenfolge zu erzielen. Da die Zeit, die für die Abarbeitung eines Verhaltens benötigt wird, noch nicht bekannt ist, und der Simulator ohne Simulationszeit arbeitet, kann vor der Simulation festgelegt werden, welcher der oben genannten Abläufe (sequenziell oder parallel) verwendet werden soll.

Anzumerken wäre noch, dass das für das Eintrittsverhalten durch eine Gabelung Genannte äquivalent für das Austrittsverhalten durch eine Vereinigung gilt.

Bei der Simulation eines zusammengesetzten orthogonalen Zustands tritt ein weiteres Problem im Zusammenhang mit den verwendeten Variablen auf: Da die Verhalten in den orthogonalen Regionen gleichzeitig auf ein und dieselbe Variable zugreifen können, muss der Zugriff auf diese Variable koordiniert werden, damit keine Race-Condition auftritt. Dieses Problem tritt ebenfalls bei parallelen Abläufen durch eine Gabelung auf, so dass das hier Genannte für eine Gabelung ebenfalls gilt.

Abbildung 4.8 zeigt, wie beispielsweise eine Race-Condition durch die gleichzeitige Verwendung einer Variablen entstehen kann. Da der Effekt der ersten Transition die Variable *i* um eins erniedrigt und der Effekt der zweiten Transition um eins erhöht, hängt der Wert der Variablen – und damit auch Wert der beiden Guard-Bedingungen – davon ab, welche Transition als erstes durchlaufen wird. In der UML 2 wird nicht spezifiziert, wie bei gleichzeitigem Zugriff auf eine Variable vorgegangen werden soll, so dass prinzipiell drei Lösungen für das Problem möglich sind: das Serialisieren des Zugriffs auf die Variablen, das Duplizieren der verwendeten Variablen für jeden Zweig und das Vermeiden einer solchen Situation.

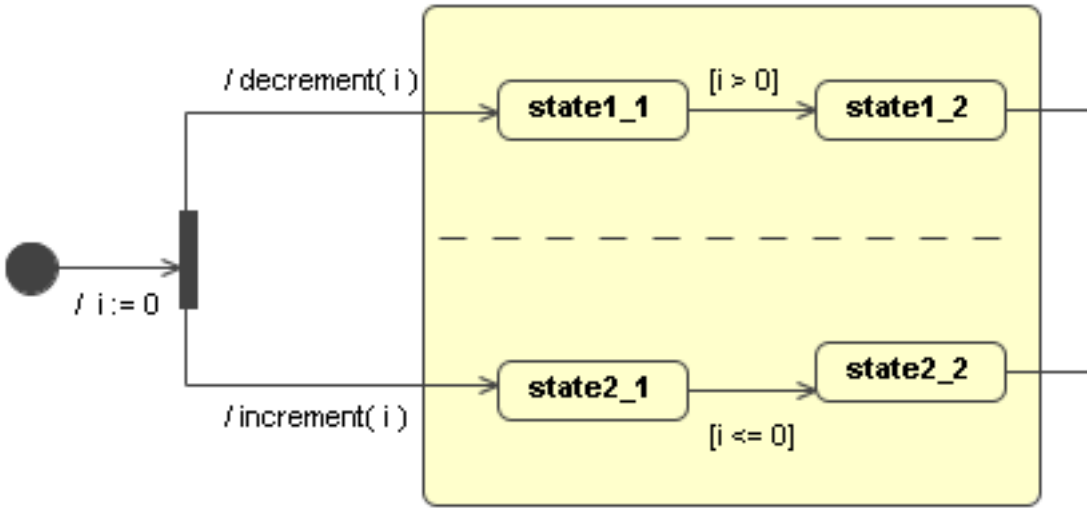

Abbildung 4.8: Konkurrierender Zugriff auf Variablen

Beim Serialisieren des Zugriffs wird eine Reihenfolge für den Zugriff festgelegt, so dass jede Operation atomar auf die Variable zugreifen kann. Es stellt sich dabei aber die Frage, in welcher Reihenfolge die Operationen ausgeführt werden sollen, und ob das Verhalten durch das Serialisieren noch korrekt ist und nicht vom gewünschten Verhalten abweicht.

Eine andere Möglichkeit ist, für jede orthogonale Region eine Kopie der Variablen zu erzeugen, so dass Änderungen nur lokal erfolgen. Leider ist diese Lösung nur dann möglich, wenn die Variable nach dem Zusammenführen des Ablaufs durch eine Vereinigung nicht mehr verwendet wird, da sonst das Problem auftritt, welcher lokale Variablenwert als neuer Wert für die ursprüngliche Variable verwendet werden soll. Abbildung 4.9 stellt die Vorgehensweise und das neue Problem dar, das entsteht.

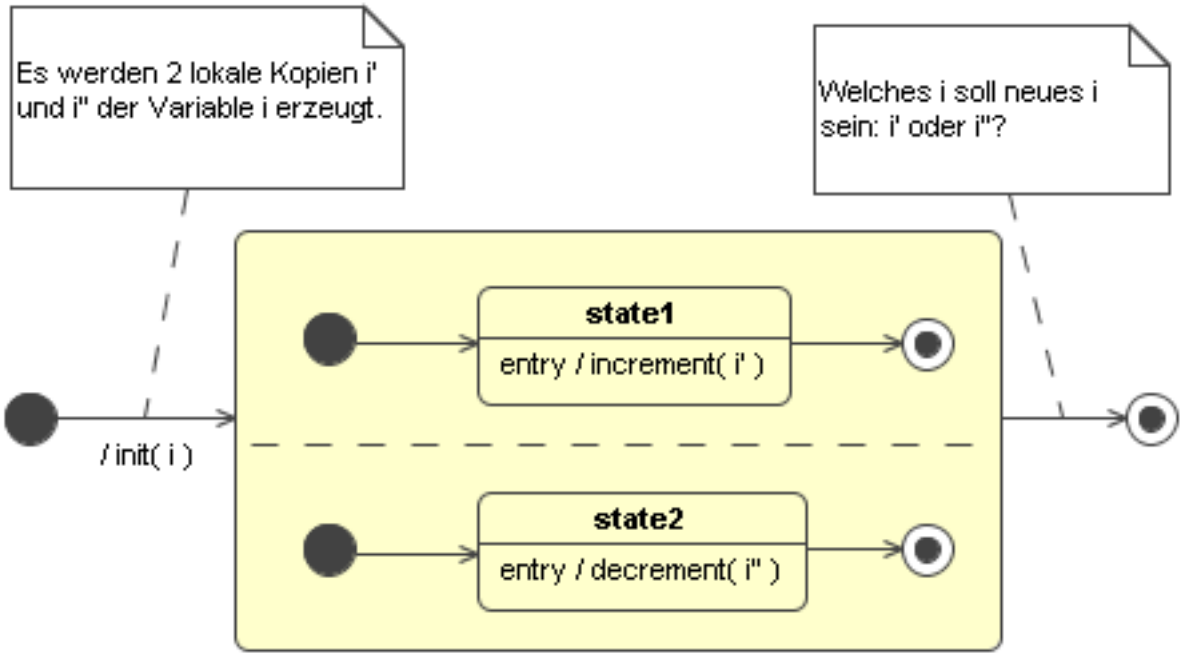

Abbildung 4.9: Duplizieren der Variable

Die beste – aber eigentlich keine richtige – Lösung ist, bereits bei der Modellierung darauf zu achten, dass eine Variable zu einem Zeitpunkt nur einmal verwendet wird. Spätestens bei der Implementierung des Verhaltens muss der konkurrierende Zugriff aufgelöst werden, um eine eventuell auftretende Race-Condidion – und damit eine schwer aufzufindende Fehlerquelle – a priori zu vermeiden.

Aufgrund der Nachteile der beiden vorangehenden Verfahren wurde keines der beiden Verfahren im Simulator implementiert, so dass das Ergebnis der Simulation bei konkurrierenden Zugriff unbestimmt ist.

Ein zusammengesetzter orthogonaler Zustand kann auf die drei Arten verlassen werden, die bereits in Abschnitt über den einfachen zusammengesetzten Zustand beschrieben wurden, wobei das Austrittsverhalten nur einmal ausgeführt wird.

Zusätzlich existiert noch die vierte Möglichkeit, über eine Vereinigung den zusammengesetzten orthogonalen Zustand zu verlassen. Dabei wird das Austrittsverhalten mehrmals ausgeführt, so dass ein ähnliches Problem auftritt, das bereits beim mehrfachen konkurrierenden Eintrittsverhalten beschrieben wurde. Abbildung 4.10 zeigt die vier Möglichkeiten, einen zusammengesetzten orthogonalen Zustand zu verlassen.

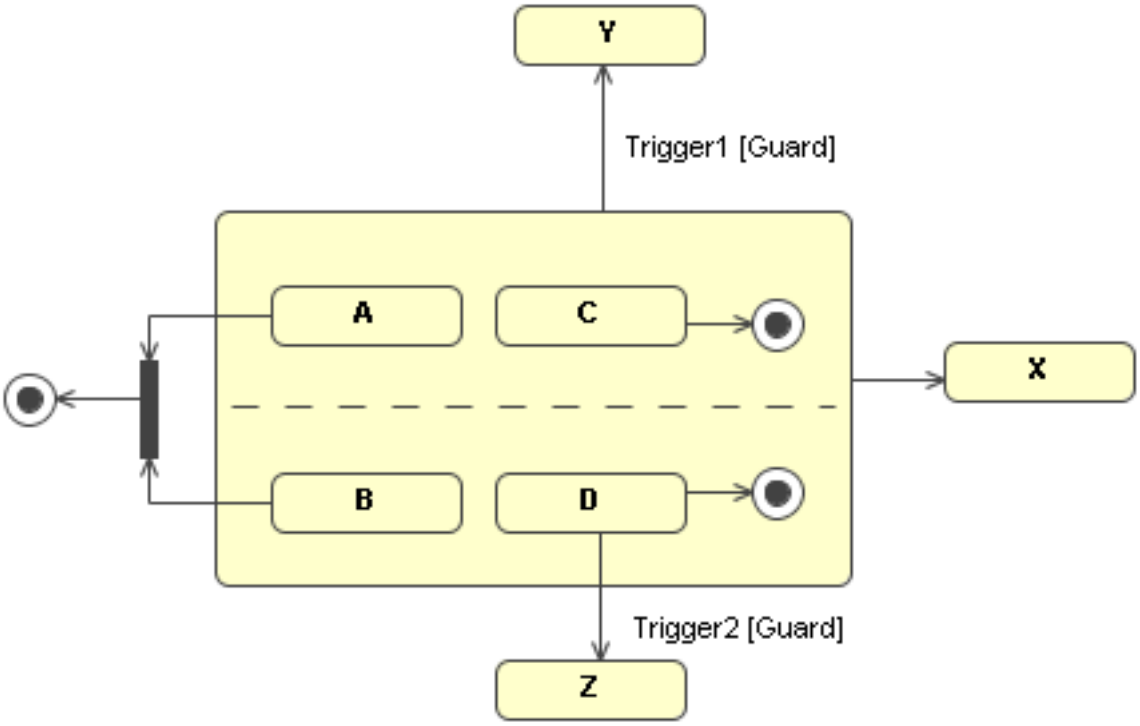

Abbildung 4.10: Verlassen eines zusammengesetzten orthogonalen Zustands

4.3.4 Unterzustandsautomatenzustand

Ein Unterzustandsautomatenzustand ist ein Platzhalter für einen Zustandsautomaten – dem Unterzustandsautomat. Unterzustandsautomaten werden entweder verwendet, um einen bestimmten Teilbereich wieder zu verwenden, oder um einen zusammengesetzten Zustand aus Platzgründen auszulagern.

Betreten werden kann ein Unterzustandsautomat entweder über einen Startzustand oder über einen Eintrittspunkt, wobei das optionale Eintrittsverhalten des Unterzustandsautomaten ausgeführt wird. Verlassen werden kann ein Unterzustandsautomat über einen Austrittspunkt oder über Endzustände, wobei wiederum das Austrittsverhalten ausgeführt wird.

Da ein Unterzustandsautomatenzustand eine Mischung aus Zustandsautomat und Zustand ist, gelten für ihn die bereits genannten Einschränkungen eines Zustandsautomaten und die eines einfachen Zustands.

4.3.5 Pseudozustand

Die UML Spezifikation beschreibt zehn verschiedene Pseudozustände, „... um komplexe Beziehungen zwischen Zuständen einfach darzustellen“ ([2], S. 355). Ansonsten besitzen Pseudozustände die bereits bei den einfachen Zuständen genannten Eigenschaften.

Zwei besondere Pseudozustände, der Start- und Endzustand, wurden bereits in Kapitel 4.3.1 beschrieben.

Entscheidung/Kreuzung

Die Entscheidung realisiert eine bedingte Verzweigung. Wird der Zustand erreicht, werden die Guard-Bedingungen der ausgehenden Transitionen dynamisch ausgewertet und die Transition mit der wahren Guard-Bedingung anschließend durchlaufen. Eine Entscheidung wird verwendet, wenn „... die Auswahl der Transition vom Ergebnis der auf dem Weg zur Entscheidung bereits getätigten Aktionen abhängt.“ ([2], S. 360)

Die Kreuzung ermöglicht es, Transitionen ohne Zustände hintereinander zu verbinden. Bei der Verwendung muss darauf geachtet werden, dass der Zustand mindestens eine eingehende und eine ausgehende Transition besitzt. Eine Kreuzung ist beispielsweise nützlich, um Transitionen zusammenzufassen, bei denen die Trigger der ausgehenden Transitionen und die Guards und Effekte der eingehenden Trigger übereinstimmen, so dass die Anzahl der Transitionen reduziert werden kann.

Die Entscheidung und die Kreuzung ähneln sich auf dem ersten Blick, unterscheiden sich aber minimal bezüglich der Semantik. Der Unterschied wird anhand der beiden Zustandsautomaten in Abbildung 4.11 beschrieben.

Bei der Kreuzung steht das Ergebnis der beiden Guard-Bedingungen – und somit auch der

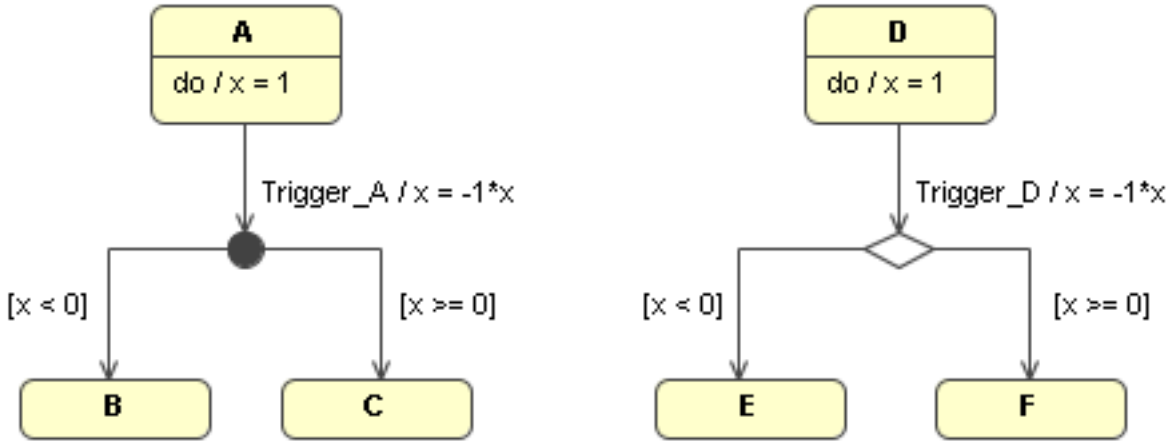

Abbildung 4.11: Unterschied zwischen Kreuzung und Entscheidung, Quelle: [2]

Weg – bereits vor dem Durchlaufen der Transition fest. Das bedeutet, der Effekt der Transition (x = *-1*x*) wirkt sich nicht mehr auf die beiden nachfolgenden Guard-Bedingungen aus, so dass der rechte Zweig durchlaufen wird. Im Gegensatz dazu werden bei einer Entscheidung die Guard-Bedingungen der ausgehenden Transitionen dynamisch berechnet, d.h., der Weg hängt vom aktuellen Wert der Variablen ab. Wird die vom Zustand *D* ausgehende Transition durchlaufen, wird durch das Verhalten der Transition der Wert der Variablen x invertiert. Diese Änderung hat wiederum eine Auswirkung auf die Auswahl das Wegs, so dass bei der Verwendung einer Entscheidung der linke Zweig durchlaufen wird.

Für die korrekte automatische Simulation sollte die unterschiedliche Semantik der beiden Pseudozustände beachtet werden. Weiterhin gelten ebenfalls für die Guard-Bedingungen die in Abschnitt 4.3.6 genannten Einschränkungen.

Gabelung/Vereinigung

Eine Gabelung dient dazu, eine Transition auf mehrere Transitionen aufzuspalten. Dadurch ist es möglich, parallele Abläufe zu modellieren. Das Gegenstück zur Gabelung ist die Vereinigung, die die mit einer Gabelung aufgespalteten Transition wieder zu einer Transition zusammenführt.

Damit die parallelen Bereiche auch wirklich parallel gestartet und beendet werden, dürfen an den ausgehenden Transitionen der Gabelung und an den eingehenden Transitionen der Vereinigung keine Trigger und keine Guards definiert werden. Trigger bzw. Guards an den Transitionen führen nämlich dazu, dass der Ablauf nicht mehr parallel sondern sequenziell ist.

Eine ausgehende Transition einer Vereinigung darf außerdem nur durchlaufen werden, wenn sich der Zustandsautomat in allen Zuständen befindet, die mit der Vereinigung durch eine ausgehende Transition direkt verbunden sind.

Historie-Zustand

Ein Historie-Zustand wird verwendet, um den Zustand beim Betreten eines zusammengesetzten Zustands oder eines Unterzustandsautomaten dynamisch zu bestimmen. Ein Historie-Zustand steht stellvertretend für den zuletzt aktiven Zustand, der beim Verlassen des zusammengesetzten Zustands aktiv war.

Wird der zusammengesetzte Zustand das erste mal betreten, wird entweder das Standardziel des Historie-Zustands aktiv, oder – falls keine Transition vom Historie-Zustand wegführt – der Zustand über einen Default Entry betreten. Deshalb darf der Historie-Zustand maximal eine ausgehende Transition besitzen.

Die UML unterscheidet zwei Typen von Historie-Zuständen: die flache und die tiefe Historie.

Die flache Historie besitzt eine Einschränkung bei mehreren Hierarchie-Ebenen: Besitzt ein zusammengesetzter Zustand weitere zusammengesetzte Unterzustände, führt die Verwendung einer flachen Historie dazu, dass beim Wiedereintritt nur der zuletzt aktive Zustand der Region aktiv wird, in der sich der Historie-Zustand befindet. Sofern der Zustand weitere zusammengesetzte Unterzustände besitzt, werden bei einer flachen Historie die Unterzustände immer ganz normal über den Default Entry betreten.

Bei der tiefen Historie wird diese Einschränkung umgangen, dass heißt, dass beim Wiedereintritt in einen zusammengesetzten Zustand alle zuletzt aktiven Zustände – ausgehend vom zusammengesetzten Zustand bis hin zu den einfachen Zuständen in der untersten Hierarchie-Ebene – wieder aktiv werden, die beim Verlassen des zusammengesetzten Zustands aktiv waren.

Weiterhin darf ein Historie-Zustand nur einmal pro Region vorkommen, damit ein Historie-Zustand eindeutig einer Region zugeordnet werden kann.

Eintritts-/Austrittspunkt

Eintritts- und Austrittspunkte dienen der Übersichtlichkeit. Durch einen Eintritts- und Austrittspunkt erspart man sich das Antragen mehrerer Transitionen von außen an einen Zustand innerhalb eines zusammengesetzten Zustands. Es gilt die Einschränkung, dass pro Region ist jeweils nur ein Eintritts- und ein Austrittspunkt erlaubt ist.

Anwendung finden diese Zustände bei zusammengesetzten Zuständen und bei Unterzustandsautomatenzuständen.

4.3.6 Transition

Eine Transition ist eine gerichtete Beziehung zwischen einem Quell- und einem Zielzustand. Tritt das Ereignis ein, das mit der Transition verbunden ist – der so genannte Trigger –, und ist außerdem die mit der Transition verknüpfte Bedingung wahr – der so genannte Guard –, wird die Transition durchlaufen und ein Zustandswechsel vom Quell- in den Zielzustand erfolgt. Einer Transition kann außerdem ein Verhalten besitzen – der so genannte Effekt –, das beim

Durchlaufen der Transition ausgeführt wird. Ein Transition, bei der der Quell- und Zielzustand identisch ist, wird als Selbsttransition bezeichnet.

Damit eine Transition automatisch simuliert werden kann, müssen für die Elemente einer Transition die folgenden Einschränkungen gelten.

Guard

Damit der Ablauf der Simulation deterministisch ist, müssen bei mehreren ausgehenden Transitionen mit gleicher Guard-Bedingung die Trigger verschieden sein. Ebenso gilt bei gleichen Triggern, dass die Guard-Bedingungen disjunkt sein müssen.

Weiterhin darf für eine automatische Simulation die Bedingung eines Guards nicht in natürlicher Sprache spezifiziert werden – was ja gemäß der UML Spezifikation erlaubt ist –, sondern muss in einer formalen Sprache spezifiziert werden, damit die Bedingung bei der Simulation automatisch ausgewertet werden kann. Je nach Ergebnis (falsch oder wahr) wird die Transition durchlaufen oder nicht. Für die Variablen, die in einer Bedingung verwendet werden, gelten ebenfalls die in Abschnitt 4.3.8 bei der Beschreibung der Operationen genannten Einschränkungen.

Beispiel: Angenommen, es existieren zwei Transitionen im Zustandsautomaten mit der Guard-Bedingung $i >= 100$ und $i < 100$. Wenn der aktuelle Wert und der Typ von i bekannt ist, kann die Bedingung einfach durch einen Ausdruck Parser automatisch berechnet werden. Die beiden Guard-Bedingungen wurden gemäß der oben genannten Einschränkung disjunkt spezifiziert, so dass die Wertebereiche nicht überlappen.

Eine Sonderform ist der „else“-Guard, der nur im Zusammenhang mit einer Entscheidung verwendet werden kann. Dieser ist wahr, sofern keine andere Guard-Bedingung wahr ist. Dadurch werden automatisch alle Fälle abgedeckt, die nicht bereits durch andere Guards abgedeckt werden. Im oben genannten Beispiel wäre es zum Beispiel möglich gewesen, durch einen „else“-Guard an einer der beiden Transitionen alle Fälle abzudecken, die vom anderen Guard nicht abgedeckt werden.

Trigger

In der UML existieren – je nach Einsatzzweck – für eine Transition unterschiedliche Trigger: SignalTrigger, CallTrigger, TimeTrigger, ChangeTrigger und AnyTrigger.

CallTrigger Um den Aufruf einer Operation zu modellieren wird ein CallTrigger verwendet. Wenn der Aufrufer eine Operation aufrufen möchte, erzeugt dieser ein CallInvocationEvent, dass beim Aufgerufenen ein CallEvent auslöst. Dieser Event löst den CallTrigger einer Transition aus, die anschließend durchlaufen wird und ein Zustandsübergang erfolgt. Zusätzlich können einer Operation Parameter mit übergeben werden, auf die innerhalb der Operation zugegriffen werden kann.

Für die Verhaltensspezifikation der Operation eines CallTriggers gelten ebenfalls die in Abschnitt 4.3.8 beschriebenen Einschränkungen.

SignalTrigger Ein SignalTrigger wird verwendet, wenn der Sender eine asynchrone Nachricht an den Empfänger übermitteln will. Der Sender wartet bei diesem Trigger auf keine Rückantwort vom Empfänger, so dass nach dem Senden der Nachricht der Sender seine Arbeit sofort fortsetzen kann.

Möchte der Sender ein Signal senden, erzeugt er ein SendInvocationEvent, das beim Empfänger als SignalEvent ankommt und das als SignalTrigger für eine Transition dient. Mit einem Signal können zusätzlich Daten als Payload mitgegeben werden, die der Empfänger verwenden kann.

Hauptsächlich werden SignalTrigger zur Modellierung verteilter Systeme verwendet, bei denen eine Nachricht vom Sender den Zustandsübergang beim Empfänger und umgekehrt auslöst.

TimeTrigger Durch einen TimeTrigger kann eine Transition zu einem bestimmten Zeitpunkt (absolute Zeitbedingung) oder nach einer gewissen Zeitdauer (relative Zeitbedingung) ausgelöst werden. Ein solcher Trigger wird nicht von einem anderen Objekt erzeugt, sondern vom konsumierenden Objekt selbst. Durch einen TimeTrigger ist es in verteilten Systemen beispielsweise möglich Transitionen zu beschreiben, die nach dem Ablauf einer bestimmten Zeit (Timeout) durchlaufen werden sollen.

Damit die TimeTrigger automatisch simuliert werden können, wird eine Simulationszeit benötigt. Eine einfache Simulationszeit ist die Rundenanzahl der Simulation. Da diese aber keine echte Zeit darstellt und bei der Modellierung die Anzahl der Runden, die bis zu einer Transition mit einem TimeTrigger ausgeführt werden, nicht bekannt ist, ist diese für die Spezifikation in einem TimeTrigger eher ungeeignet.

Die Verwendung einer virtuellen Zeit ähnlich der realen Zeit ist dafür besser geeignet. Bei einer virtuellen Zeit ist es aber notwendig, die Zeit nach der Abarbeitung eines Verhaltens entsprechend der Dauer vorzustellen. Deshalb muss bei der Modellierung des Verhaltens eine Dauer mit angegeben werden. Wie bereits erwähnt, ist aber in den frühen Phasen der Entwicklung die konkrete Implementierung – und damit auch die Dauer des Verhaltens – noch nicht bekannt, so dass bei einer Simulation die Dauer nicht genau angegeben werden kann.

Für eine automatische Simulation muss weiterhin die Bedingung eines TimeTriggers vom Simulator automatisch ausgewertet werden können, so dass die natürlich Sprache zur Spezifikation ausscheidet. Damit der Rechner die Zeitbedingung auch interpretieren kann, ist beispielsweise die Spezifikation in einer formalen Sprache möglich, wie sie auch für die Spezifikation einer Guard-Bedingung verwendet wird. Beispiel: *time > 23:00* (absolut) oder *time > now + 5 secs* (relativ).

ChangeTrigger Ein ChangeTrigger wird ausgelöst, wenn sich die zugeordnete ChangeExpression von falsch nach wahr ändert. Diese Änderung erfolgt üblicherweise durch die Wertänderung

einer Variablen der ChangeExpression durch eine vorausgehende Aktivität. Eine Transition mit einem ChangeTrigger besitzt normalerweise keine anderen Trigger, so dass diese bei einer Änderung der ChangeExpression von falsch nach wahr umgehend durchlaufen wird.

„Jedes Mal wenn sich der Wert einer Variablen ändert, werden die Werte der Guards, in denen die Variable referenziert wird, neu berechnet.“ ([2], S. 349) Das bedeutet, dass sobald sich der Wert einer Guard-Bedingung von falsch nach wahr ändert, wird ein ChangeTrigger für diesen Guard erzeugt.

Zu erwähnen ist noch, dass in ChangeTrigger nicht explizit an einer Transition angetragen wird, sondern implizit im verwendeten Guard steckt.

AnyTrigger Eine Transition mit einem AnyTrigger wird nur dann durchlaufen, wenn keine andere ausgehende Transition des gleichen Zustands durchlaufen wurde. Der AnyTrigger ist also äquivalent zum „else“-Guard bei einer Entscheidung.

Verhalten (Effekt)

Beim Durchlaufen einer Transition wird das mit der Transition verbundene Verhalten ausgeführt. Das Verhalten wird in Abschnitt 4.3.8 ausführlich beschrieben.

Da das Durchlaufen der Transitionen keine Zeit in Anspruch nimmt (siehe 4.3), darf das Verhalten keine lang andauernden Aktionen enthalten. Es ist auch darauf zu achten, dass durch die Abarbeitung des Verhalten kein weiteres komplexes Verhalten angestoßen wird.

4.3.7 Regionen

Regionen ermöglichen es, Hierarchien und zueinander parallele Abläufe zu beschreiben. Sie sind deshalb nur in Zusammenhang mit zusammengesetzten Zuständen sinnvoll, so dass auf den Abschnitt 4.3.3 verwiesen wird, in dem die zusammengesetzten Zustände ausführlich beschrieben werden.

4.3.8 Verhalten

Die Spezifikation des Verhaltens von Zuständen – das heißt, das Eintritts-, Zustands- und Austrittsverhalten –, sowie die Spezifikation des Effekts einer Transition kann in der UML auf drei unterschiedliche Arten erfolgen: durch eine Interaktion in Form einer Operation, durch ein Aktivitätsdiagramm, oder in Form eines Zustandsautomaten.

Wenn das Verhalten durch eine Operation spezifiziert wird und diese Operation die Bedingungen der Guards beeinflussen können soll, muss für eine automatische Simulation die Spezifikation in einer vom Rechner interpretierbaren Form vorliegen, zum Beispiel in Form einer formalen Sprache. Formale Sprachen erlauben es – im Gegensatz zur natürlichen Sprache – das

Verhalten der Operation automatisch auszuführen. Der bekannteste Vertreter dieser Sprache ist die Programmiersprache.

Zu erwähnen ist noch, dass eine Operation, die die Bedingungen der Guards nicht beeinflusst bei der automatischen Simulation nicht benötigt wird, so dass das Verhalten auch nicht spezifiziert werden muss.

Beim ARTiSAN Studio wird beispielsweise das Verhalten einer Operation durch C bzw C++ Code spezifiziert. Leider ist zu Beginn der Entwicklung der Quelltexts einer Operation noch relativ unbekannt, so dass das Verhalten in einer abstrakteren Form dargestellt werden muss. Eine Möglichkeit – die auch der selbst entwickelte Simulator verwendet – ist, die Änderungen von Variablenwerten durch (mathematische) Ausdrücke zu modellieren.

Beispiel: Angenommen es existiert eine Operation *increment(y: integer)*, die irgendein Verhalten aufweist, als Resultat aber die Variable y um eins erhöht, so kann das Verhalten der Operation durch den Ausdruck $y := y + 1$ abstrahiert werden. Der Aufruf der Operation mit einer Variable x, die zum Beispiel in der Bedingung eines Guards verwendet wird (Beispiel: $x > 0$), erhöht die Variable um eins, so dass die Operation die Bedingung des Guards durch die Änderung des Variablenwerts von x beeinflussen kann. Ein solcher Ausdruck kann sehr leicht durch einen Ausdruck Parser automatisch berechnet werden, wenn der aktuelle Wert und der Typ der Variablen x bekannt ist. Der Typ einer Variablen kann beispielsweise bei der Deklaration der Variablen als Attribut einer Klasse spezifiziert werden.

Damit das Verhalten eines Zustands auch als Aktivitätsdiagramm modelliert werden kann, muss der Simulator auch Aktivitätsdiagramme automatisch simulieren können. Da ein Aktivitätsdiagramm eine Semantik ähnlich den Petri-Netzen besitzt und bereits einige Petri-Netz-Simulatoren existieren (vgl. [8]), können diese relativ einfach simuliert werden. Wie der neu entwickelte Simulator geändert werden muss, damit dieser zusätzlich zu den Zustandsautomaten auch Aktivitätsdiagramme simulieren kann, wird in Kapitel 6.1 näher beschrieben.

Wenn das Verhalten als Zustandsautomat spezifiziert wurde, kann es mit dem bestehenden Simulator simuliert werden. Für die Elemente dieses Zustandsautomaten gelten natürlich ebenfalls die bereits in den vorangehenden Abschnitten genannten Einschränkungen.

Kapitel 5

Simulator

Der Vergleich der Simulatoren in Kapitel 3.3 hat gezeigt, dass die beiden auf dem Markt befindlichen Simulatoren die Anforderungen in Kapitel 3 nur unzureichend erfüllen. Deshalb wurde entschieden, einen Simulator im Rahmen dieser Diplomarbeit neu zu entwickeln, der möglichst alle Anforderungen erfüllt. Durch die Eigenentwicklung ergeben sich außerdem weitere Vorteile:

So kann die Entwicklung in Hinblick auf das verwendetet Datenmodell des bestehenden Projekts erfolgen, sodass die keine Schnittstelle notwendig ist. Ein weiter sehr großer Vorteil einer Eigenentwicklung ist die Verfügbarkeit des Quelltexts, so dass der Simulator im Gegensatz zu den kommerziellen Produkten leicht anpassbar, testbar – zum Beispiel durch JUnit Testfälle – und wartbar ist.

Für den Datenaustausch zwischen dem Simulator und den existierenden Projekt wird das Eclipse UML2 Modeling Framework verwendet, das eine Implementierung des UML2 Metamodells auf der Basis des Eclipse Modeling Frameworks (EMF) ist. Das EMF ist ein Java-Framework zum Erzeugen, Abfragen, Manipulieren, Serialisieren (als XMI) und Validieren strukturierter Modelle, das aus einem Modell automatisch Quellcode erzeugen kann. Das Ziel des Eclipse UML2 Projekts ist eine verwendbare Implementierung des Metamodells zur Unterstützung bei der Entwicklung von Modellierungswerkzeugen und zur Vereinfachung des Modellaustauschs durch ein einheitliches XMI Schema. Für weiterführende Informationen verweise ich auf [4].

Aktuelle unterstützt der Simulator die folgenden Elemente:

- Einfacher Zustand
- Start- und Endzustand, Terminator
- Entscheidung
- Zusammengesetzter (orthogonaler) Zustand
- Gabelung und Vereinigung

- Unterzustandsautomatenzustand
- Historie
- Transition
  - Trigger (CallTrigger und SignalTrigger)
  - Guard
  - Effekt
- Verhalten
  - Operation (inkl. Paramterübergabe)
  - Zustandsautomat

Im Folgenden wird die Arbeitsweise des neu entwickelten Simulators beschrieben, wobei auf die Lösungen besonders komplizierter Sachverhalte detailliert eingegangen wird.

Anmerkung: Im Gegensatz zur UML Spezifikation ist bei der Simulation durch den Simulator der Aufenthalt in Pseudozuständen erlaubt. Aufenthalt bedeutet in diesem Fall aber nicht, dass der Zustandsautomat sich dauerhaft in diesem Zustand aufhält, sondern maximal bis zur nächsten Runde. Durch dieses Vorgehen ist der Ablauf der Simulation einfacher, da pro Runde für jeden aktiven Zustand nur eine Transition und keine Transitionskaskade durchlaufen wird. Außerdem wird durch dieses Vorgehen die nötige Überprüfung auf Sonderfälle vereinfacht.

5.1 Allgemeiner Ablauf

Der Simulator arbeitet rundenbasiert. Zu Beginn einer Runde wird ein Zustand aus der Liste der aktiven Zustände (*ArrayList<Vertex> activeStates*) entnommen und die ausgehenden Transitionen betrachtet. In der Liste der aktiven Zustände werden die Zustände gespeichert, in denen sich der Zustandsautomat gerade befindet. Eine Liste wurde verwendet, da sich der Zustandsautomat durch eine Gabelung oder durch zusammengesetzte orthogonale Zustände in mehr als einem Zustand gleichzeitig befinden kann.

Wenn eine Transition des entnommenen Zustands durchlaufen wurde, wird der Zielzustand zur Liste der in der nächsten Runde aktiven Zustände (*ArrayList<Vertex> nextActiveStates*) hinzugefügt, und ein optional vorhandenes Verhalten – d.h., der Effekt einer Transition und das Eingangs-, Zustands- und Ausgangsverhalten der verbundenen Zustände – wird ausgeführt. Die *nextActiveStates*-Liste wird am Ende der Runde zur *activeStates*-Liste für die nächste Runde.

Wenn keine ausgehende Transition, die durchlaufen werden kann, existiert, wird der entnommene Zustand wieder zur Liste der in der nächsten Runde aktiven Zustände hinzugefügt. Das

erneute Hinzufügen ist notwendig, da zu Beginn der Runde der Zustand aus der Liste entnommen wird. Ist eine Runde beendet, dass heißt, in der Liste der aktiven Zustände befinden sich keine Zustände mehr, werden alle Zustände aus der Liste der in der nächsten Runde aktiven Zustände zur Liste der aktiven Zustände hinzugefügt und anschließend geleert.

Alle Aktionen, die in einer Runde durchgeführt werden, werden als parallel ausgeführt betrachtet, obwohl diese aufgrund der Beschaffenheit des Simulators sequenziell ausgeführt werden. Durch diese Vorgehensweise können parallele Abläufe simuliert werden, obwohl sie tatsächlich in einer bestimmten Reihenfolge ausgeführt werden. Ansonsten wären für eine echte parallele Ausführung mehrere Threads notwendig (siehe Kapitel 6.2.2).

5.2 Simulationsarten

Der neu entwickelte Simulator unterstützt zwei Simulationsarten: die schrittweise manuelle und die automatische Simulation. Bei der Ansteuerung des Simulators (siehe dazu Abschnitt 5.7) kann gewählt werden, ob die Simulation schrittweise manuell oder automatisch bis zu einem vorher festgelegtem Ereignis ausgeführt wird.

5.2.1 Schrittweise manuelle Simulation

Die schrittweise manuelle Simulation erfolgt durch die parameterlose Methode *doStep()*. Diese Methode führt eine der beschriebenen Runde aus und kehrt anschließend zum Aufrufer zurück. Dieser kann dann die nächste Runde vorbereiten, indem er neue aktive Ereignisse für die Trigger hinzufügt, Variablen ausliest und ändert, oder die bereits besuchten Zustände und durchlaufenen Transitionen untersucht. Die Kontrolle des Simulators liegt dabei vollständig beim Aufrufer.

Die schrittweise manuelle Simulation kann beispielsweise dazu verwendet werden, um das Modell zu verifizieren und Fehlverhalten bereits vor der nächsten Runde feststellen zu können. Weiterhin kann dieser Modus zur Überprüfung des Simulators auf Korrektheit und Vollständigkeit verwendet werden; nützlich vor allem bei neu implementierter Funktionalität.

5.2.2 Automatische Simulation

Die automatische Simulation ist die Simulationsart, die für das UNiTeD Projekt relevant ist. In diesem Modus wird das Modell solange simuliert, bis während des Simulation ein vorher festgelegtes Ereignis auftritt. Dazu wird die Methode *StopReason simulate()* verwendet, die am Ende der Simulation zurückkehrt und den Grund für den Abbruch der Simulation angibt. Anhand dieses Rückgabewerts kann anschließend entschieden werden, ob die Simulation mit neuen Eingabedaten wieder aufgenommen, oder vollständig abgebrochen werden soll.

Die *simulate()*-Methode liefert den Grund für das Ende der Simulation in Form eines StopReason-Werts zurück: Der Wert *NoMoreEvents* deutet auf eine leer gelaufene Ereigniswarte-

schlange hin; *partialDeadlock* und *totalDeadlock* geben an, dass ein partieller (nicht implementiert) oder ein totaler Deadlock aufgetreten ist; *Terminated* gibt schließlich an, das die Simulation beendet wurde, weil ein Terminator-Zustand betreten wurde.

5.3 Zustände

In jeder Runde wird die Liste der aktiven Zustände durchlaufen und für jeden Zustand in der Liste überprüft, ob eine ausgehende Transition durchlaufen werden kann (*doSingleStep()*-Methode). Dazu wird mittels der Methode *checkTrigger(Transition)* überprüft, ob das benötigte Ereignis in der Ereigniswarteschlange vorhanden ist. Wenn das Ereignis am Kopf der Warteschlange vorhanden ist, wird die Guard-Bedingung durch die Methode *checkGuard(Transition)* überprüft. Diese Methode übergibt die Bedingung Zwecks Evaluierung an den Ausdruck Parser, der die Bedingung auswertet und bei einem korrekten Ausdruck entweder den Wert wahr oder falsch zurück liefert. Der im Simulator verwendete Ausdruck Parser wird in Abschnitt 4.3.8 genauer beschrieben.

Wenn die Transition durchlaufen werden kann, wird die *addActiveState (Vertex, Vertex)*-Methode aufgerufen, die die durchlaufenen Transitionen in der *passedTransitions*-Liste und die besuchten Zustände in der *visitedVertcies*-Liste speichert. Werden beim Durchlaufen einer Transition zusammengesetzte Zustände betreten oder verlassen, ist die Methode weiterhin für das rekursive Speichern der Unterzustände beim Verlassen eines zusammengesetzten Zustands (Historie), die korrekte Reihenfolge der Eintritts-, Zustands- und Austrittsverhalten und für die Aktualisierung der Liste der externen Trigger zusammengesetzter Zustände zuständig.

Wenn alle Verhalten vollständig abgearbeitet wurden, wird anschließend der Zielzustand zur Liste der in der nächsten Runde aktiven Zustände hinzugefügt.

Zum momentanen Stand der Entwicklung kann das Verhalten einer Transition und das Eintritts-, Zustands- und Austrittsverhalten durch eine Operation oder durch einen weiteren Zustandsautomaten modelliert werden.

5.3.1 Einfacher Zustand

Wird ein einfacher Zustand betreten, werden das Eintritts- und anschließend das Zustandsverhalten ausgeführt (*doEntry()* und *doDo()*-Methode). Das Verlassen eines einfachen Zustand führt zur Ausführung des Austrittsverhaltens (*doExit()*-Methode). Eine weitere Behandlung eines einfachen Zustand ist nicht nötig.

5.3.2 Start-, Endzustand und Terminator

Wird die Simulation durch die *setupSimulation()*-Methode initialisiert, wird der Startzustand des Zustandsautomaten gesucht und als erster Zustand zur Liste der aktiven Zustände hinzugefügt.

Beim Betreten eines zusammengesetzten (orthogonalen) Zustands durch einen Default Entry wird nicht der Zustand selbst, sondern die in ihm enthaltenen Startzustände zur Liste der in der nächsten Runde aktiven Zustände hinzugefügt. Der Simulator erkennt einen Default Entry am *isComposite()* oder *isOrthogonal()*-Flag eines Zustands.

Ist der aktuelle Zustand ein Endzustand innerhalb eines zusammengesetzten Zustands wird der zusammengesetzte Zustand zur Liste der in der nächsten Runde aktiven Zustände hinzugefügt. Dadurch wird in der nächsten Runde die vom zusammengesetzten Zustand ausgehende ungetriggerte Transition durchlaufen. Handelt es sich um einen Endzustand in einer Region eines zusammengesetzten Zustands, wird geprüft, ob sich die anderen Regionen bereits in einem Endzustand befinden und anschließend erst der zusammengesetzte Zustand zur Liste hinzugefügt.

Da ein Endzustand keine ausgehenden Transitionen besitzt, kann dieser nicht mehr verlassen werden, so dass dieser bis zum Ende der Simulation in der Liste der aktiven Zustände verbleibt und von diesem aus kein weiteres Verhalten mehr ausgeführt wird. Sofern andere ausgehende Transitionen durchlaufen werden können, läuft die Simulation weiter; ansonsten bricht die Simulation ab, wenn das *stopIfTotalDeadlock*-Flag gesetzt ist.

Im Gegensatz dazu wird beim Betreten eines Terminator-Zustands die automatische Simulation sofort beendet und als Rückgabewert *terminated* zurück gegeben.

5.3.3 Entscheidung

Eine Entscheidung wird vom Simulator wie ein normaler Zustand behandelt, der mehrere ausgehende ungetriggerte Transitionen mit Guard-Bedingungen besitzt. Bevor durch die Auswertung der Bedingungen entschieden wird, welche Transition durchlaufen werden soll, wird nach einer ausgehenden Transition mit einem *else*-Guard gesucht und zwischen gespeichert. Anschließend wird die Methode *checkGuard()* auf die Guard-Bedingung jeder Transition angewendet, die die bool'sche Bedingung auswertet und den Wert zurück liefert. Ist der Rückgabewert von checkGuard() wahr, wird der Zielzustand der Transition mit *addActiveState()* zur Liste der in der nächsten Runde aktiven Zustände hinzugefügt.

War bei allen ausgehenden Transition – exklusive der *else*-Transition – die Guard-Bedingung falsch, dass heißt, das *atLeastOneTransitionPassed*-Flag hat den Wert falsch, wird stattdessen die zwischengespeicherte *else*-Transition durchlaufen und der Zielzustand hinzugefügt. Ist keine *else*-Transition vorhanden, wird der alte Zustand wieder zur Liste der in der nächsten Runde aktiven Zustände hinzugefügt.

Anmerkung Die Kreuzung, die eine recht ähnliche Semantik besitzt wie eine Entscheidung, wird vom Simulator nicht unterstützt.

5.3.4 Zusammengesetzte Zustände

Die zusammengesetzten Zustände müssen bei der Simulation besonders behandelt werden, da diese eine oder mehrere Regionen mit weiteren Unterzuständen besitzen, und dadurch auf zwei unterschiedliche Art und Weisen betreten werden können. Abgesehen von den anschließend erwähnten Sonderfällen gilt für die zusammengesetzten Zustände das bereits für die Zustände allgemein Genannte.

Einfache zusammengesetzte Zustände

Ein einfacher zusammengesetzter Zustand ist ein Zustand, der einen oder mehrere Unterzustände besitzt. Unterzustände können dabei alle in einem Zustandsautomaten erlaubten Zustände sein, auch wiederum zusammengesetzte Zustände, wodurch eine Hierarchie von Zuständen möglich ist.

Da ein einfacher zusammengesetzter Zustand weitere Unterzustände besitzt, kann dieser auf zwei unterschiedliche Arten betreten werden: durch den Default Entry oder durch einen Explicit Entry. Unabhängig von der Art des Eintritts werden beim Betreten des Zustands die externen Trigger in der *externalTransitions*-Hashtabelle gespeichert.

Betreten durch den Default Entry Ein zusammengesetzter Zustand wird über den Default Entry betreten, wenn eine Transition durchlaufen wurde, die an den Rand des zusammengesetzten Zustands zeigt. Der Simulator erkennt einen Default Entry eines zusammengesetzten Zustands daran, dass das bool'sche Flag *isComposite()* des Zielzustands einer Transition gesetzt ist (siehe Listing 5.1) und fügt den vorhandenen Start-Unterzustand zur Liste der in der nächsten Runde aktiven Zustände hinzu.

Außerdem ist die *addActiveState()*-Methode für das Ausführen der Eintritts-, Zustands und Austrittsverhalten in der korrekten Reihenfolge zuständig. Weiterhin sammelt diese Methode die vom zusammengesetzten Zustand ausgehenden Transition, damit der zusammengesetzte Zustand von jedem seiner Unterzustände aus durch einen Trigger verlassen werden kann.

Betreten durch einen Explicit Entry Wird ein zusammengesetzter Zustand über einen Explicit Entry betreten, muss ebenfalls das Eintrittsverhalten der zusammengesetzten Zustände ausgeführt werden, die den Zielzustand direkt oder indirekt beinhalten.

Wird zum Beispiel der zusammengesetzte Zustand *complex3* in Abbildung 5.2 durch die Transition vom Startzustand aus betreten, wird das Eintrittsverhalten wie folgt ausgeführt:

1. Betreten von Zustand *complex1* durch einen Explicit Entry → Verhalten *entry_complex1* wird ausgeführt
2. Betreten von Zustand *complex2* durch einen Explicit Entry → Verhalten *entry_complex2* wird ausgeführt

```
if ( state.isComposite() || state.isOrthogonal() ) {
  // needed to fix problem with normal states marked as complex
      state
  boolean complexDefaultEntry = false;
  EList regions = state.getRegions();
  for ( int i = 0; i < regions.size(); i++ ) {
    Region region = ( Region ) regions.get( i );
    EList elements = region.getOwnedElements();
    for ( int j = 0; j < elements.size(); j++ ) {
      Object element = elements.get( j );
      if ( element instanceof Pseudostate ) {
        Pseudostate ps = ( Pseudostate ) element;
        if ( ps.getKind() == PseudostateKind.INITIAL_LITERAL )
           {
          complexDefaultEntry = true;
          visitedVertices.add( ps );
          nextActiveStates.add( ps );
        }
      }
    }
  }
  if ( complexDefaultEntry ) {
    // store all outgoing transitions in the hashtable
    EList outgoings = state.getOutgoings();
    if (  outgoings != null && outgoings.size() > 0 ) {
      externalTransitions.put( state, outgoings );
    }
    return;
  }
}
```

Abbildung 5.1: Auszug aus der Methode *addActiveState()*

3. Betreten von Zustand *complex3* durch den Default Entry → Verhalten *entry_complex3* wird ausgeführt

Da der Zustand *complex3* durch einen Default Entry betreten wird, muss dieser einen Start-Unterzustand besitzen, damit kein Semantic Variation Point auftreten kann (siehe 4.3.3). Trifft der Simulator auf einen zusammengesetzten Zustands mit einem Semantic Variation Point, wird der Zustand zwar aktiv, aber aufgrund des fehlenden Startzustand nicht betreten, so dass die Simulation der Unterzustände an diesem Punkt abgebrochen wird.

Wird nun Zustand *complex3* über die gezeigte Transition verlassen, ist die Reihenfolge der einzelnen Verhalten etwas komplizierter: Zuerst wird das Austrittsverhalten des Zustands „complex3“ ausgeführt. Anschließend das Austrittsverhalten aller zusammengesetzten Zustände, die den Zustand direkt oder indirekt beinhalten, bis der zusammengesetzte Zustand erreicht wird, in dem sich der Quell- **und** Zielzustand direkt oder indirekt befindet – im Beispiel ist das der Zustand *complex1*. Bei diesem Zustand wird weder das Eintritts- noch das Austrittsverhalten ausgeführt. Wurden alle Austrittsverhalten der Reihe nach ausgeführt, werden die Eintrittsverhalten der zusammengesetzten Zustände ausgeführt, bis der Zielzustand *state3* schließlich über einen Default Entry betreten wird.

Das heißt allgemein, dass bei einer Hierarchie von Zuständen das Austrittsverhalten der darüber liegenden Zustände solange ausgeführt wird, bis der gemeinsame Zustand erreicht wurde. Anschließend erfolgt das Ausführen der Eintrittsverhalten vom gemeinsamen Zustand aus zum Zielzustand.

Für die korrekte Reihenfolge der verschiedenen Verhalten ist die *addActiveState(Vertex, Vertex)*-Methode zuständig. Damit die Reihenfolge der Zustände und der Verhalten richtig bestimmt werden kann muss festgestellt werden, in welcher Beziehung der Quell- und Zielzustand einer Transition zueinander stehen. Im Simulator wurde dazu ein Algorithmus implementiert, der die Beziehung der beiden Zustände feststellen kann, gleich die zusammengesetzten Zustände liefert, die sich zwischen den beiden Zuständen befinden, und ob es sich um einen Eintritt oder Austritt handelt:

```
if ( sourceRegion != targetRegion ) {
  // determine regions path to the root region of both regions
  ArrayList<Region> rl1 = getPathToRoot( sourceRegion );
  ArrayList<Region> rl2 = getPathToRoot( targetRegion );
  // determine common region in path
  Region common = getCommonRegion( rl1, rl2 );
...
```

Anmerkung Der Algorithmus arbeitet nicht auf der Basis von Zuständen, sondern mit Regionen. Dies wurde so implementiert, da die Wurzelregion, in der sich alle Unterzustände befinden, nicht einem zusammengesetzten Zustand zugeordnet ist, sondern dem Zustandsautomaten selbst.

Zuerst wird der Pfad von der Region, in der sich der Quellzustand befindet, bis zur Wurzelregion ermittelt. Dazu wird die Methode *getPathToRoot(Region)* verwendet, die ausgehend

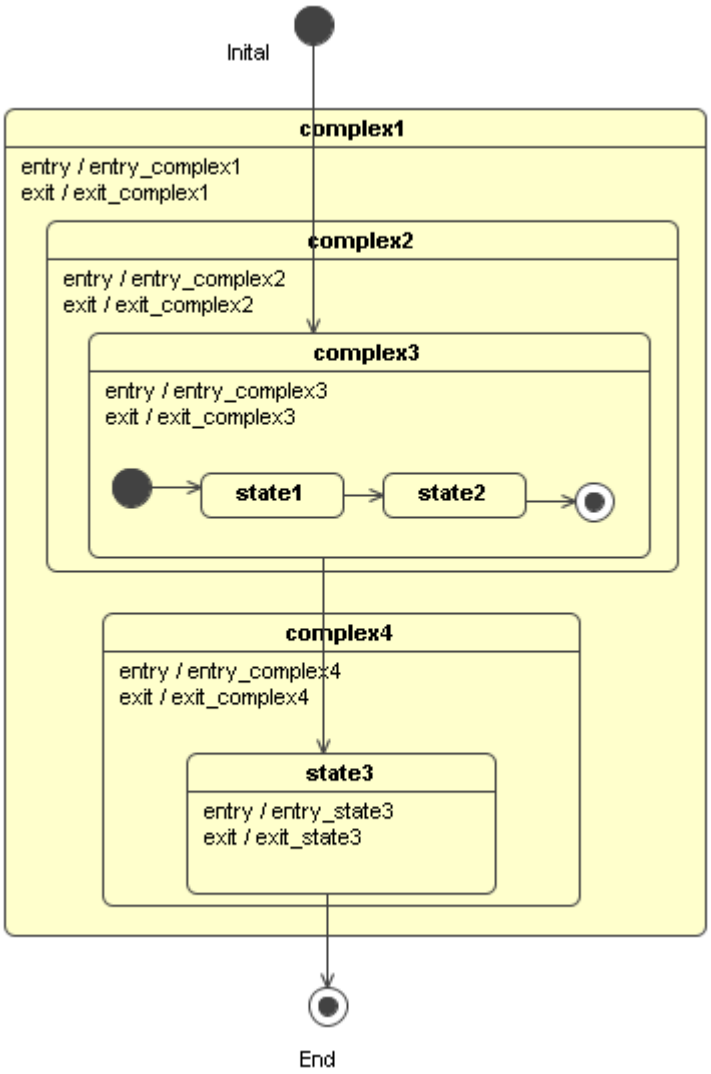

Abbildung 5.2: Zustandsautomat mit einem zusammengesetzten Zustand

von der angegeben Region rekursiv alle Regionen bis zur Wurzelregion bestimmt und in einer Liste speichert. Die Wurzelregion ist die Region, die keinem Zustand zugeordnet ist, sondern dem Zustandsautomaten selbst angehört. Die Wurzelregion wird erkannt, wenn die Abfrage der Methode *getState()* einer Region null zurückliefert.

Für den Zielzustand werden ebenfalls die Regionen bis zur Wurzelregion ermittelt. Anschließend wird mittels der Methode *getCommonRegion(ArrayList<Region>, ArrayList<Region>)* die erste gemeinsame Region beider Listen gesucht. Anschließend kann die Liste *rl1* bis zur gemeinsamen Region durchlaufen und das Austrittsverhalten der zusammengesetzten Zustände ausgeführt werden. Wurden alle Austrittsverhalten ausgeführt, wird der Effekt der Transition ausgeführt. Anschließend wird die Liste *rl2* ausgehend von der gemeinsamen Region durchlaufen und das Eintrittsverhalten der entsprechenden Regionen ausgeführt, bis das Ende der Liste erreicht ist.

Verlassen durch Erreichen des Endzustands Das Verlassen eines zusammengesetzten Zustands über einen Endzustand wird in Abschnitt 5.3.2 bei der Beschreibung der Start und Endzustände näher dargestellt. Zu erwähnen ist noch, dass die *addActiveState*-Methode für die korrekte Reihenfolge der Verhalten der beteiligten Zustände sorgt.

Verlassen durch Trigger für den zusammengesetzten Zustand Wird ein Unterzustand aus der Liste der aktiven Zustände entnommen, wird, bevor die ausgehenden Transitionen des Unterzustands in der Methode *doSingleStep()* überprüft werden, die Hashtabelle *externalTransitions* durchlaufen und mit den gerade aktiven Ereignissen verglichen (*checkTrigger()*). Befindet sich das benötigte Ereignis in der Liste und ist die mit der Transition verbundene Guard-Bedingung wahr (*checkGuard()*), wird der mit der Transition verbundene Effekt ausgeführt und der Zielzustand mittels *addActiveState()* hinzugefügt. Die Methode sorgt außerdem dafür, dass die zuletzt aktiven Zustände des zusammengesetzten Zustands rekursiv in der Hashtabelle *historyStates* gespeichert werden, damit diese beim Betreten eines Historien-Zustands wieder verfügbar sind.

Verlassen durch Trigger für einen Unterzustand Wird eine ausgehende Transition eines Unterzustands durchlaufen, die auf einen Zustand außerhalb des zusammengesetzten Zustands zeigt, wird durch Aufruf der Methode *addActiveState()* mit dem Zielzustand das entsprechende Verhalten in der korrekten Reihenfolge ausgeführt und die zuletzt aktiven Zustände des zusammengesetzten Zustands rekursiv in der Hashtabelle *historyStates* gespeichert, so dass diese beim Betreten eines Historien-Zustands wieder verfügbar sind.

Orthogonale zusammengesetzte Zustände

Zusammengesetzte orthogonale Zustände besitzen im Gegensatz zu den einfachen zusammengesetzten Zuständen mehr als eine Region, die orthogonal – d.h., unabhängig – zueinander sind. Dadurch ist es möglich, parallele Abläufe zu modellieren, die vollständig unabhängig voneinander agieren können.

Orthogonale Zustände können – genauso wie zusammengesetzte Zustände – auf zwei Arten betreten werden: durch den Default Entry oder durch einen Explicit Entry. Unabhängig von der Art des Eintritts werden beim Betreten des Zustands die externen Trigger in der *externalTransitions*-Hashtabelle gespeichert.

Betreten durch den Default Entry Das Betreten eines orthogonalen Zustand mittels eines Default Entrys erfolgt durch eine Transition, die an den Rand des orthogonalen Zustands zeigt. Damit kein Semantic Variantion Point auftreten kann, muss in jeder Region ein Startzustand vorhanden sein. Die Startzustände werden stellvertretend für den Zustand zur Liste der in der nächsten Runde aktiven Zustände hinzugefügt (*addActiveState()*). Der Simulator erkennt orthogonal zusammengesetzte Zustände anhand des *isOrthogonal()*-Flags des Zustands.

Betreten durch einen Explicit Entry Ein orthogonaler Zustand wird durch einen Explicit Entry betreten, wenn ein Zustand innerhalb jeder orthogonalen Regionen betreten wird. Dazu muss eine eingehende Transition mittels einer Gablung für jede Region aufgespaltet werden.

Damit der Simulator einen Explicit Entry erkennt, testet der Simulator bei einer Gabelung, ob die ausgehenden Transitionen auf einen Unterzustand in verschiedenen Regionen eines zusammengesetzten Zustands zeigen. Außerdem wird für jede ausgehende Transition das Verhältnis vom Zielzustand zur Gabelung durch die *getPathToRoot()*-Methode berechnet und die Regionen in einer Liste gespeichert, die anschließend für das Ausführen der Verhalten in der gewählten Reihenfolge verwendet wird.

Bei zusammengesetzten orthogonalen Zuständen, bei denen das Eintritts- und Austrittsverhalten der verschiedenen Regionen durch einen Explicit Entry bzw. Explicit Exit auch mehrmals ausgeführt werden kann, ermöglicht der Simulator zwischen zwei Reihenfolgen für die Abarbeitung der einzelnen Verhalten (*multipleConcurrentBehavior*-Flag) zu wählen: sequenziell oder parallel.

Die sequenzielle Reihenfolge entspricht dem in Abschnitt 5.3.4 beschriebenen Algorithmus, dass heißt, dass bei mehreren eingehenden bzw. ausgehenden Transitionen zuerst die Verhalten eines gesamten Pfads vom Quell- zum Zielzustand geschlossen ausgeführt werden, bevor die Verhalten des nächsten Pfads ausgeführt werden. Für den Zustandsautomaten in Abbildung 5.3 sieht eine mögliche Reihenfolge im sequenziellen Modus wie folgt aus:

1. *entry_complex_1*, *entry_state_2_1*, *entry_state_2_2*, *entry_state_2_3*
2. *entry_complex_1*, *entry_state_1_1*, *entry_state_1_2*, *entry_state_1_3*

Die parallele Reihenfolge ist nicht wirklich parallel, sondern führt die einzelnen Verhalten verschränkt aus. Dazu werden die Regionen für jede von der Gabelung ausgehende Transition bestimmt (*getPathToRoot()*), die zwischen der Gabelung und dem Zielzustand der Transition liegen. Anschließend werden die Listen durchlaufen, die jeweils oberste Region entnommen und das Verhalten des assoziierten Zustands ausgeführt. Dadurch werden die einzelnen Verhalten verschränkt ausgeführt.

Die Reihenfolge der Eintrittsverhalten für das Beispiel in Abbildung 5.3 könnte in etwa wie folgt aussehen:

1. *entry_complex_1* zwei mal
2. *entry_state_1_1*
3. *entry_state_2_1*
4. *entry_state_1_2*
5. *entry_state_2_2*
6. *entry_state_1_3*
7. *entry_state_2_3*

Verlassen durch Erreichen der Endzustände Ist der aktuell ausgewählte Zustand ein Endzustand innerhalb einer Region wird überprüft, ob die Endzustände der anderen Regionen bereits in der Liste der aktiven Zustände vorhanden sind. Wenn ja, wird der zusammengesetzte orthogonale Zustand zur Liste der in der nächsten Runde aktiven Zustände hinzugefügt und alle Endzustände der Regionen aus der Liste der aktiven Zustände entfernt.

Verlassen durch Trigger für den zusammengesetzten orthogonalen Zustand Die Trigger für einen zusammengesetzten orthogonalen Zustand werden genauso behandelt wie bereits bei den einfachen zusammengesetzten Zuständen in Abschnitt 5.3.4 beschrieben. Es werden aber alle aktiven Zustände **jeder** Region rekursiv in der Hashtabelle *historyStates* gespeichert und aus der Liste der aktiven Zustände entfernt.

Verlassen durch Trigger für einen Unterzustand Ein Trigger für eine ausgehende Transition eines Unterzustands, die auf einen Zustand außerhalb des zusammengesetzten Zustands zeigt, wird wie bereits in Abschnitt 5.3.4 beschrieben behandelt. Es werden aber alle aktiven Zustände **jeder** Region rekursiv in der Hashtabelle *historyStates* gespeichert und aus der Liste der aktiven Zustände entfernt.

Verlassen durch eine Vereinigung Zusätzlich existiert bei einem zusammengesetzten orthogonalen Zustand die Möglichkeit, diesen durch eine Vereinigung zu verlassen. Dazu testet der Simulator bei einer Vereinigung, ob sich die Quellzustände aller zur Vereinigung eingehenden Transitionen bereits in der Liste der aktiven Zustände befinden. Ist das der Fall, werden die Regionen bestimmt, die zwischen den Zielzuständen und der Vereinigung liegen, und anschließend die Austrittsverhalten gemäß dem *multipleConcurrentBehavior*-Flag parallel oder sequenziell ausgeführt. Die Vorgehensweise ist dabei die gleiche wie beim Betreten des zusammengesetzten

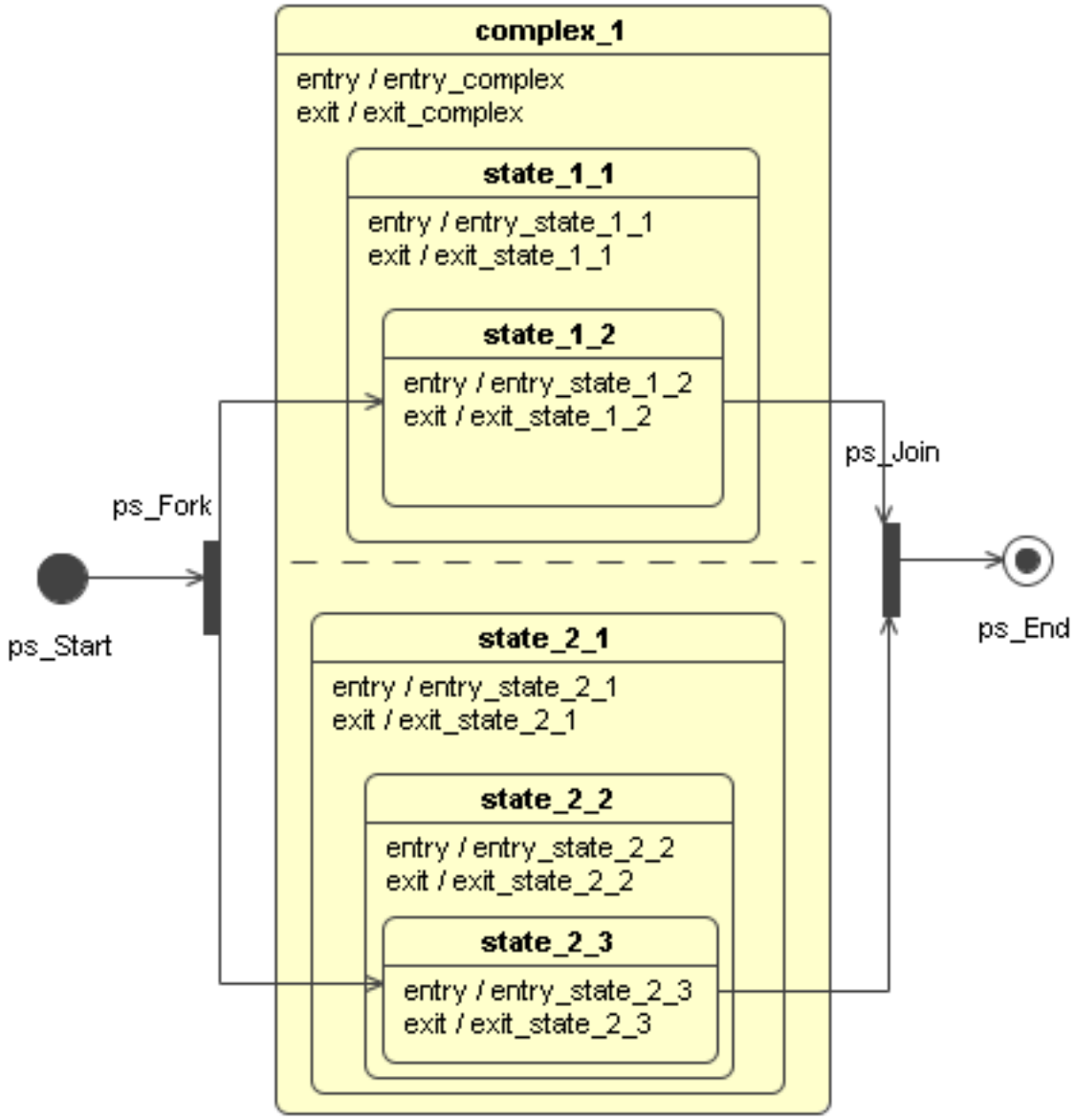

Abbildung 5.3: Zustandsautomat mit einem zusammengesetzten orthogonalen Zustand

orthogonalen Zustands über einen Explicit Entry, nur werden die Verhalten umgekehrt ausgeführt.

Die Abbildung 4.10 zeigt noch mal die vier Möglichkeiten, einen zusammengesetzten orthogonalen Zustand zu verlassen.

5.3.5 Gabelung und Vereinigung

Eine Gabelung teilt den Kontrollfluss auf mehrere parallele Flüsse auf, die mit einer Vereinigung wieder zu einen Kontrollfluss zusammengefasst werden. Entnimmt der Simulator aus der Liste der aktiven Zustände eine Gabelung, werden die Zielzustände der ausgehenden Transitionen zur Liste der in der nächsten Runde aktiven Zustände hinzugefügt. Eine weitere Behandlung der Transitionen ist nicht notwendig, da die von einer Gabelung ausgehenden Transitionen keine Trigger und keinen Guard besitzen dürfen.

Die Behandlung einer Vereinigung ist dagegen etwas aufwändiger, da die von einer Vereinigung ausgehende ungetriggerte Transition erst durchlaufen werden darf, wenn die Quellzustände aller eingehenden Transitionen aktiv sind (eine so genannte UND-Verknüpfung). Wird die *addActiveState()*-Methode mit einer Vereinigung aufgerufen, wird überprüft, ob die Quellzustände aller eingehenden Transitionen bereits in der *activeStates*-Liste vorhanden sind. Ist das der Fall, werden die Quellzustände entfernt und der Zielzustand der ausgehenden Transition zur Liste der in der nächsten Runde aktiven Zustände hinzugefügt – andernfalls wird der alte Zustand wieder hinzugefügt.

5.3.6 Unterzustandsautomatenzustand

er Simulator erkennt einen Unterzustandsautomatenzustand durch die *getSubmachine(StateMachine)*-Methode eines Zustands, die den referenzierten Zustandsautomaten zurückliefert. Dieser wird anschließend durch die Methode *doStateMachine()* simuliert, die in Abschnitt 5.5 detaillierter beschrieben wird.

5.3.7 Historie

Ein Historien-Zustand speichert beim Verlassen eines zusammengesetzten Zustands die zuletzt aktiven Unterzustände. Dabei existieren in der UML zwei Arten von Historien-Zuständen, die sich bezüglich der Ebene unterscheiden, die beim Speichern der Zustände berücksichtigt wird.

Die flache Historie speichert nur die aktiven Zustände der Region, in der sich der Zustand befindet. Dass heißt, die aktiven Zustände von eventuell vorhandenen zusammengesetzten Unterzuständen werden nicht berücksichtigt. Sollen ebenfalls die Unterzustände in den tieferen Ebenen berücksichtigen werden, muss die tiefe Historie verwendet werden, die die aktiven Zustände auch in allen tieferen Ebenen speichert.

Um die Historien-Zustände im Simulator zu simulieren, wird die Hashtabelle *historyStates* verwendet, die in jeder Runde die aktiven Zustände und die zugehörige Region speichert (*storeAllActiveStatesInRegionsRec()*). Falls die aktiven Zustände Endzustände sind und der zusammengesetzte Zustand verlassen wird, wird die Region aus der Hashtabelle entfernt, so dass beim Betreten des Historien-Zustands nicht die zuletzt aktiven Zustände gesetzt werden, sondern der Historien-Zustand über die ungetriggerte Transition verlassen wird.

Flache Historie

Wird in einer Runde eine flache Historie aus der Liste der aktiven Zustände entnommen, wird in der *historyStates* Hashtabelle ein Eintrag für die Region gesucht, in der sich der Historien-Zustand befindet. Wird ein Eintrag in der Hashtabelle gefunden, werden die mit dem Eintrag verbundenen Zustände zur Liste der in der nächsten Runde aktiven Zustände hinzugefügt und die weitere Bearbeitung des Zustands beendet. Falls die Region nicht in der Liste vorhanden ist, wird der Historie-Zustand wie ein normaler Zustand behandelt, dass heißt, der Zielzustand der einzigen ausgehenden, ungetriggerten Transition wird zur Liste der in der nächsten Runde aktiven Zustände hinzugefügt.

Tiefe Historie

Bei einem tiefen Historien-Zustand reicht es nicht aus, nur die Region zu berücksichtigen, in der sich der Historien-Zustand befindet. Deshalb werden bei einer tiefen Historie alle zusammengesetzten Unterzustände in der Region gesucht und rekursiv geprüft, ob für die Regionen der zusammengesetzten Unterzustände ein Eintrag in der Hashtabelle *historyStates* vorhanden ist. Existiert ein solcher Eintrag, werden die Zustände aus der Hashtabelle entnommen und zur Liste der in der nächsten Runde aktiven Zustände hinzugefügt (*setDeepHistoryStatesRec()*). Anschließend wird die weitere Behandlung ebenfalls beendet.

5.4 Transition

In jeder Runde wird die Liste der aktiven Zustände durchlaufen und für jeden Zustand in der Liste überprüft, ob eine ausgehende Transition durchlaufen werden kann. Besitzt eine Transition keinen Trigger (*getTriggers() == null*) und keinen Guard (*getGuard() == null*), wird die Transition umgehend durchlaufen, der Zielzustand zur Liste der in der nächsten Runde aktiven Zustände hinzugefügt und der optionale Effekt der Transition ausgeführt (*doEffect(Transition)*). Der Effekt einer Transition kann – wie in Abschnitt 5.5 beschrieben – durch eine Operation oder durch einen weiteren Zustandsautomaten spezifiziert werden.

5.4.1 Guard

Wenn eine Transition einen Guard besitzt, wird die Guard-Bedingung überprüft. Ist die Guard-Bedingung wahr (*checkGuard() == true*), wird die Transition durchlaufen, der Zielzustand zur Liste der in der nächsten Runde aktiven Zustände hinzugefügt und der optionale Effekt der Transition ausgeführt. Ist die Guard-Bedingung falsch, wird nichts unternommen.

Eine Sonderstellung besitzt ein Guard mit einer *else*-Bedingung, der nur im Zusammenhang mit einer Entscheidung (siehe Abschnitt 5.3.3) verwendet werden darf: Ist keine Guard-Bedingung wahr (*atLeastOneTransitionPassed == false*), wird die Transition mit dem *else*-Guard durchlaufen.

Wie bereits in Kapitel 4.3 beschrieben wurde, müssen bei einigen Elementen Einschränkungen bezüglich der Mächtigkeit der UML gemacht werden, damit diese automatisch simuliert werden können. Für einen Guard gilt, das die Bedingung automatisch berechenbar sein muss, so dass die Spezifikation in einer natürlichen Sprache ausscheidet.

Für die automatische Auswertung der Guard-Bedingung verwendet der Simulator einen Ausdruck Parser, der automatisch Ausdrücke auswerten kann, die mit der Programmiersprache Java vergleichbar sind. In Abschnitt 5.8.2 wird der Ausdruck Parser näher beschrieben. Relevant für die Guard-Bedingungen sind Ausdrücke mit relationalen und logischen Operatoren, die entweder den Wert falsch oder wahr ergeben.

Damit der Ausdruck Parser die Bedingung auswerten kann, ohne eine Exception zu werfen, müssen zudem die Variablen der Guard-Bedingungen in der Klasse als Attribut deklariert werden, für die der Zustandsautomat das Verhalten darstellt. Zusätzlich muss bei der Deklaration der Typ der Variablen mit angegeben werden, so dass der Ausdruck Parser eine Typenüberprüfung vornehmen kann bevor er den Ausdruck auswertet. Abhängig vom Wert des Ausdrucks, den der Ausdruck Parser zurückliefert, wird eine Transition durchlaufen oder nicht.

Außerdem muss, damit eine Bedingung vom Simulator ausgewertet wird, bei der Modellierung der Bedingung als Sprache „Expression“ angegeben werden, da der Simulator ansonsten davon ausgeht, dass zur Spezifikation die natürlich Sprache verwendet wurde und die Guard-Bedingung deshalb nicht betrachtet.

Beispiel: Am Anfang dieser Arbeit in Kapitel 2.1 wurde ein Beispiel eines Zustandsautomaten und die dazugehörende Klasse gezeigt (Abbildung 2.2). Dort wurde die Variable *CD_drin: boolean = false* deklariert, die in einem Guard verwendet wird um festzustellen, ob eine CD breits eingelegt wurde (*CD_drin == true*). Wird nun im *Radiobetrieb*-Zustand die Operation *CD_manuell* durch ein *MyEvent*-Objekt (siehe Abschnitt 5.4.2) in der Ereigniswarteschlange aufgerufen, wertet der Simulator die Bedingung des Guards aus, indem er sie an den Ausdruck Parser übergibt. Anhand des berechnten Wertes wird die Transition durchlaufen oder nicht.

5.4.2 Trigger

Besitzt die Transition einen oder mehrere Trigger, wird mittels der Methode *checkTrigger(Transition)* überprüft, ob das benötigte Ereignis in der Ereigniswarteschlange vorhanden ist. Wenn das benötigte Ereignis am Kopf der Warteschlange vorhanden ist, wird das Ereignis „konsumiert“, bei einem CallEvent die referenzierte Operation ausgeführt, und wahr zurück geliefert. Falls das benötigte Ereignis sich nicht am Kopf der Warteschlange befindet, wird abhängig vom *eventsQueueBehavior*-Flag, das einen Wert der Aufzählung *EventsQueueBehavior* annehmen kann, wie folgt vorgegangen:

Hat das Flag den Wert *discard*, wird das Ereignis am Kopf der Warteschlange verworfen und *checkTrigger()* liefert den Wert falsch zurück, so dass die überprüfte Transition nicht durchlaufen wird. Da das Ereignis aus der Warteschlange entfernt wird, aber weitere ausgehende Transitionen existieren können, die auf das Ereignis warten, wird es zur *nextUnActiveEvents*-Liste hinzugefügt. In dieser Liste werden die Ereignisse gespeichert, die am Ende der Runde aus der Liste der aktiven Zustände entfernt werden. Die Methode *checkTrigger()* überprüft diese Liste ebenfalls, wenn das Ereignis nicht im Kopf der Warteschlange ist und das *eventsQueueBehavior*-Flag den Wert *discard* hat.

Wenn das Flag den Wert *reAdd* hat, wird das Ereignis am Kopf wieder ans Ende der Warteschlange hinzugefügt. Dadurch ist das Ereignis für die nächsten Runden wieder vorhanden, und wandert in jeder Runde Richtung Kopf der Warteschlange. *checkTrigger()* liefert in diesem Fall ebenfalls falsch zurück.

Durch den Wert *skip* wird die Warteschlange nach dem benötigten Ereignis durchsucht, bis es gefunden, oder das Ende der Warteschlange erreicht wurde. Wurde das Ereignis gefunden, liefert *checkTrigger()* den Wert wahr, andernfalls falsch zurück. Der Unterschied zu *reAdd* besteht darin, dass das Ereignis nicht am Kopf der Warteschlange stehen muss, sondern nur in der Warteschlange vorhanden sein muss. Bei *reAdd* muss dagegen das benötigte Ereignis zum richtigen Zeitpunkt am Kopf der Warteschlange stehen.

Die Ereignisse können durch die Methode *addEvent()* zur Ereigniswarteschlange hinzugefügt werden. Die Methode erwartet ein Objekt der Klasse *MyEvent*, das anschließend an das Ende der Ereigniswarteschlange eingefügt wird. Ein solches Objekt kann entweder mit einer Referenz auf eine Operation oder auf ein Signal erzeugt werden. Dabei entspricht ein Objekt mit einer Referenz auf eine Operation den Aufruf dieser Operation (CallTrigger) und ein Objekt mit einer Signal-Referenz dem Senden des Signals (SignalTrigger).

Eine Sonderform stellt eine Transition mit einem AnyTrigger dar: Wurde kein Trigger der ausgehenden Transitionen ausgelöst, dass heißt, keine ausgehende Transition wurde durchlaufen (*atLeastOneTransitionPassed == false*), wird die Transition mit dem AnyTrigger durchlaufen und der Zielzustand zur Liste der in der nächsten Runde aktiven Zustände hinzugefügt.

5.5 Verhalten

Zum aktuellen Stand der Entwicklung kann das Verhalten eines Zustands (Eintritts-, Zustands- und Austrittsverhalten) oder einer Transition (Effekt) durch eine Operation oder durch einen weiteren Zustandsautomaten modelliert werden.

5.5.1 Operation

Zum Zeitpunkt der Modellierung ist die konkrete Implementierung einer Operation noch nicht bekannt. Da aber Operationen bei der automatischen Simulation einen Einfluss auf den Kontrollfluss haben können sollen, müssen die Auswirkungen einer Operation irgendwie modelliert werden.

Beim Simulator werden die Auswirkungen einer Operation durch Wertänderungen der Variablen modelliert, die in den Bedingungen der Guards verwendet werden. Diese Wertänderung erfolgt durch mathematische Ausdrücke, die als Constraint bei der Modellierung an die Operation angehängt werden, und vom Ausdruck Parser ausgewertet werden können. Dazu muss zusätzlich zum aktuellen Wert der Typ der Variablen bekannt sein, weshalb die Variablen als Attribut in der Klasse deklariert werden müssen, für die der Zustandsautomat das Verhalten darstellt. Anschließend können die Operationen darauf lesend und schreibend zugreifen, und dadurch den Kontrollfluss steuern.

Beispiel: Am Anfang dieser Arbeit in Kapitel 2.1 wurde ein Beispiel eines Zustandsautomaten und die dazugehörende Klasse gezeigt (Abbildung 2.2). Dort wurde die Operation *CD_eingelegt(b: boolean)* deklariert, die nach dem Aufruf die Variable *CD_drin* entsprechend dem Parameter *b* setzt, so dass später festgestellt werden kann, ob bereits eine CD eingelegt wurde. Dazu wurde der Operation bei der Modellierung ein Constraint angehängt, der den Wert des Parameters *b* an die Variable *CD_drin* zuweist: *CD_drin := b*. Dadurch wird beispielsweise beim Aufruf der Operation mit dem Parameter *true* ebenfalls der Wert an die Variable *CD_drin* zugewiesen.

Um einer Operation aktuelle Parameter übergeben zu können, müssen den formalen Parametern Werte zugeordnet werden können. Um das EMF Modell nicht ändern zu müssen, erfolgt das Mapping von formalem Parameter zu Wert über die Hashtabelle *parameterConfigurations*. Da eine Transition mehrere Trigger mit gleichen Operationen besitzen kann, ist der Parameter alleine als Schlüssel für die Hashtabelle ungeeignet, weshalb als Schlüssel das Ereignis (*MyEvent*-Objekt) gewählt wurde.

Wird eine Operation über ein Ereignis aufgerufen, werden die Parameter und Werte aus der Hashtabelle ausgelesen und temporär der Laufzeitumgebung des Ausdruck Parsers übergeben. In der Laufzeitumgebung (*MyRuntimeEnvironement*) werden die Variablen, der Typ der Variablen und der aktuelle Wert für den Ausdruck Parser gespeichert. Anschließend werden die Ausdrücke des Constraint der Operation ausgewertet und nach der Auswertung die Parameter wieder aus der Laufzeitumgebung entfernt.

Damit die verwendeten Variablen zu Beginn einen sinnvollen Wert haben, können diese auf

zwei Arten initialisiert werden: Durch einen Default-Wert, der bei der Deklaration der Variablen in der Klasse angegeben wird, oder durch den Effekt der ersten Transition vom Startzustand zum ersten Zustand. Ein Beispiel für die Initialisierung über einen Default-Wert ist *i : Integer = 0* und über den Effekt in Form einer Operation ist *init (i)* mit *i := 0* als Constraint der Operation.

5.5.2 Zustandsautomat

Wenn das Verhalten als Zustandsautomat modelliert wurde, wird durch die Methode *doStateMachine(StateMachine)* eine neue Instanz des Simulator erzeugt und der Zustandsautomat simuliert. Währenddessen wird die Simulation des ursprünglichen Zustandsautomaten angehalten. Die aktiven Ereignisse und deklarierten Variablen werden der neuen Instanz mit übergeben und die Einstellung genauso gesetzt wie beim ursprünglichen Simulator.

Nach dem Beenden der Teilsimulation werden die besuchten Zustände und durchlaufenen Transitionen zur entsprechenden List des aufrufenden Simulators hinzugefügt und die Simulation des ursprünglichen Zustandsautomaten fortgesetzt.

5.6 Einschränkungen des UML Modells

Die folgende Auflistung fast alle Einschränkungen des Simulators, die in den vorherigen Abschnitten erörtert wurden, nochmals zusammen:

- Zustandsautomat:
  - Muss der UML 2 Spezifikation genügen
  - Muss deterministisch sein, d.h., disjunkte Guards und verschiedene Trigger bei ausgehenden Transitionen
- Zustand allgemein:
  - Eintritts-, Zustands- und Austrittsverhalten nur als Operation oder Zustandsautomat modellierbar
  - Zustandsverhalten nicht unterbrechbar, d.h., der Zustand kann erst verlassen werden, wenn das Verhalten vollständig abgearbeitet wurde
- Transition:
  - Als Guard-Bedingungen nur bool'sche Ausdrücke mit deklarierten Variablen oder *else*-Ausdruck erlaubt
  - Trigger
    - Als Trigger nur CallTrigger, SignalTrigger und AnyTrigger möglich
    - Keine Verzögerung eines Triggers durch */defer*

    - Effekt nur durch eine Operation oder durch einen Zustandsautomaten modellierbar

- Entscheidung: Als Bedingung nur bool'sche Ausdrücke mit definierten Variablen erlaubt
- Kreuzung: Nicht implementiert
- Zusammengesetzter orthogonaler Zustand: Keine echte Parallelität
- Gabelung und Vereinigung: Keine echte Parallelität
- Eintritts- und Austrittspunkt: Werden nicht besonders behandelt → keine Connection Point References
- Unterzustandsautomatenzustand:
  - Einschränkungen des referenzierten Unterzustandsautomaten gleich den Einschränkungen eines Zustandsautomaten
  - Keine Connection Point References
- Spezialisierung: Nicht implementiert
- Operation:
  - Verhalten muss durch einen oder mehrere Ausdrücke modelliert werden
  - Verhalten muss als Constraint an die Operation angehängt werden
  - Wertzuweisung an Operationsparameter muss über das Mapping erfolgen

5.7 Ansteuerung des Simulators

Der Simulator wird immer nach dem folgenden Schema angesprochen:

Zuerst muss ein simulierbarer Zustandsautomat erzeugt werden. Dazu existiert die statische Methode *getSimulateableStateMachine (Modell, StateMachine)* der Klasse *SimulateableStateMachine*, die als Parameter das komplette EMF UML2 Modell und den zu simulierenden Zustandsautomaten erwartet und ein Objekt der Klasse *getSimulateableStateMachine* zurückliefert.

Anschließend kann eine Instanz des Simulators durch den Aufruf von *new Simulator(SimulateableStateMachine)* erzeugt werden. Als Parameter erwartet der Konstruktor ein *SimulateableStateMachine* Objekt. Der Konstruktor nimmt außerdem beim Aufruf sinnvolle Voreinstellungen für die Simulation vor.

Die Simulation wird anschließend durch den Aufruf der Methode *simulator.setupSimulation()* vorbereitet. Diese Methode kann auch dazu verwendet werden, die Simulation neu zu starten und die Standardwerte wiederherzustellen.

Nach diesen drei vorbereitenden Schritten kann der Simulator verwendet werden. Für eine schrittweise manuelle Simulation kann die *simulator.doStep()*-Methode ausgeführt werden, die eine Runde der Simulation ausführt. Die *simulator.simulate()*-Methode führt die Simulation automatisch aus, bis ein vorher festgelegtes Ereignis während der Simulation auftritt. Dazu existieren verschiedene Methoden, um die Abbruch-Flags zu setzen und auszulesen:

- *setStopIfNoMoreEvents(boolean)* – Abbruch der Simulation, wenn die Ereigniswarteschlange leer gelaufen ist
- *setStopIfPartialDeadlock (boolean)* – Abbruch, wenn ein partieller Deadlock aufgetreten ist (nicht implementiert)
- *setStopIfTotalDeadlock (boolean)* – Abbruch, wenn ein totaler Deadlock aufgetreten ist

Bevor die *doStep()* bzw. *simulate()*-Methode aufgerufen werden, können am Simulator weitere Einstellungen vorgenommen werden:

- *setEventQueueBehavior(EventsQueueBehavior)* – Setzt das Verhalten der Ereigniswarteschlange (siehe Abschnitt5.4.2)
- *setMultipleConcurrentBehavior(MultipleConcurrentBehavior)* – Setzt das Verhalten bei mehrfach konkurrierendem Eintritts- und Austrittsverhalten (siehe Abschnitt 5.3.4)

Außerdem existieren Methoden,

- um Ereignisse hinzuzufügen (*addEvent(MyEvent)*, *addEvents(ArrayList<MyEvent>)*),
- Variablen zu setzen und auszulesen (*setVariable(String)*, *getVariable(String)*),
- die besuchten Zustände und durchlaufenen Transitionen abzufragen (*getVisitedVertices()* und *getPassedTransitions()*),
- und um die Parameter der Operationen zu setzen (*setEventParameter(MyEvent, Parameter, String)*).

Weitere nützliche Methoden sind in der API im Anhang A.3 dokumentiert.

5.8 Aufbau des Simulators

Der Simulator besteht aus zwei Teilen: der Simulator-Klasse, die für die eigentliche Simulation des Modells verantwortlich ist, und der GUI, die den Verlauf der Simulation grafisch darstellt.

Außerdem verwendet der Simulator die folgenden vier externen Projekte: den Ausdruck Parser zur Auswertung der Ausdrücke, das *CommonModelData* zum Laden von EMF UML2 XMI-Dateien, das Eclipse Modeling Framework und das Eclipse UML2 Version Franework.

5.8.1 Simulator

Die wichtigste Klasse des Simulators ist die *Simulator*-Klasse, in der die gesamte Funktionalität des Simulators steckt. Die Funktionsweise wurde ausführlich in den vorhergehenden Abschnitten beschrieben und die zur Verfügung gestellten Methoden werden in Anhang A.3 dokumentiert.

Der Simulator benötigt zusätzlich zum Ausdruck Parser und dem *CommomModelData* Projekt die folgenden Java-Bibliotheken:

- Eclipse Modeling Framework (EMF) Version 2.2.0
- Eclipse UML2 Framework Version 2.0.1
- OpenArchitectureWare Version 4.1.0 (wird vom *CommonModelData* Projekt benötigt)

5.8.2 Ausdruck Parser

Für die Auswertung der Bedingungen wird der Ausdruck Parser von Heiner Kücker verwendet. Dieser kann automatisch numerische und logische Ausdrücke auswerten, die eine aus der Programmiersprache Java bekannte Syntax haben. Durch Anpassen der Methode *checkGuard()* und *doOperation()* kann aber dieser einfach durch einen anderen Ausdruck Parser ersetzt werden.

Der Ausdruck Parser wird wie folgt verwendet: Zuerst wird eine Instanz der Laufzeitumgebung erzeugt, in der anschließend die Variablen und Variablenwerte für den Ausdruck Parser hinzugefügt werden. Durch die Methode *parse(String)* wird anschließend der Ausdruck geparst und durch *eval(AbstractRuntimeEnvironement)* der Ausdruck ausgewertet. Diese Methode liefert – abhängig vom Wert des Ausdrucks – ein Wrapper-Objekt zurück, dass den Wert beinhaltet.

Der Ausdruck Parser unterstützt die folgenden Operatoren (vgl. [9]):

- String Operatoren
  - `+`: String-Verkettung
- Numerische Operatoren
  - `+`: Addition
  - `-`: Subtraktion oder Negation
  - `*`: Multiplikation
  - `/`: Division
  - `**` oder `^`: Potenz
  - `%`: Modulo
  - `var++`: Erhöhen der Variable um 1 nach der Abfrage
  - `++var`: Erhöhen der Variable um 1 vor der Abfrage

  - `var-`: Verringern der Variable um 1 nach der Abfrage
  - `-var`: Verringern der Variable um 1 vor der Abfrage

- Logische Operatoren
  - `!`: Negation
  - `and` oder `&&`: UND-Verknüpfung mit verkürzter Auswertung
  - `&`: UND-Verknüpfung ohne verkürzte Auswertung
  - `or` oder ||: ODER-Verknüpfung mit verkürzter Auswertung
  - |: ODER-Verknüpfung ohne verkürzte Auswertung
  - `nand`: Negiertes UND mit verkürzter Auswertung
  - `nor`: Negiertes ODER mit verkürzter Auswertung
  - `xor`: Exclusiv-Oder ohne verkürzte Auswertung
- Relationale Operatoren
  - ==: Gleichheit
  - <= oder =<: Kleiner oder gleich
  - >= oder =>: Größer oder gleich
  - != oder <>: Ungleich
- Zuweisungsoperatoren
  - :=: Einfache Zuweisung
  - +=: Addititon und Zuweisung
  - -=: Subtraktion und Zuweisung
  - *=: Multiplikation und Zuweisung
  - /=: Division und Zuweisung
  - :>= oder :`min`=: Zuweisung wenn kleiner als
  - :<= oder :`max`=: Zuweisung wenn größer als

Bei der Auswertung der Ausdrücke gilt folgende Operatorpriorität (absteigend):

1. Prefix- und Postfix-Operatoren
2. Multiplikation und Division
3. Addition und Subtraktion
4. Relationale Operatoren

5. Logische Operatoren
6. Zuweisungsoperatoren

Außerdem unterstützt der Ausdruck Parser einen besonderen Ausdruck: die Sequenz. Eine Sequenz wird in geschwungene Klammern geschrieben und besteht aus keinem bis beliebig vielen Ausdrücken, die durch Semikolon getrennt werden; leere Ausdrücke sind erlaubt. Die einzelnen Ausdrücke werden nacheinander ausgeführt und der Wert des letzten Ausdrucks wird zurück gegeben.

Unterstützte Datentypen für Variablen:

- Numerisch - Aus Genauigkeitsgründen rechnet der Ausdruck Parser aber immer mit BigDecimal
  - `BigDecimal`
  - `BigInteger`
  - `Byte`
  - `Double`
  - `Float`
  - `Integer`
  - `Long`
  - `Short`
- `Boolean` - Relevant für die Guard-Bedingungen (siehe 5.4.1)
- `String` - Mögliche Schreibweisen sind „abc“ oder ’abc’
- Datum/Uhrzeit
  - HH:mm (Beispiel: 0:00 bis 23:59)
  - HH:mm:ss (Beispiel: 0:00:00 bis 23:59:59)
  - HH:mm:ss.S (Beispiel: 0:00:00.00 bis 23:59:59.9999999)

Beispiel: Ist der Wert und der Typ der Variable x und y in der Laufzeitumgebung bekannt, kann zum Beispiel der folgende Ausdruck ausgewertet werden, der x mit der Summe von x und y potenziert und anschließend mit 100 vergleicht: $z := (x \hat{} (x + y)) >= 100$.

Ist die Variable z in der Laufzeitumgebung noch nicht vorhanden, wird vom Ausdruck Parser automatisch die Variable z hinzugefügt. Als Typ der Variable wird der Typ verwendet, der bei der vollständigen Auswertung des Ausdrucks ensteht, so dass im Beispiel die Variable z vom Typ *Boolean* mit dem entsprechenden Wert hinzugefügt wird.

Des Weiteren unterstützt der Ausdruck Parser Arrays, Listen, Maps und das Ansprechen von Objekten über den Punkt-Operator. Auch selbst definierte Funktionen und Operatoren können einfach durch Ableiten von einer abstrakten Klasse und bearbeiten der *parse*-Methode hinzugefügt werden. Für weitere Informationen bezüglich des Ausdruck Parsers verweise ich auf ([9]).

5.8.3 GUI

Der Simulator wird vom UnITeD Projekt direkt über die angebotenen Methoden verwendet, so dass normalerweise eine GUI nicht hätte entwickelt werden müssen. Zu Demonstrations- und Testzwecken wurde aber eine GUI entwickelt, die eine grafische Schnittstelle zwischen Benutzer und der Simulator-API zur Verfügung darstellt. Abbildung 5.4 zeigt einen Screenshot der Oberfläche.

Die GUI ist in vier Bereiche geteilt: Oben ist das Menü und die Werkzeugleiste mit den wichtigsten Funktionen. Links befindet sich die Liste der aktiven Zustände, die Liste aller im Modell möglichen Operationen und Signale und die Tabelle mit den in der Klasse definierten Variablen. In der Mitte werden Informationen über den Simulationsstatus ausgegeben und rechts befindet sich die Liste der gerade aktiven Zustände und eine Liste mit den aktuell besuchten Zustände und durchlaufenen Transitionen.

Über *File/Load model...* kann eine EMF UML2 XMI-Datei ausgewählt werden, die anschließend durch die Klasse ModelLoader geladen wird. Nachdem das Modell vollständig geladen wurde, kann durch die Schaltfläche *Simulate* das Modell automatisch simuliert, oder über *Do Step* (1) eine Runde der Simulation ausgeführt werden. Unter *Simulation/Options* ist es außerdem möglich, Einstellungen am Simulator vorzunehmen.

Bevor die Simulation gestartet wird, kann durch Doppelklicken auf eine Operation oder auf ein Signal (3) ein Ereignis für die Operation oder das Signal in die Liste der aktiven Ereignisse (2) hinzugefügt werden, das anschließend bei der Simulation konsumiert wird. Handelt es sich um eine Operation, kann im darauf folgenden Dialog jedem formalen Parameter ein Wert zugewiesen werden, der für das Verhalten verwendet werden soll.

Die aktuellen Werte der Variablen, die in der Klasse deklariert wurden, werden in der Tabelle (4) angezeigt. Diese Werte werden bei der Auswertung der Bedingungen in einem Guard verwendet. Jede Wertänderung durch ein Verhalten wird in der Tabelle sofort sichtbar.

Im Ausgabebereich (5) werden während der Simulation Informationen über den Ablauf der Simulation – beispielsweise die vom aktuellen Zustand ausgehenden Transitionen, die benötigten Trigger, das Ergebnis der Auswertung einer Bedingung, usw. – angezeigt. Außerdem wird vor den Einträgen einer Runde die Rundenanzahl angezeigt. Wie bereits bei der Beschreibung des Simulators erwähnt, werden alle in einer Runde ausgeführten Aktionen als parallel ausgeführt betrachtet.

Weiterhin werden auf der rechten Seite die aktuell aktiven Zustände angezeigt (6). Durch Aufklappen eines Eintrags können die eingehenden und ausgehenden Transitionen betrachtet werden. Darunter befinden sich die beiden Listen, in denen die besuchten Zustände (7) und

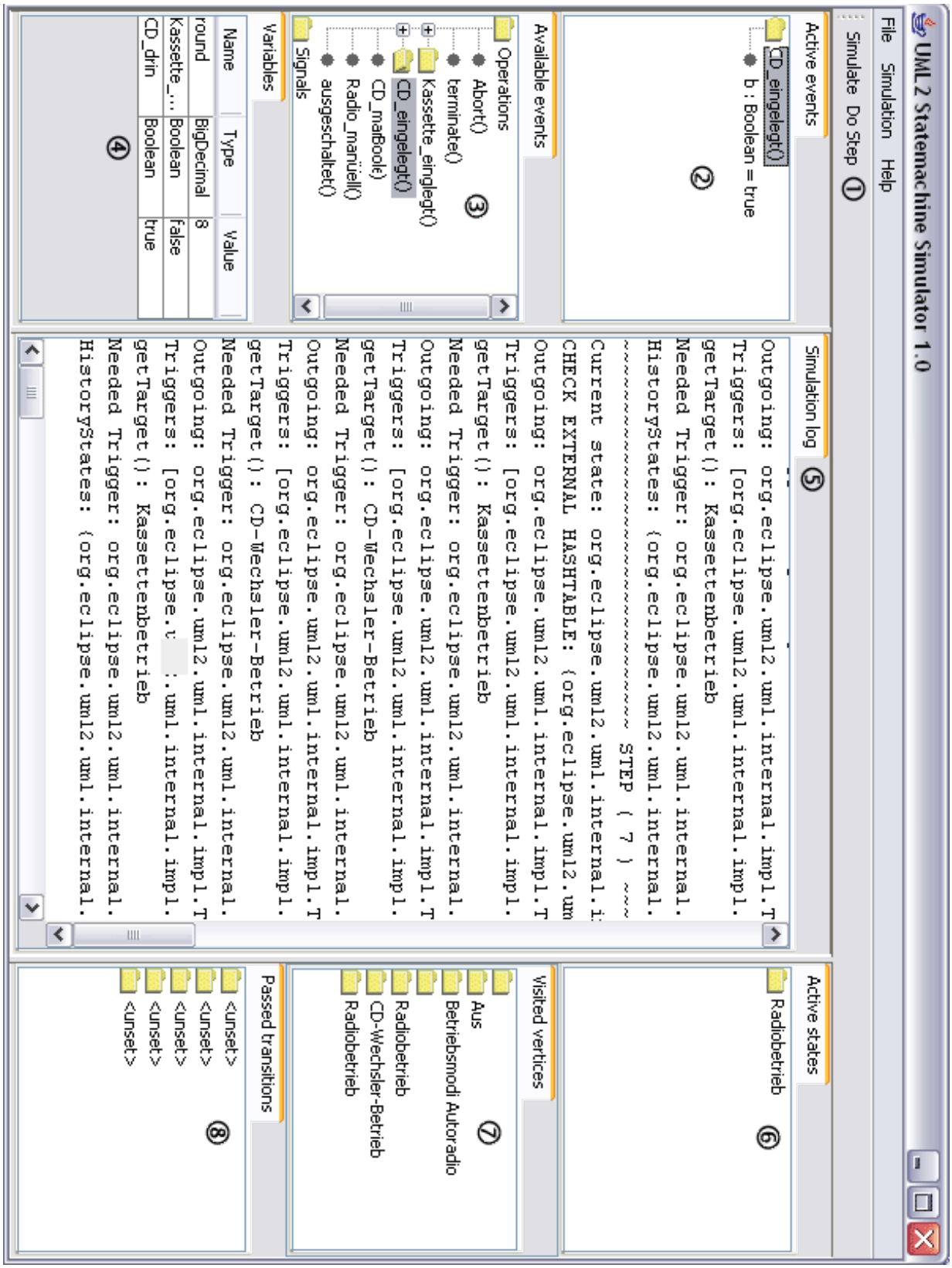

Abbildung 5.4: Screenshot der GUI

durchlaufenen Transitionen (8) dargestellt werden. Die Einträge der Liste können ebenfalls aufgeklappt werden, so dass weitere Informationen über den Eintrag angezeigt werden.

5.8.4 ModelLoader

Für das Laden von EMF UML 2 XMI-Dateien, wie sie beispielsweise das UML Modellierungstool Magic Draw erzeugen kann, wird von der GUI die Klasse ModelLoader verwendet. Diese Klasse verwendet das CommonModelData Projekt, das im Rahmen einer anderen Arbeit am Lehrstuhl entwickelt wurde, um das Modell aus einer XMI Datei zu rekonstruieren.

Kapitel 6

Ausblick

Obwohl der Simulator bereits die Simulation gängiger Zustandsautomaten unterstützt, werden in diesem Kapitel einige Erweiterungen und Verbesserungen kurz vorgestellt, die aus zeitlichen Gründen nicht mehr implementiert wurden. Außerdem wird in einem Abschnitt kurz dargestellt, wie der Simulator angepasst werden muss, damit dieser ebenfalls Aktivitätsdiagramme simulieren kann.

6.1 Übertragung auf andere Diagramme

Der Simulator wurde so implementiert, dass die Unterstützung anderer dynamischer Modelle relativ einfach hinzugefügt werden kann. So kann der ganze Bereich der Variablenverwaltung und -auswertung durch den Ausdruck Parser für alle anderen dynamischen Modelle verwendet werden, die die bedingte Verzweigung oder das bedingte Ausführen von Verhalten ermöglichen.

Im folgenden Abschnitt werden die Maßnahmen beschrieben, die nötig sind, um den Simulator die Simulation von Aktivitätsdiagrammen zu ermöglichen.

6.1.1 Übertragung auf ein Aktivitätsdiagramm

Wenn der Simulator für die Simulation von Aktivitätsdiagrammen angepasst werden soll, können die folgenden Teile wieder verwendet werden:

Die Behandlung der Zustände und Transitionen eines Zustandsautomaten kann relativ einfach für die Behandlung von Aktionen und Kanten verwendet werden, da diese eine recht ähnliche Semantik aufweisen. So erfolgen Übergänge zwischen Aktionen durch Kanten, die – genauso wie Transition – eine Bedingung besitzen können. Für die Behandlung und Auswertung der Bedingungen kann die Variablenverwaltung- und -auswertung durch den Ausdruck Parser vollständig wieder verwendet werden.

Die Behandlung einiger Elemente eines Zustandsautomaten kann für die Behandlung be-

stimmter Elemente eines Aktivitätsdiagramms komplett, oder mit nur leichten Änderungen übernommen werden. So kann die Behandlung der Start- und Endzustände für die Start- und Endknoten vollständig wieder verwendet werden. Das Gleiche gilt für die Behandlung der Verzweigungsknoten, die eine sehr ähnliche Semantik wie die Entscheidungen besitzen.

Genauso ist es grundsätzlich möglich, die Behandlung der Gabelungen und Vereinigungen nach einigen Anpassungen für die Parallelisierungs- und Synchronisationsknoten zu verwenden. Eine Anpassung ist deshalb notwendig, weil bei einem Synchronisationsknoten nicht zwangsläufig eine UND-Verknüpfung gelten muss. Durch eine Synchronisations-Spezifikation ist es möglich, andere Verknüpfungen, zum Beispiel ODER, XOR, usw., zu spezifizieren, so dass der Simulator diese Spezifikation interpretieren muss.

Weiterhin ist es möglich, die Behandlung zusammengesetzter Zustände und der Regionen für Unterbrechungsbereiche und strukturierte Knoten nach geringfügigen Änderungen wieder zu verwenden. Schließlich kann die Ereigniswarteschlange und die Behandlung der Signale nach einigen Änderungen für die Simulation der Signale eines Aktivitätsdiagramms benutzt werden.

Obwohl sehr viele Teile des Simulator für die Simulation von Aktivitätsdiagrammen wieder verwendet werden können, müssen noch einige Erweiterungen hinzugefügt werden, damit eine Simulation ermöglich wird.

Da ein Aktivitätsdiagramm in der UML 2 auf ein erweitertes Petri-Netz basiert, wurde das Konzept der Tokens ebenfalls übernommen, so dass für die Simulation eines Aktivitätsdiagramms die Unterstützung der Tokens hinzugefügt werden muss. So können in einem Aktivitätsdiagramm nicht nur bedingte Kanten, sondern auch gewichtete Kanten existieren, die abhängig von de Anzahl der Token durchlaufen werden oder nicht. Auch die Unterstützung gewichteter Zustände muss in diesem Zusammenhang noch hinzugefügt werden.

Weiterhin existieren einige Elemente, für die es in einem Zustandsautomaten keine entsprechenden Elemente gibt, so dass die Unterstützung noch hinzugefügt werden muss. Das Betrifft die Sprungmarke, die eine Unterbrechung einer Kante ermöglicht, den Verbindungsknoten, der Kanten ohne Synchronisation zusammenführt, die verschiedenen Schleifen (*for*, *while* und *do*) und den Entscheidungsknoten (*if-then-else*).

6.2 Verbesserungen und Erweiterungen

In diesem Abschnitt werden Verbesserungen und Erweiterungen vorgestellt, die aus zeitlichen Gründen noch nicht vorgenommen bzw. implementiert wurden.

6.2.1 Effizienz

Der Simulator verwendet aktuell für die Simulation sehr viele Listen und Hashtabellen. Um die Effizienz des Simulator zu steigern könnten diese durch performantere Listen und Hashtabellen, oder durch andere Datenstrukturen ausgetauscht werden.

Auch die Verwendung von Threads – wie im anschließenden Abschnitt beschrieben – würde vor allem auf Mehrprozessorsystemen einen deutlichen Geschwindigkeitsvorteil bringen.

6.2.2 Threads für echte parallele Ausführung

Die Simulation erfolgt momentan nicht wirklich parallel, sondern alles was in einer Runde durchgeführt wird, wird als parallel ausgeführt betrachtet. Um das Verhalten der parallelen Elemente – dazu zählen die orthogonal zusammengesetzten Zustände und die Gabelung – echt parallel auszuführen, könnte der Simulator durch Threads erweitert werden.

So könnten beispielsweise bei einer Gabelung alle ausgehenden Pfade bis zu einer Vereinigung ermittelt, und anschließend für jeden Zweig ein eigener Thread gestartet werden. In der Vereinigung wird anschließend geprüft, ob die einzelnen Threads fertig abgearbeitet wurden. Wenn ja, wird die Simulation wieder von der Vereinigung aus sequenziell durchgeführt. Das Genannte kann auch in ähnlicher Weise für die parallele Simulation von orthogonalen Regionen in zusammengesetzte Zuständen verwendet werden.

6.2.3 Erkennung partieller Deadlocks

Der Simulator unterstützt momentan nur die Erkennung eines totalen Deadlocks, die Erkennung eines partiellen Deadlocks könnte aber wie folgt hinzugefügt werden.

Vor jeder Runde wird die Liste der aktiven Zustände durchlaufen und die Trigger und Effekte jeder ausgehenden Transition eines Zustands werden paarweise in einer Liste gespeichert. Wird nun im Laufe der Simulation festgestellt, dass mehrere Transitionen nicht durchlaufen werden können, weil der passende Trigger fehlt, wird die Liste durchlaufen und nach einem Zyklus gesucht. Wird ein Zyklus gefunden, wird der Effekt von einer der beteiligten Transition vorgezogen ausgeführt, so dass der partielle Deadlock aufgelöst wird.

6.2.4 Fehlende Elemente

Wie die Aufzählung in Kapitel 5.6 zeigt, unterstützt der Simulator einige Elemente oder Konzepte noch nicht, oder noch nicht vollständig. Dazu zählen die Kreuzung, der Eintritts- und Austrittspunkt, die damit verbundenen Connection Point References, der Time- und ChangeTrigger und die Spezialisierung. Deshalb möchte ich in diesem Kapitel kurz beschreiben, wie einige momentan nicht unterstützte Elemente eventuell hinzugefügt werden könnten.

Kreuzung

Die Kreuzung kann relativ einfach implementiert werden, da die Semantik recht ähnlich einer Entscheidung ist. Es werden aber die Guard-Bedingungen der ausgehenden Transitionen bei einer Kreuzung nicht dynamisch mit den aktuell anliegenden Werten ausgewertet, sondern mit den

Werten vor dem Durchlaufen der eingehenden Transition. Dadurch steht der Weg bereits fest und kann nicht mehr durch einen Effekt der eingehenden Transition beeinflusst werden.

Um die Behandlung einer Kreuzung hinzuzufügen, könnte man den Effekt der eingehenden Transition so lange verzögern, bis die Bedingungen der Guards ausgewertet wurden. Anschließend wird der Effekt der Transition ausgeführt und die Variablen entsprechend aktualisiert.

ChangeTrigger

Der ChangeTrigger wird ausgelöst, wenn sich eine Guard-Bedingung von falsch nach wahr ändert. Die Unterstützung könnte im Simulator wie folgt aussehen:

Wenn bei der Überprüfung der Trigger einer Transition ein ChangeTrigger entdeckt wird und die mit dem Trigger verbundene Bedingung falsch ist, wird dieser in einer Liste gespeichert und die Transition nicht durchlaufen. Ändert sich nun der Wert einer Variablen – dazu muss die Laufzeitumgebung so angepasst werden, so dass jede Änderung gemeldet wird –, werden die mit dem ChangeTrigger verbundenen Bedingungen mit den neuen Werten ausgewertet. Wenn die Auswertung einer Bedingung wahr ergibt, wird die damit verbundene Transition durchlaufen und der Trigger aus der Liste entfernt.

Anhang A

Wichtige Datenstrukturen und Klassen

A.1 Aufzählung StopReason - Grund, warum die Simulation gestoppt wurde

```
public enum StopReason {
  None,
  partialDeadlock,
  totalDeadlock,
  noMoreEvents,
  terminated
};
```

A.2 Aufzählung EventsQueueBehavior - Verhalten der Ereigniswarteschlange

```
public enum EventsQueueBehavior {
  discard,
  reAdd,
  skip
};
```

A.3 Die API

In diesem Kapitel werden die wichtigsten Methoden und Klassen des Simulators beschrieben.

A.3.1 Simulator.java

Initialisierung

`public Simulator( AbstractSimulateableStateMachine ssm );`
Konstruktor der einen simulierbaren Zustandsautomaten erwartet (siehe A.3.3)

`public void setupSimulation() throws NoValidStateMachineException;`
Initialisiert die Simulation und setzt den Startzustand des Zustandsautomaten als aktiv.

Diverse Optionen

`public void setEventQueueBehavior ( EventsQueueBehavior eqb );`
Setzt das Verhalten der Ereigniswarteschlange (discard, reAdd, skip)

`public EventsQueueBehavior getEventQueueBehavior();`
Liefert das Verhalten der Ereigniswarteschlange zurück.

`public void setMultipleConcurrentBehavior( MultipleConcurrentBeavior mcq );`
Setzt die Reihenfolge bei mehrfach konkurrierendem Verhalten.

`public MultipleConcurrentBeavior getMultipleConcurrentBehavior();`
Liefert die Reihenfolge bei mehrfach konkurrierendem Verhalten zurück.

`public void setStopIfNoMoreEvents( boolean b );`
Automatische Simulation anhalten, wenn die Ereigniswarteschlange leer gelaufen ist.

`public boolean getStopIfNoMoreEvents();`
Gibt an, ob die automatische Simulation bei einer leer gelaufenen Ereigniswarteschlange angehalten werden soll.

`public void setStopIfPartialDeadlock( boolean b );`
Automatische Simulation bei einem partielle Deadlock anhalten (nicht implementiert).

`public boolean getStopIfPartialDeadlock();`
Gibt an, ob die automatische Simulation bei einem partiellen Deadlock angehalten werden soll (nicht implementiert).

`public void setStopIfTotalDeadlock( boolean b );`
Automatische Simulation bei einem totalen Deadlock anhalten.

`public boolean getStopIfTotalDeadlock();`
Gibt an, ob die automatische Simulation bei einem totalen Deadlock angehalten werden soll.

Ereignisse

```
public void addActiveEvent( MyEvent event );
```
Fügt ein Ereignis am Ende der Ereigniswarteschlange ein.

```
public void addActiveEvents( ArrayList<MyEvent> al );
```
Fügt eine Liste von Ereignissen am Ende der Ereigniswarteschlange ein.

```
public void removeActiveEvent( MyEvent event );
```
Entfernt das Ereignis *event* aus der Ereigniswarteschlange.

```
public void setActiveEvents(ArrayList<MyEvent> al );
```
Setzt alle Ereignisse in der Ereigniswarteschlange.

```
public ArrayList<MyEvent> getActiveEvents();
```
Liefert die Ereigniswarteschlange zurück.

```
public MyEvent callOperation( Operation operation );
```
Erzeugt ein *MyEvent* Objekt mit der Operation *operation* und fügt es am Ende der Ereigniswarteschlange ein.

```
public void setCallEventParameter( MyEvent callEvent, Parameter parameter,
String value );
```
Setzt den Wert des Parameters *parameter* für das Ereignis *callEvent* auf *value*.

```
public String getCallEventParameterValue( CallEvent callEvent, Parameter
parameter );
```
Liefert den Wert des Parameters *parameter* für das Ereignis*callEvent* zurück.

```
public MyEvent sendSignal( Signal signal );
```
Erzeugt ein *MyEvent* Objekt mit dem Signal *signal* und fügt es am Ende der Ereigniswarteschlange ein.

Variablen

```
public void setVariable( String name, Object value );
```
Setzt den Wert der Variablen *name* auf *value*.

```
public Object getVariable ( String name ) throws ExpressionEvaluationException;
```
Liefert den Wert der Variable *name* zurück.

```
public void removeVariable( String name ) throws ExpressionEvaluationException;
```

Entfernt die Variable *name* aus der Laufzeitbibliothek.

`public Hashtable<String, Object> getVariables();`
Liefert eine Liste der Variablen in der Laufzeitumgebung zurück.

Laufzeitumgebung des Ausdruck Parsers

`public AbstractRuntimeEnvironment getRuntimeEnvironment();`
Liefert die aktuelle Laufzeitumgebung zurück.

Simulation

`public StopReason doStep() throws NoValidStateMachineException;`
Führt eine Runde der Simulation aus.

`public StopReason simulate() throws NoValidStateMachineException;`
Simuliert den Zustandsautomaten automatisch, bis ein festgelegtes Ereignis eintritt.

`public ArrayList<Vertex> getVisitedVertices();`
Liefert eine Liste der besuchten Zustände zurück.

`public ArrayList<Transition> getPassedTransitions();`
Liefert eine Liste der durchlaufenen Transitionen zurück.

`public ArrayList<Vertex> getActiveStates();`
Liefert eine Liste der aktuell aktiven Zustände zurück.

`public ArrayList<Vertex> getNextActiveStates();`
Liefert eine Liste der in der nächsten Runde aktiven Zustände zurück.

A.3.2 ModelLoader.java

`public ModelLoader( File file );`
Lädt das durch *file* spezifizierte Modell aus einer EMF UML2 XMI-Datei.

`public Model getModel();`
Liefert das geladene Modell zurück.

A.3.3 SimulateableStateMachine.java

`public StateMachine getStateMachine();`
Liefert den Zustandsautomaten zurück.

`public MyRuntimeEnvironment initalRuntimeEnvironement();`
Liefert die Laufzeitbibliothek zurück.

`public ArrayList<MyEvent> getAvailableEvents();`
Liefert eine Liste der im Modell möglichen Ereignisse (CallEvents, SignalEvents, ...) zurück.

`public Model getModel();`
Liefert das Modell zurück.

`public static SimulateableStateMachine getSimulateableStateMachine( Model model, StateMachine stateMachine );`
Erzeugt aus dem Modell ein *SimulateableStateMachine* Objekt für den Zustandsautomaten, das mit dem Simulator simuliert werden kann.

`public static Hashtable<String, StateMachine> getStateMachines( Model model );`
Liefert eine Liste aller im Modell vorhandenen Zustandsautomaten zurück.

Literaturverzeichnis

[1] http://www11.informatik.uni-erlangen.de/Forschung/Projekte/United/, Stand 3. Dezember 2006

[2] *UML 2 glasklar. Praxiswissen für die UML -Modellierung und Zertifizierung*, Chris Rupp, Jürgen Hahn, und Stefan Queins, Juni 2005

[3] http://www.eclipse.org/emf/, Stand 4. Oktober 2006

[4] http://www.eclipse.org/uml2/, Stand 4. Oktober 2006

[5] http://www.telelogic.de, Stand 17. Oktober 2006

[6] http://www.artisansw.com/, Stand 13. Oktober 2006

[7] http://www.oose.de/umltools.htm, Stand 26. Dezember 2006

[8] http://de.wikipedia.org/wiki/Petri-Netz, Stand 30.12.2006

[9] http://www.heinerkuecker.de/Expression.html, Stand 4. Oktober 2006

[10] http://tech-www.informatik.uni-hamburg.de/applets/java-fsm/, Stand 1. Dezember 2006

Index