



ulm university

universität
uulm

Zur automatischen Verifikation von UML 2 Aktivitätsdiagrammen

DISSERTATION

zur Erlangung des Doktorgrades Dr. rer. nat.
der Fakultät für Ingenieurwissenschaften und Informatik
der Universität Ulm

Alexander Johannes Raschke
aus München

Universität Ulm
Fakultät für Ingenieurwissenschaften und Informatik
Institut für Programmiermethodik und Compilerbau
Institutsdirektor: Prof. Dr. Helmuth Partsch

2009

Amtierender Dekan: Prof. Dr. Michael Weber

1. Gutachter: Prof. Dr. Helmuth Partsch

2. Gutachter: Prof. Dr. Friedrich von Henke

Tag der Promotion: 7. Oktober 2009

Danksagung

An dieser Stelle möchte ich mich bei allen Personen bedanken, die zum Gelingen meiner Arbeit direkt oder indirekt beigetragen haben.

Zunächst möchte ich mich herzlich bei Herrn Prof. Dr. Helmuth Partsch für die Unterstützung meiner Forschungsarbeit und für die angenehme Arbeitsatmosphäre bedanken. Gerade im letzten Jahr der Arbeit hat er mir großzügig Freiräume in der Arbeitsgestaltung eingeräumt. Für Anliegen und Fragen hatte er stets ein offenes Ohr. Herrn Prof. Dr. Friedrich von Henke danke ich für die unkomplizierte Bereitschaft zur Übernahme des Zweitgutachtens und einige Hinweise, die zum Gelingen der Arbeit beigetragen haben.

Die freundliche Umgebung und der humorvolle Umgang mit meinen Kollegen und Kolleginnen des Instituts für Programmiermethodik und Compilerbau hat mir in all den Jahren große Freude bereitet. Besonders hervorzuheben sind hier mein langjähriger Zimmerkollege Dr. Jens Kohlmeyer und Dominik Gessenharter.

Herrn Prof. Dr. Stefan Sarstedt danke ich für die langjährige Zusammenarbeit und für die Wegbereitung dieser Arbeit. Ohne ihm wäre diese Arbeit sicherlich nicht entstanden.

Mein ganz besonderer Dank gilt meiner Frau Brigitte ohne deren Unterstützung, Geduld und Motivation ich niemals so weit gekommen wäre.

Alexander Raschke

Inhaltsverzeichnis

1	Einleitung	1
1.1	Modellgetriebene Softwareentwicklung	2
1.2	Automatisierte Verifikation	3
1.3	Einbettung und Ziele dieser Arbeit	3
1.4	Aufbau der Arbeit	5
1.5	Notationelle Konventionen	5
2	Grundlagen	7
2.1	Model Checking	7
2.1.1	Modellierung des Systems	8
2.1.2	Spezifikation der zu prüfenden Eigenschaften	9
2.1.3	Automatische Verifikation	12
2.1.3.1	Explizites Model Checking	13
2.1.3.2	Symbolisches Model Checking	14
2.2	Abstract State Machines	16
2.2.1	Hintergrund der Abstract State Machines	16
2.2.2	Sequentielle Abstract State Machines	17
2.2.3	Erweiterungen	18
2.2.4	ASM-Konstrukte	19
2.3	UML 2 Aktivitätsdiagramme	19
2.3.1	Syntax	21
2.3.2	Semantik	24
3	Transformation von Aktivitätsdiagrammen in Zustandsübergangssysteme	29
3.1	Einschränkungen	29
3.2	Transformation	34
3.2.1	Übersicht	34
3.2.2	Normalisierung	35
3.2.3	Zustände	38
3.2.4	Flussberechnung	40
3.2.4.1	Datenstruktur	40
3.2.4.2	Vorbereitungen	41
3.2.4.3	Berechnung der einfachen Flüsse	44
3.2.4.4	Berechnung der kombinierten Flüsse	60
3.2.5	Erzeugung der Zustandsübergänge	64
3.2.5.1	Variablendefinitionen	67

3.2.5.2	Zustandsübergänge durch Tokenflüsse	69
3.2.5.3	Zustandsübergänge durch Aktionsausführungen	75
3.3	Zur Korrektheit der Transformation	82
3.3.1	Deduktiver Nachweis	82
3.3.2	Konformitätstest	85
4	Prototypische Umsetzung mit Model Programs und weitere Aspekte	87
4.1	Prototypische Umsetzung mit Model Programs	87
4.1.1	Model Programs	88
4.1.2	Zustandsübergänge in Model Programs	90
4.1.2.1	Definition der Zustandsvariablen	90
4.1.2.2	Zustandsübergänge	93
4.1.3	Fallbeispiele	96
4.1.3.1	SysML Beispiel	97
4.1.3.2	Zweihandpresse	100
4.2	Umgebungsmodellierung	104
4.3	Optimierungen	106
4.3.1	Größenbetrachtungen	107
4.3.2	Optimierungen auf Ebene der Aktivitätsdiagramme	109
4.3.3	Optimierungen auf Übersetzungsebene	111
5	Diskussion und verwandte Arbeiten	115
5.1	Alternative Vorgehensweisen	115
5.2	Diskussion der gewählten Lösung und mögliche Erweiterungen	116
5.3	Verwandte Arbeiten	120
5.3.1	Verifikation von ASMs	120
5.3.2	Verifikation von UML Modellen	122
5.3.3	Verifikation von Workflow Modellen	123
5.3.4	Verifikation von UML 2 Aktivitätsdiagrammen	123
6	Fazit und Ausblick	127
6.1	Beiträge	127
6.2	Ausblick	128
A	ASM-Regeln	131
A.1	Domänendefinitionen	131
A.1.1	Grundlegende Datentypen	131
A.1.2	UML 2 Metamodell	131
A.1.3	Datenstrukturen zur Transformation	135
A.2	Makros zur Flussberechnung	136
A.3	Makros zur Zustandsübergangsbestimmung	149
A.4	Hilfsfunktionen und -makros	159
Glossar		171

Zusammenfassung

Die Anforderungen an die Leistungsfähigkeit und insbesondere an die Qualität neuer Softwaresysteme nehmen stetig zu. Moderne Softwareentwicklungsprozesse haben daher zum Ziel, möglichst schnell qualitativ hochwertige Software zu erstellen. Ein Ansatz, um dieser Herausforderung zu begegnen, ist die modellgetriebene Softwareentwicklung (Model-Driven Software Development, MDSD). Der Einsatz von Modellen erhöht den Abstraktionsgrad in der Softwareentwicklung und ermöglicht eine Steigerung der Produktivität durch Automatisierung.

Die Unified Modeling Language (UML) in der Version 2 ist eine weit verbreitete, für diesen Zweck eingesetzte, grafische Modellierungssprache. Sie definiert verschiedene Diagrammart, um ein System aus verschiedenen Blickwinkeln zu beschreiben. Während z. B. Klassendiagramme die statische Struktur eines Systems beschreiben, dienen Aktivitätsdiagramme der grafischen Verhaltensspezifikation von Klassen oder Methoden.

Aktivitätsdiagramme bestehen aus Aktionen und dazwischenliegenden Kontroll- und Objektflüssen. Zur Steuerung des Ablaufs werden Kontrollknoten eingesetzt, die Flüsse vereinigen, parallel aufspalten und synchronisieren oder zwischen mehreren Alternativen entscheiden. Spezialisierte Aktionen ermöglichen das Versenden und Empfangen von Signalen oder die Einbindung von Zeitaspekten.

Ein Vorteil der Abstraktion ist neben der Produktivitätssteigerung die Möglichkeit der Verifikation. Im Gegensatz zum Test kann Verifikation nicht nur Fehler finden, sondern auch deren Abwesenheit zeigen.

Die vorliegende Arbeit stellt einen Ansatz zur automatisierten Verifikation (Model Checking) von UML 2 Aktivitätsdiagrammen vor. Damit vorhandene Werkzeuge wiederverwendet werden können, wird eine Transformation von Aktivitätsdiagrammen in Zustandsübergangssysteme entwickelt. Durch die Flexibilität bezüglich des eingesetzten Model Checkers kann unmittelbar von der weiteren Entwicklung auf diesem Gebiet profitiert werden.

Während bisherige Arbeiten nur wenige Elemente der Aktivitätsdiagramme unterstützen, deckt diese Arbeit weitgehend den Umfang der von Sarstedt in [Sar06a] entwickelten Semantik ab. Dies beinhaltet auch „schwierigere“ Aspekte wie Datenflüsse, Signalverarbeitung und Unterbrechungsbereiche. Darüber hinaus kann die Semantik an bestimmten Stellen über semantische Variationspunkte den Bedürfnissen des Entwicklers angepasst werden.

Die formalen Ergebnisse der Arbeit wurden auch prototypisch in ein Werkzeug zur modellgetriebenen Softwareentwicklung mit UML 2 Aktivitätsdiagrammen integriert.

In Kapitel 1 wird die Arbeit motiviert und der Bezug zu vorherigen Arbeiten hergestellt, in denen Aktivitätsdiagramme zur modellgetriebenen Softwareentwicklung eingesetzt werden. Bei diesem Ansatz werden Aktionen durch Code implementiert und die Aktivitätsdiagramme direkt ausgeführt. Die dafür erforderliche formale Semantik ist in [Sar06a] spezifiziert.

Die vorliegende Arbeit ermöglicht die automatisierte Verifikation von Aktivitätsdiagram-

men. Dadurch können bereits in frühen Phasen der Softwareentwicklung Systemeigenschaften verifiziert werden. Auch sicherheitskritische Systeme können damit in Aktivitätsdiagrammen modelliert werden.

Die notwendigen Grundlagen werden in Kapitel 2 vermittelt. Das Prinzip des Model Checkings wird erläutert und die Temporallogik CTL* zur Spezifikation von zu prüfenden Eigenschaften eingeführt. Verschiedene Methoden des Model Checkings werden ebenfalls dargestellt.

Dieses Kapitel umfasst außerdem eine Einführung in den Formalismus der Abstract State Machines (ASM). Dieser Formalismus wird einerseits zur Spezifikation des Transformationsalgorithmus im Hauptteil der Arbeit verwendet, andererseits ist die Semantikdefinition von Sarstedt in diesem Formalismus notiert.

Die Grundlagen werden mit einer informellen Beschreibung der Syntax und Semantik der UML 2 Aktivitätsdiagramme abgeschlossen. Die Semantik lehnt sich dabei an die Tokenflusssemantik von Petri-Netzen an.

Der Hauptteil der Arbeit findet sich in Kapitel 3. Es beschreibt die entwickelte Transformation von Aktivitätsdiagrammen in abstrakte Zustandsübergangssysteme.

Zunächst werden notwendige Einschränkungen der verwendeten Semantik erläutert. Der Schwerpunkt liegt jedoch auf der Berechnung der möglichen Kontroll- bzw. Objektflüsse zwischen Aktionen.

Der naive Ansatz, jeweils nur den unmittelbaren Nachfolger eines Knotens zu betrachten, reicht nicht aus, da die UML 2 eine „*traverse-to-completion*“-Semantik für Aktivitätsdiagramm definiert. Dies bedeutet, dass die Kontroll- oder Datentokens immer von einer Aktion zur nächsten fließen und *nicht* an Kontrollknoten gepuffert werden können.

Eine Ausnahme davon ist die parallele Aufspaltung (*ForkNode*), an der Tokens gepuffert werden können, wenn nachfolgende Aktionen noch nicht bereit sind.

Durch diese „globale“ Sicht auf die möglichen Flüsse in Aktivitäten ist die Berechnung dieser entsprechend komplex. Aus diesem Grund werden zunächst nur „einfache Flüsse“ bestimmt. Einfache Flüsse sind Flüsse, die von einer Aktion zu einem Puffer, von einem Puffer zu einer Aktion oder, falls kein Puffer dazwischen liegt, direkt von einer Aktion zu einer anderen gehen. Erst in einem zweiten Schritt werden diese „Flussfragmente“ zu komplexeren Flüssen kombiniert.

Der letzte Abschnitt dieses Kapitels befasst sich mit dem Nachweis der Korrektheit der Umsetzung.

In Kapitel 4 wird das abstrakte Zustandsübergangssystem beispielhaft in eine Eingabesprache für einen Model Checker überführt. Die gewählte Eingabesprache ist Model Programs, die vom Model Program Checker verifiziert werden kann.

Diese Eingabesprache bietet sich an, da sich darin einerseits die abstrakten Zustandsübergänge ohne großen Aufwand abbilden lassen und andererseits höherwertige Datentypen unterstützt werden, die die Umsetzung vereinfachen. Bei der Beschreibung der Umsetzung werden auch mögliche Alternativen diskutiert.

Die Möglichkeiten des Model Checkings von Aktivitätsdiagrammen werden anschließend durch zwei Fallbeispiele verdeutlicht.

In den folgenden Abschnitten des Kapitels werden Möglichkeiten der Umgebungsmodellierung, die für das Model Checking häufig notwendig ist und Optimierungsmöglichkeiten zur

Reduktion des Rechenaufwands diskutiert.

Der gewählte Ansatz wird alternativen Lösungsideen in Kapitel 5 gegenübergestellt. Die eingangs gemachten Einschränkungen werden aufgegriffen und mögliche Erweiterungen zur Überwindung dieser beschrieben.

Schließlich wird die vorliegende Arbeit in den wissenschaftlichen Kontext eingebettet. Dazu werden aus verschiedenen Richtungen verwandte Arbeiten vorgestellt und mit dem entwickelten Ansatz verglichen.

Kapitel 6 fasst die wesentlichen Ergebnisse der Arbeit zusammen. Diese sind:

- Automatisierte Verifikation von UML 2 Aktivitätsdiagrammen für eine umfangreichere Teilmenge als bisherige Ansätze
- Formalisierung der Transformation in Abstract State Machines
- Berücksichtigung von semantischen Variationspunkten
- Prototypische Implementierung

Abschließend wird ein Ausblick auf weitere Schritte und Forschungsthemen gegeben, die auf der vorliegenden Arbeit aufbauen können.

Im Anhang finden sich eine kommentierte Auflistung aller in der Arbeit verwendeten ASM-Makros und ein Glossar der wichtigsten Begriffe.

Summary

Demands on the performance and particularly on the quality of new software systems grow continuously. Therefore, modern software development processes are aimed at creating top quality software as fast as possible. One approach to face this challenge is given by the model-driven software development (MDSO). The use of models increases the abstraction level in software development and allows an enhancement of productivity by automation.

The Unified Modeling Language (UML) version 2 is a wide-spread graphical modeling language applied for this purpose. It defines several diagram types to describe a system from different points of view. While, for example, class diagrams specify the static structure of a system, activity diagrams are designed to describe the behavior of classes and operations graphically.

Activity diagrams consist of actions which may be connected by control- and objectflows. Control nodes are provided for a more detailed control of flows. These nodes split, merge and join flows or decide among different alternatives. Special actions allow for sending and receiving signals or the consideration of time aspects.

Besides the increase of productivity, another advantage of abstraction is verification. Contrary to tests, verification does not only find errors, but it can prove their absence.

This thesis presents an approach for model checking of UML 2 activity diagrams. A transformation of these diagrams into a state transition system is developed in order to reuse existing tools. This flexibility concerning the underlying model checker makes it possible to profit by further enhancements in this area of research.

Whereas previous approaches cover only few aspects of activity diagrams, this thesis supports for the most part of the semantics developed by Sarstedt in [Sar06a]. This includes more difficult aspects like object flows, signal handling and interruptible activity regions. Additionally, the semantics can be adjusted to individual needs by semantic variation points.

The formal results of this thesis are also prototypically integrated into a tool for model-driven development with UML 2 activity diagrams.

Chapter 1 motivates this thesis and relates it to previous work, where activity diagrams are used for model-driven software development. In this approach actions are implemented by code and activity diagrams are interpreted directly, without recoding the control flow. The semantics needed for this approach is specified in [Sar06a].

This thesis facilitates model checking of activity diagrams. This makes it possible to verify system properties already in early phases of the software development process. Thus, also safety critical systems can be modeled with activity diagrams.

Necessary basics are given in chapter 2. The principle of model checking is explained and the temporal logic CTL* is introduced for formalizing the property to be proved. Several

methods of model checking are presented.

In addition, this chapter includes an introduction of the formalism of Abstract State Machines (ASM). On the one hand, this formalism is used to specify the transformation algorithm in the main part. On the other hand, the semantics definition of Sarstedt is listed by this formalism.

The basics are completed by an informal description of the syntax and semantics of UML 2 activity diagrams. The semantics follows the token flow semantics of petri nets.

The main part of this thesis is to be found in chapter 3 where the developed transformation from activity diagrams into a state transition system is described.

First, necessary restrictions on the used semantics are explicated. However, the main focus lies on the computation of possible control- or objectflows between actions.

The naive idea, just to consider the direct successor of a node will not lead to success since the UML 2 defines a "traverse-to-completion" semantics for activity diagrams. This means, control or object tokens flow directly from one action to another one. It does not "stop" at any control node.

An exception of this rule is the fork node realizing a parallel split. This node buffers the control or object tokens if the subsequent action is not ready.

This "global" view on flows in activities leads to a complex computation. For this reason, the computation is done in two steps: First, only "simple flows" are collected. Simple flows are flows from one action to a buffer, from a buffer to an action or, if there is no buffer in-between, from an action to another action. In a second step, these flow fragments are combined to more complex flows.

The last section of this chapter deals with the correctness proof of the transformation.

In chapter 4 the abstract state transition system is prototypically translated into an input language of a model checker. The chosen input language is model programs, which can be verified by the model program checker.

This language is appropriate for two reasons: on the one hand, the mapping of abstract state transitions into model programs is straight forward. On the other hand, complex data structures are supported, which simplifies the translation. Possible alternatives are discussed as well.

Subsequently, the possibilities of model checking activity diagrams are exemplified by two case studies.

In the next section of this chapter environment modeling is discussed, which often is necessary for proper model checking. Some possibilities of how to optimize activity diagrams in order to reduce the state space are presented in the last section.

In chapter 5, the chosen approach is contrasted with alternative solutions. The restrictions are picked up again and several extensions for resolving those are presented.

Furthermore, this thesis is embedded in its scientific context. Related work is discussed and compared to the developed approach.

Chapter 6 summarizes the main contributions of this thesis. These include:

- Model checking of UML 2 activity diagrams for a more comprehensive subset than covered by previous approaches

- Formalization of the transformation algorithm with Abstract State Machines
- Consideration of semantic variation points
- Prototypical implementation

Finally, an outlook on further steps and research topics is given.

The appendix consists of a complete commented listing of all developed ASM-macros and a glossary of important terms.

Kapitel 1

Einleitung

Bereits in den 1960er Jahren erkannte man aufkommende Probleme der immer wichtiger gewordenen Softwareentwicklung und prägte den Begriff des dadurch notwendigen „Software Engineering“ [NR69, RB70]. Die Probleme der damaligen Softwarekrise waren hohe Softwareentwicklungskosten und fehlerhafte Systeme. Diese rührten daher, dass die bis dahin eingesetzten Techniken nicht mit der Entwicklung und den damit verbundenen Möglichkeiten der Hardware Schritt gehalten hatten. Edger W. Dijkstra fasste diesen Umstand in seiner Dankesrede zum Turing Award klar zusammen: „[...] *as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.*“ [Dij75]

Der Trend zu immer leistungsfähigeren Maschinen hält bis heute an [Moo06]. Dies ist unter anderem der Grund dafür, warum die Softwarekrise bis heute noch nicht als überwunden betrachtet werden kann, obwohl verschiedene Ansätze zur Etablierung „ingenieurmäßiger“ Vorgehensweisen entwickelt wurden. Diese Ansätze können in folgende drei Hauptrichtungen unterschieden werden:

- **Prozesse und Methoden**

Für eine Verbesserung der Softwareentwicklung wurden viele unterschiedliche Methoden vorgeschlagen, die verschiedene Aspekte wie Anforderungsanalyse, Entwurf, Implementierung, Qualitätssicherung und Wartung abdecken. Eine Zusammenstellung mehrerer Methoden in einer bestimmten Reihenfolge bildet einen Prozess.

- **Abstraktion**

Die zunehmende Abstraktion in der Softwareentwicklung einerseits von der verwendeten Hardware, andererseits von unterschiedlichen Konzepten wie Datenhaltung, Datenein- und -ausgabe, grafische Darstellung usw. ermöglicht immer komplexere Systeme, die auf immer komplexeren Komponenten aufbauen.

- **Test und Verifikation**

Die beiden vorherigen Punkte haben als Ziel Fehler*vermeidung*. Dennoch ist es unerlässlich, die Qualität der erstellten Systeme zu überprüfen. Dies ist zum einen via systematischen Tests möglich, womit man aber nur die Anwesenheit von Fehlern, nicht aber deren Abwesenheit zeigen kann [Dij75]. Zum anderen ermöglicht die (automatische) Verifikation den Beweis, dass ein System spezifizierten Eigenschaften entspricht.

Jedem dieser Punkte liegt der Versuch zugrunde, die wachsende Komplexität moderner Softwaresysteme zu beherrschen. Dies ist allerdings nur mit Hilfe entsprechend mächtiger Werkzeugunterstützung möglich.

Außerdem können diese unterschiedlichen Richtungen nicht getrennt voneinander gesehen werden: Tests und Verifikation müssen in den Entwicklungsprozess eingeplant werden. Sie können wiederum nur dann automatisiert durchgeführt werden, wenn ein zu einem gewissen Grad abstraktes Modell des Systems vorliegt.

1.1 Modellgetriebene Softwareentwicklung

Der oben erwähnte zweite Punkt, die Abstraktion, wird durch immer mächtigere Sprachen erreicht, die komplexe Zusammenhänge mit wenigen Elementen darstellen können. Häufig sind diese Sprachen grafisch notiert und bieten somit eine höhere Präzision und eine bessere Lesbarkeit als natürlichsprachlicher oder formaler Text.

Mit diesen Formalismen wird ein abstraktes Modell des zu erstellenden Systems erstellt. Dieses wird dann mit entsprechenden Werkzeugen möglichst automatisiert in eine konkrete lauffähige Anwendung übersetzt.

Dieser Ansatz, der unter der Bezeichnung Modellgetriebene Softwareentwicklung (engl. Model-Driven Software Development (MDSD)) [SVEH07, BBG05, PTN⁺07] zusammengefasst wird, erfuhr in den letzten Jahren eine zunehmende Bedeutung. Neben der Beherrschung der steigenden Komplexität erhofft man sich eine erhöhte Produktivität und Qualität. Dies wird durch eine bessere Wiederverwendbarkeit der erstellten Komponenten, die sich durch die Abstraktion und Automatisierung ergibt, erreicht.

Die Model-Driven Architecture-Initiative (MDA) [MDAa, KWB03] der Object Management Group (OMG)¹ ist eine konkrete Umsetzung dieses Ansatzes. Dabei werden plattformunabhängige Modelle (PIM) in plattformspezifische Modelle (PSM) und schließlich in Code automatisiert transformiert. Auch wenn dieser Ansatz die zu verwendende Notation offen lässt, schlägt die OMG die Unified Modeling Language (UML) [UML, RQZ07] als Modellierungssprache vor.

Die UML ist ein bekannter Vertreter der grafischen Notationen. Sie ist vor allem wegen der Integration verschiedener Diagrammart, welche unterschiedliche Blickwinkel auf ein System abdecken, weit verbreitet. Die aktuelle Version 2.2 definiert sechs Struktur- und sieben Verhaltensdiagramme. Die Spezifikation beschreibt diese Diagrammtypen nicht direkt. Sie ist stattdessen in sogenannte *Spracheinheiten* (engl. *language units*) unterteilt.

In diesen Einheiten werden eng zusammenhängende Elemente zusammengefasst, die zur Modellierung eines ausgewählten Systemaspekts in einem Formalismus verknüpft werden können. In der Spracheinheit *Aktivitäten* (engl. *Activities*) werden z. B. alle UML-Elemente zusammengefasst, die für die Modellierung von Systemverhalten mit Hilfe von Daten- und Kontrollflüssen bereitgestellt werden.

Die meisten Spracheinheiten sind wiederum in mehrere Schichten (engl. *compliance level*) unterteilt. Diese geben zunächst elementare und häufig verwendete und dann zunehmend komplexere Modellierungselemente an.

Für die Spracheinheit der Aktivitäten sind beispielsweise (in aufsteigender Reihenfolge) folgende Schichten definiert: *Fundamental Activities*, *Basic Activities*, *Intermediate Activities*,

¹<http://omg.org>

Complete Activities, Structured Activities, Complete Structured Activities, Extra Structured Activities (vgl. [UML, Seiten 295ff]).

Ein Hauptkritikpunkt gegenüber der UML ist die fehlende formale Semantik. Das Spezifikationsdokument der OMG ist rein natürlichsprachlich und damit weder eindeutig noch widerspruchsfrei oder vollständig. Da aber zum einen für den Austausch zwischen den Projektbeteiligten und zum anderen für die Modelltransformation eine präzise eindeutige Semantik notwendig ist, wurden in den letzten Jahren für verschiedene Teilaspekte der UML formale Semantiken definiert (siehe [BCR04, LLJ04, CK04, Stö05, SH05, SG07, Sar06a, FS07, Koh08, CD08, KG09, Koh09]).

1.2 Automatisierte Verifikation

Eine zugrunde liegende formale Semantik ist insbesondere auch für die automatisierte Verifikation eines Modells wichtig. Diese Verifikationsart wird unter dem Begriff *Model Checking* [CGP99] zusammengefasst. Sie ermöglicht den vollautomatischen Nachweis, ob ein System eine gewisse Eigenschaft erfüllt. Dabei wird der mögliche Zustandsraum des Systems durch ein Zustandsübergangssystem repräsentiert und die nachzuweisende Eigenschaft durch eine temporallogische Formel spezifiziert. Entsprechende Werkzeuge liefern dann ohne weitere Benutzerinteraktion den Nachweis, dass das System die spezifizierte Eigenschaft erfüllt oder — falls dies nicht zutrifft — ein Gegenbeispiel, dessen Ausführung die Eigenschaft verletzt.

Der entscheidende Vorteil dieser Methode gegenüber einfachem Testen ist der formale Nachweis, dass das System die Eigenschaft erfüllt und kein Fehler diesbezüglich enthalten ist. Dies wäre beim Testen nur möglich, wenn alle möglichen Testfälle abgedeckt werden würden, was in der Realität zu aufwändig ist. Andererseits erfordert die Verifikation mehr Aufwand in der Erstellung der formalen Modelle, welche aber bei Einsatz der modellgetriebenen Softwareentwicklung ohnehin erfolgt.

Eine entscheidende Rolle spielt die Verifikation bei der Entwicklung sicherheitskritischer Systeme. In Fällen, wo Menschenleben oder hohe Sachwerte von der korrekten Funktion eines Systems abhängen, wird sehr viel Aufwand getrieben, um die Fehlerfreiheit nachweisen zu können. So finden diese Verfahren bereits seit einigen Jahren in der Luft- und Raumfahrt und immer mehr auch im Automobilbereich Einsatz (vgl. [BBB⁺99, LNR⁺08, tBGKM08]).

1.3 Einbettung und Ziele dieser Arbeit

Als Beitrag zur ersten der drei oben vorgestellten Hauptrichtungen des Software Engineering „Prozesse und Methoden“, wurde am Institut für Programmiermethodik und Compilerbau der Universität Ulm eine Methode entwickelt, bei der UML 2 Aktivitätsdiagramme mit Code verknüpft werden können.

Der im Aktivitätsdiagramm modellierte Kontroll- und Datenfluss wird dabei direkt während der Ausführung eines Systems verwendet und ruft an den entsprechenden Stellen Klassenmethoden auf [SGK⁺05, SRKS05, SKRG07]. Damit muss der in Aktivitätsdiagrammen spezifizierte Kontrollfluss nicht mehr in eine Programmiersprache überführt werden.

Die für diese Vorgehensweise notwendige formale Spezifikation der Semantik der Aktivitätsdiagramme, welche in der Dissertation von Sarstedt [Sar06a] beschrieben ist, liefert einen Beitrag zur zweiten der oben vorgestellten Hauptrichtungen, der „Abstraktion“.

Die vorliegende Arbeit liefert nun einen Beitrag zur dritten Hauptrichtung „Test und Verifikation“: ein Verfahren zur automatischen Verifikation von UML 2 Aktivitätsdiagrammen.

Die Möglichkeit, Aktivitätsdiagramme zu verifizieren, ist eine wichtige Voraussetzung dafür, dass diese Diagrammart zur Erstellung sicherheitskritischer Systeme eingesetzt werden kann.

Mögliche Gründe, warum ein System mit Hilfe von Aktivitätsdiagrammen anstatt mit z. B. Zustandsautomaten modelliert werden soll, sind unter anderem die Kombination von Kontroll- und Objektflüssen im Aktivitätsdiagramm und die explizite Darstellung von für eine gewisse Zeitspanne andauernden Aktivitäten. Die OMG selbst schreibt: „*However, for many behaviors, the choice of specification mechanism is one of convenience.*“ [UML, Seite 422] Damit soll dem Modellierer die Wahl gelassen werden, welchen der vorgeschlagenen Diagrammtypen er zur Modellierung des Systems verwendet. Diese Wahl ist aber stark eingeschränkt, wenn es nur für einen kleinen Teil davon Werkzeugunterstützung gibt.

Darüber hinaus erlaubt es die UML, mehrere Beschreibungsarten zu verknüpfen, so dass beispielsweise eine Aktivität mit Hilfe eines Zustandsdiagramms dargestellt werden kann. Kohlmeyer beleuchtet in seiner Dissertation diesen Aspekt genauer [Koh09].

Schließlich ist die vorliegende Arbeit ein Beitrag dazu, dass Aktivitätsdiagramme in einem breiteren Feld eingesetzt werden können. Dies ist unter anderem deswegen besonders interessant, da Aktivitätsdiagramme in der Systems Modeling Language (SysML) [MDAb] eine zentrale Rolle spielen. Dies SysML basiert auf UML und wurde entworfen, "[...] *to provide simple but powerful constructs for modeling a wide range of systems engineering problems*" [MDAb, Seite 3].

Die Semantik der Aktivitätsdiagramme wurde in der Version 2 der UML komplett überarbeitet. Dadurch können die existierenden Ansätze des Model Checkings, welche auf zustandsorientierten Systemen aufbauen, nicht einfach wieder verwendet werden.

Um die rund 15-jährige Forschungsarbeit dieser Ansätze dennoch nutzen zu können, wird in dieser Arbeit eine semantikerhaltende Transformation der Aktivitätsdiagramme in Zustandsübergangssysteme definiert. Diese Zustandsübergangssysteme können von gängigen Model Checkern überprüft und die Ergebnisse wieder auf das ursprüngliche Diagramm zurückgespiegelt werden.

Der Schwerpunkt liegt dabei auf dem Einsatz von symbolischen Model Checkern (siehe Abschnitt 2.1.3.2), obwohl dafür einige Einschränkungen (siehe Abschnitt 3.1) in Kauf genommen werden müssen. Explizites Model Checking (siehe Abschnitt 2.1.3.1) hat den Nachteil, dass es nur für sehr kleine Systeme angewendet werden kann.

Die Übersetzung in Zustandsübergangssysteme wird allgemein definiert, so dass beliebige vorhandene Werkzeuge eingesetzt werden können. Als semantische Basis dient die formale Semantik aus [Sar06a]. Bei der Übersetzung werden auch „schwierigere“ Aspekte von Aktivitätsdiagrammen, wie Objektflüsse, Unterbrechungsbereiche und Signalverarbeitung unterstützt, welche bei bisherigen Ansätzen gar nicht oder nur rudimentär beachtet wurden.

Auch die in [Sar06a] definierten semantischen Variationspunkte werden berücksichtigt. Diese Variationspunkte ermöglichen es dem Anwender die Semantik einzelner Elemente punktuell zu modifizieren. Die Transformation wird mit dem Formalismus der Abstract State Machines [BS03] beschrieben.

1.4 Aufbau der Arbeit

Zunächst werden in Kapitel 2 die notwendigen Grundlagen erläutert. Neben den theoretischen Grundlagen des Model Checkings wird der Formalismus der Abstract State Machines und die Syntax und Semantik von UML 2 Aktivitätsdiagrammen beschrieben.

In Kapitel 3 wird die Transformation der Aktivitätsdiagramme in Zustandsübergangssysteme mit Hilfe der Abstract State Machines formalisiert.

Kapitel 4 stellt an Hand eines Beispiels die Übersetzung der Zustandsübergangssysteme in eine Eingabesprache für einen Model Checker vor. Anschließend werden Möglichkeiten der Umgebungsmodellierung, die häufig für effektives Model Checking notwendig ist und einige mögliche Optimierungen des Model Checkings von Aktivitätsdiagrammen betrachtet.

Der vorgestellte Ansatz wird in Kapitel 5 diskutiert und einige verwandte Arbeiten dargestellt. Schließlich gibt Kapitel 6 eine Zusammenfassung und einen Ausblick auf weitere Arbeiten, die auf dem vorgestellten Ansatz aufbauen können.

1.5 Notationelle Konventionen

Zur besseren Lesbarkeit werden folgende notationelle Konventionen getroffen:

- Der Begriff „UML“ bezieht sich im Weiteren auf die durch die Superstructure Specification in der Version 2.2 [UML] definierte Modellierungssprache.
- Die UML 2 Aktivitätsdiagramme werden nur mit „Aktivitätsdiagramme“ bezeichnet. Andere Versionen werden entsprechend kenntlich gemacht.
- Elemente des UML-Metamodells werden *kursiv* gesetzt.
- Begriffe der UML, die nicht Teil des Metamodells sind, werden in spitze Klammern gesetzt, z. B. \langle FundamentalActivities \rangle .

Tabelle 1.1 beschreibt die für diese Arbeit geltenden mathematischen Konventionen.

Tabelle 1.1: Mathematische Konventionen.

Operator	Beschreibung
Verschiedene	
$x.g$	Anwendung der Funktion g , äquivalent zu $g(x)$
Tupel-Operatoren	
$t.First$	liefert das erste Element eines Tupels t zurück
$t.Second$	liefert das zweite Element eines Tupels t zurück
Mengen-Operatoren	
$\cup, \cap, \in, \subset, \subseteq, \setminus, \times, \emptyset$	werden wie üblicherweise benutzt
$\bigcup X$	Vereinigung aller Mengen aus der Menge von Mengen X
$\mathcal{P}(S)$	Potenzmenge der Menge S
$\{ s \in S \mid P(s) \}$ oder $\{ F(s) \mid P(s) \}$	Mengenkomprehension, P ist ein Prädikat, F ist eine Funktion
$ S $	Anzahl der Elemente der Menge S
$asList(S)$	liefert eine Liste der in S enthaltenen Elemente in einer beliebigen, aber festen Reihenfolge zurück
Listen-Operatoren	
A^*	Listentyp
$L_1 \uplus L_2$	Konkatenation zweier Listen
$L \oplus e$	fügt Element e zur Liste L hinzu, äquivalent zu $L \uplus [e]$
$[l \in L \mid P(l)]$ oder $[F(l) \mid P(l)]$	Listenkompheension, P ist ein Prädikat, F ist eine Funktion
$l \in L$	prüft, ob l in der Liste L enthalten ist
$[]$	leere Liste
$ L $	Anzahl der Elemente der Liste L
$elementAt(L, i)$	liefert das i -te Element der Liste L
$first(L)$	liefert das erste Element der Liste L , äquivalent zu $elementAt(L, 1)$
$indexOf(L, e)$	liefert den Index des ersten Vorkommens von e in der Liste L zurück

Kapitel 2

Grundlagen

Dieses Kapitel beschreibt die für die weitere Arbeit notwendigen Grundlagen. Zunächst werden in Abschnitt 2.1 die theoretischen Grundlagen des Model Checkings erläutert. Danach wird in Abschnitt 2.2 der Formalismus der Abstract State Machines kurz beschrieben. Dieser Formalismus wird zum einen zur Semantikbeschreibung der Aktivitätsdiagramme in [Sar06a] verwendet, zum anderen wird der hier vorgestellte Übersetzungsansatz damit formalisiert. Abschnitt 2.3 führt neben der Syntax auch kurz in die in [Sar06a] beschriebene Semantik der Aktivitätsdiagramme ein, auf der die Übersetzung in Zustandsübergangssysteme aufbaut.

2.1 Model Checking

In diesem Abschnitt wird der Ansatz des Model Checkings in den Kontext der möglichen Qualitätssicherungsmaßnahmen eingebettet und die allgemeine Vorgehensweise erläutert. Für eine detaillierte Beschreibung wird auf [CGP99] und [BBF⁺01] verwiesen.

Prinzipielle Qualitätssicherungsmaßnahmen für komplexe Systeme sind Simulation, Test, deduktive Verifikation und Model Checking. Bei Simulationen und Tests werden ein Modell bzw. das fertige Produkt mit bestimmten Eingaben versehen und das Ergebnis mit der Spezifikation verglichen. Auf diese Art und Weise kann man bei der richtigen Wahl der Eingaben Fehler finden. Allerdings ist der Eingaberaum typischerweise zu groß, um alle möglichen Kombinationen abzudecken. Daher ist das Simulieren und das Testen eine relativ kosteneffiziente Möglichkeit, Fehler zu *finden*, jedoch kann damit nicht gezeigt werden, dass *keine Fehler mehr* in einem System sind. Für diese Aufgabe kommen die deduktive Verifikation und das Model Checking in Betracht.

Deduktive Verifikation ist ein meist sehr aufwändiger, händisch durchgeführter Vorgang, die Korrektheit eines Systems nachzuweisen. Ausgehend von einer formalen Beschreibung des Systems in einem Kalkül, wird über anzuwendende Regeln und einbezogene Axiome die Korrektheit bezüglich gegebener Eigenschaften abgeleitet.

Den verfügbaren maschinellen Theorembeweisern (z. B. [NPW02, Gor88, MKB95, ORS92]) ist gemein, dass sie bei der Beweisführung lediglich unterstützend helfen können. Diese Werkzeuge überprüfen die korrekte Anwendung der Regeln und machen eventuell Vorschläge für weitere anwendbare Regeln. Ein vollautomatischer Theorembeweiser für allgemeine Systeme ist auf Grund der allgemeinen Unentscheidbarkeit der Terminierung eines Programms (Halteproblem [Tur36]) nicht möglich [HMU02].

Model Checking hingegen beschränkt sich auf die Verifikation von nebenläufigen Systemen

die eine endliche Menge verschiedener Zustände annehmen können. Ein Vorteil dieser Einschränkung ist, dass die Verifikation vollautomatisch ablaufen kann. Dabei wird der komplette Zustandsraum aufgespannt und nach Zuständen, die der gegebenen Spezifikation entsprechen, durchsucht. Bei genügenden Ressourcen terminiert dieses Verfahren immer mit einer eindeutigen Antwort.

Die Einschränkung auf eine endliche Zustandsmenge reicht aus, um sehr viele Anwendungen automatisch zu verifizieren. Falls ein System unendlich viele Zustände annehmen kann, kann häufig mit Hilfe von Abstraktions- oder Induktionsmethoden ein endlicher Zustandsraum hergestellt werden. In vielen Fällen können Fehler gefunden werden, obwohl unbegrenzte Datenstrukturen auf eine endliche Anzahl von Instanzen reduziert werden. Zum Beispiel in Modellen mit unendlichen Eingabelisten können trotz Beschränkung der Listenlänge auf wenige Elemente Fehler gefunden werden.

Neuere Forschungsrichtungen verbinden die deduktive Verifikation mit den Vorteilen des Model Checkings, so dass endliche Teile eines komplexen Systems automatisch verifiziert werden können [NS07].

Model Checking läuft typischerweise in folgenden Schritten ab [CGP99]:

1. Modellierung des Systems
2. Spezifikation der zu prüfenden Eigenschaften
3. Automatische Verifikation mit Ergebnis

Jeder dieser Schritte wird nun in den folgenden Abschnitten eingehender betrachtet.

2.1.1 Modellierung des Systems

Um Model Checking durchführen zu können, muss ein Modell des Systems vorliegen bzw. erstellt werden. Die Detaillierung des Modells muss einerseits so gewählt sein, dass die zu prüfenden Eigenschaften noch abgedeckt sind. Andererseits soll das Modell von den Einzelheiten abstrahieren, die die Verifikation unnötig kompliziert machen.

Ein Hauptmerkmal eines Systems ist sein *Zustand*. Als Zustand eines Gesamtsystems wird die Menge aller Variablenwerte zu einem bestimmten Zeitpunkt definiert.

Das ebenso wichtige Merkmal der Zustandsänderung als Ergebnis einer Aktion des Systems beschreibt, wie das System von einem Zustand in einen anderen gelangt. Eine *Transition* ist ein Zustandspaar, welches den Zustand vor und nach einer Aktion beschreibt.

Eine *Ausführung* ist eine unendliche Sequenz von Zuständen, wobei jeder Folgezustand aus dem vorherigen Zustand durch eine Transition erhalten werden kann. Von einem Initialzustand ausgehend, kann das Verhalten eines reaktiven Systems auch nur mit Transitionen beschrieben werden.

Eine formale Definition dieser intuitiven Modellbeschreibung stellt die von Saul A. Kripke in [Kri63] eingeführte *Kripke-Struktur* dar. Diese Struktur, die man auch als Zustandsübergangsgraph bezeichnen kann, wurde ursprünglich verwendet, um eine Semantik für Modallogik und damit auch für Temporallogik [KK06] zu definieren (siehe 2.1.2). Sie ist folgendermaßen definiert [CGP99]:

Sei AP eine Menge von atomaren Aussagen. Eine *Kripke-Struktur* über AP ist ein Vier-tupel $M = (S, S_0, R, L)$ mit:

1. S ist eine endliche Menge von Zuständen

2. $S_0 \subset S$ ist eine Menge von Initialzuständen
3. $R \subset S \times S$ ist eine totale Transitionsrelation (zu jedem Zustand existiert ein Nachfolgezustand)
4. $L : S \rightarrow \mathcal{P}(AP)$ ist eine Abbildungsfunktion, die dem Zustand S eine nicht-leere Menge von atomaren Aussagen zuweist, die in diesem Zustand gelten.

Ein *Pfad* ausgehend von Zustand s ist eine unendliche Sequenz von Zuständen $\pi = s_0 s_1 s_2 \dots$ so dass $s_0 = s$ und $R(s_i, s_{i+1})$ für alle $i \geq 0$ gilt.

Diese universelle Zwischenstruktur ist die Basis für viele Model Checker und es existieren einige automatische Übersetzungen von anderen Beschreibungssprachen in diese Struktur.

Die vorliegende Arbeit definiert eine Übersetzung von Aktivitätsdiagrammen in diese Zwischenstruktur, so dass davon ausgehend beliebige Model Checker zu deren Verifikation verwendet werden können.

2.1.2 Spezifikation der zu prüfenden Eigenschaften

Nachdem das Modell des Systems in der gewünschten Granularität und in der vorgestellten Struktur definiert ist, muss noch die zu prüfende Eigenschaft formal definiert werden.

Da diese Eigenschaften typischerweise das dynamische Verhalten des Systems betreffen, muss der zeitliche Aspekt berücksichtigt werden. Dies könnte durch Einführung einer expliziten Zeit t in der Prädikatenlogik erster Stufe (engl. *first-order logic*) beschrieben werden. Allerdings führt dieses Vorgehen zu schwer verständlichen komplizierten Formeln.

Für diesen Zweck besser geeignet ist die Temporallogik. Pnueli war der erste, der diese Logik zur Beschreibung von Systemeigenschaften vorschlug [Pnu81]. Diese Logik kommt ohne die explizite Beschreibung der Zeit aus. Stattdessen werden neue zeitliche Operatoren eingefügt, die der natürlichen Sprache nachempfunden sind. Damit lassen sich Aussagen formulieren wie zum Beispiel: *irgendwann* wird ein bestimmter Zustand erreicht oder ein Fehlerzustand wird *nie* erreicht.

Es gibt verschiedene Varianten der Temporallogik. Im Folgenden wird nur die erweiterte *Computation Tree Logic* (CTL*) [EH86] näher betrachtet, da diese die *lineare temporale Logik* (LTL) [Pnu81] und die *Verzweigungslogik* (CTL) [BAMP81] einschließt [CD89]. Abb. 2.1 verdeutlicht, dass die Logiken LTL und CTL unterschiedliche Ausdrucksstärken besitzen, aber eine gemeinsame Schnittmenge haben und beide komplett in CTL* enthalten sind.

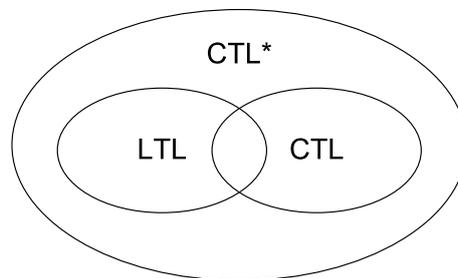


Abbildung 2.1: Mächtigkeit verschiedener Varianten der Temporallogik.

Im Rahmen dieser Einführung wird auf eine umfassende Darstellung der Semantik der CTL* verzichtet und stattdessen auf [CGP99] und [BBF⁺01] verwiesen. An Hand eines Beispiels wird ein grundlegendes intuitives Verständnis vermittelt.

Die CTL* trifft Aussagen über unendliche Ausführungs bäume. Diese Bäume entstehen, indem ein Zustand der Kripke-Struktur als Ausgangszustand definiert wird und ausgehend von diesem alle möglichen Ausführungen in einer Baumstruktur entfaltet werden. Abb. 2.2 zeigt auf der rechten Seite den Anfangsausschnitt des zum Zustandsübergangssystem links gehörigen Ausführungsbaums.

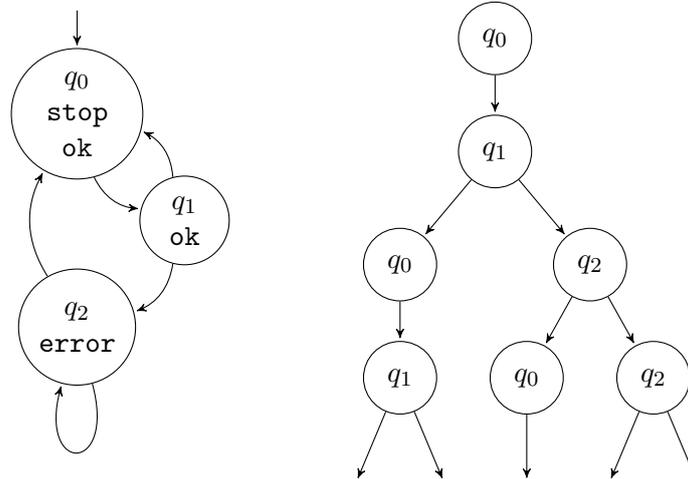


Abbildung 2.2: Beispiel für ein Zustandsübergangssystem und der Anfang des dazugehörigen Ausführungsbaums.

In der Temporallogik werden Zustände über logischen Aussagen beschrieben. Das System befindet sich in einem Zustand q , wenn die dazugehörige logische Aussage wahr ist. Um nun nicht nur Aussagen über einen einzelnen Zustand, sondern über eine ganze Sequenz von Zuständen treffen zu können, werden neue Operatoren eingeführt:

- X („next time“)

Falls P für eine Eigenschaft eines Zustands steht, dann bedeutet XP , dass der *nächste* Zustand P erfüllt. Betrachtet man die folgenden Ausführungspfade des Beispiels aus Abb. 2.2

$$\sigma_1 : (q_0 : \text{stop, ok}) \rightarrow (q_1 : \text{ok}) \rightarrow (q_0 : \text{stop, ok}) \rightarrow (q_1 : \text{ok}) \rightarrow (q_0 : \text{stop, ok}) \rightarrow \dots$$

$$\sigma_2 : (q_0 : \text{stop, ok}) \rightarrow (q_1 : \text{ok}) \rightarrow (q_2 : \text{error}) \rightarrow (q_0 : \text{stop, ok}) \rightarrow (q_1 : \text{ok}) \rightarrow \dots$$

$$\sigma_3 : (q_0 : \text{stop, ok}) \rightarrow (q_1 : \text{ok}) \rightarrow (q_2 : \text{error}) \rightarrow (q_2 : \text{error}) \rightarrow (q_2 : \text{error}) \rightarrow \dots$$

⋮

dann gilt beispielsweise für die Zustände der Ausführungen σ_1 , σ_2 und σ_3 : $\text{XX error} \vee \text{XX ok}$.

- F („in the future“)

FP beschreibt, dass es *irgendeinen* zukünftigen Zustand gibt, der P erfüllt. Für obiges Beispiel trifft zu, falls im aktuellen Zustand **stop** gilt, dann gibt es einen zukünftigen Zustand, indem **ok** gültig ist: $\text{stop} \Rightarrow F \text{ ok}$

- G („globally“)

GP fordert die Gültigkeit von P für *alle* zukünftigen Zustände. Die temporallogische

Formel $G(\text{stop} \Rightarrow F \neg \text{stop})$ drückt z. B. aus, dass es entlang einer Ausführung immer der Fall ist, dass, nachdem ein Zustand erreicht wurde, in welchem **stop** gilt, in einem zukünftigen Zustand (der sicher eintritt) dies nicht mehr gilt.

- U („until“)
Der zweistellige Operator U kann verwendet werden, um darzustellen, dass eine Eigenschaft bis zur Gültigkeit einer anderen Eigenschaft gilt. Für die Zustände der Ausführung σ_3 gilt beispielsweise, dass **ok** solange wahr ist bis **error** gilt: **ok U error**.
- R („release“)
Dieser Operator ist der logische Gegensatz zu U. In der Formel $Q R R$ gilt R solange, bis Q gilt oder für immer, falls Q nie gilt. **error R ok** gilt im obigen Beispiel.

Jeder dieser Operatoren geht von einem festen (aktuellen) Zustand aus. Von diesem Zustand ausgehend kann man also bestimmen, ob eine Formel gültig ist oder nicht.

Um Aussagen wie „in allen Zuständen gilt Formel F“ treffen zu können, werden die Operatoren A und E eingeführt. Diese Operatoren quantifizieren mehrere Ausführungspfade und werden daher auch *Pfadquantoren* genannt.

Die Formel $A\phi$ („für alle Ausführungspfade“) sagt aus, dass die temporallogische Formel ϕ für alle Ausführungspfade, die vom aktuellen Zustand ausgehen, erfüllt ist. Dementsprechend beschreibt $E\phi$ („für einige Ausführungspfade“), dass es mindestens einen Ausführungspfad gibt, auf dem ϕ erfüllt ist.

In der CTL* werden zwei Arten von Formeln unterschieden (vgl. [CGP99]):

- **Zustandsformeln** gelten in einem bestimmten Zustand und sind folgendermaßen definiert (AP sei die Menge der atomaren Aussagen einer Kripke-Struktur):
 - Wenn $p \in AP$, dann ist p eine Zustandsformel.
 - Wenn f und g Zustandsformeln sind, dann sind $\neg f$, $f \vee g$ und $f \wedge g$ Zustandsformeln.
 - Wenn f eine Pfadformel ist, dann sind Ef und Af Zustandsformeln.
- **Pfadformeln** sind entlang eines bestimmten Pfades gültig. Ihre Syntax ist folgendermaßen definiert:
 - Wenn f eine Zustandsformel ist, dann ist f auch eine Pfadformel.
 - Wenn f und g Pfadformeln sind, dann sind auch $\neg f$, $f \vee g$, $f \wedge g$, Xf , Ff , Gf , fUg und fRg Pfadformeln.

CTL* ist dann die durch diese Definition generierte Menge der Zustandsformeln.

Bei der Bildung von Formeln in der CTL* sind allerdings feine Unterschiede zu beachten: Die Formel $AF(EX \text{error})$ beschreibt z. B. für obiges Beispiel die (wahre) Tatsache, dass es für jeden möglichen Ausführungspfad irgendwann einen Zustand gibt, für den ein Folgezustand existiert, in welchem **error** wahr ist.

Diese Aussage ist nicht mehr gültig, wenn der Existenzquantor weggelassen wird. Denn es gibt mindestens einen Pfad, bei dem der Folgezustand nicht der Fehlerzustand ist, die neue Formel aber für alle Ausführungspfade gelten muss.

Das Fragment der CTL entsteht aus der CTL*, indem gefordert wird, dass jeder Temporaloperator unmittelbar durch genau einen Pfadquantor quantifiziert werden muss. LTL

hingegen ist die Teilmenge der Formeln ohne die beiden Pfadquantoren, wobei der Allquantor implizit jeder Formel vorangestellt wird.

Die Ausdrucksmächtigkeit der CTL* beschränkt sich (nach Definition) auf die Eigenschaften, die sich auf den Ausführungsbaum beziehen. Dementsprechend können stochastische Aspekte genauso wenig beschrieben werden wie realzeitliche. Zur Verifikation dieser Eigenschaften muss auf spezielle Model Checker, wie z. B. PRISM [HKNP06] oder CASPA [KS06] bzw. Rabbit [BLN03], KRONOS [Yov97] oder UPPAAL [BDL04] und deren speziellen Eingabesprachen zurückgegriffen werden.

Für die betrachteten (abstrakten) Automaten wird davon ausgegangen, dass keine Hardwaredefekte auftreten können und insofern von der Realität abstrahiert. Dennoch, so schließt [BBF⁺01], reicht die Mächtigkeit der CTL* zur Spezifikation der zu prüfenden Eigenschaften in den meisten Fällen aus. Das größere Problem ist aber, die korrekte Formel für einen zu überprüfenden Sachverhalt zu finden.

Erleichtert wird dies durch eine grobe Klassifikation häufig betrachteter Eigenschaften mit dafür typischen Formelmustern. Tabelle 2.1 zeigt die wichtigsten Klassen, deren Bedeutung und welche Operatoren eine dazu passende temporallogische Formel beinhaltet.

Noch einen Schritt weiter gehen *Property Specification Patterns*, wie in [DAC99] vorgestellt. Dabei werden in Anlehnung an Design Patterns [GHJ95] aus der objektorientierten Programmierung immer wieder auftretende Spezifikationsfälle abstrakt beschrieben. Nachdem das passende Muster gefunden wurde, kann dieses leicht mit den entsprechenden Zustandsbedingungen in der Eingabesprache des gewünschten Formalismus instantiiert werden.

Tabelle 2.1: Wichtigste Klassen von Systemeigenschaften.

Eigenschaft	Bedeutung	CTL*-Formel
Erreichbarkeit	eine bestimmte Situation kann erreicht werden	$EF P$
Sicherheit	unter gewissen Bedingungen tritt ein Zustand nie ein	$AG \neg P$
Lebendigkeit	unter gewissen Bedingungen wird etwas Bestimmtes schlussendlich eintreten	$AG(P_1 \Rightarrow AF P_2)$
Fairness	unter gewissen Bedingungen wird etwas Bestimmtes immer wieder geschehen	$AG(AF P)$

2.1.3 Automatische Verifikation

Mit dem formalen Modell und einer formalisierten zu prüfenden Spezifikation einer Eigenschaft kann das *Model Checking Problem* — zumindest formal — recht einfach definiert werden. Für eine gegebene Kripke-Struktur $M = (S, S_0, R, L)$ und eine temporallogische Formel f wird die Menge aller Zustände aus S gesucht, die f erfüllen:

$$\{s \in S \mid M, s \models f\}.$$

Wenn in dieser Menge alle Initialzustände aus S_0 enthalten sind, dann erfüllt das System die spezifizierte Eigenschaft. Die Frage ist nun, wie kann diese Menge berechnet werden?

Dazu gibt es zwei prinzipielle Vorgehensweisen: Man kann entweder den kompletten Zustandsraum aufspannen und dann über verschiedene Suchalgorithmen die entsprechenden Zustände suchen (explizites Model Checking) oder versuchen, ganze Zustandsmengen symbolisch zu beschreiben und nur auf diesen zu operieren (symbolisches Model Checking).

In den folgenden Abschnitten wird kurz auf die Vor- und Nachteile dieser beiden Verfahren eingegangen. Für weitere Details wird auf [CGP99] verwiesen.

2.1.3.1 Explizites Model Checking

Beim expliziten Model Checking wird zunächst der komplette Zustandsraum des Systems berechnet und als Graph dargestellt. Dies erfordert zum einen, dass das Modell endlich ist (nicht nur endlich beschreibbar) und zum anderen benötigt dieses Verfahren erheblichen Speicher für größere Systeme.

Zur Berechnung der Erfüllbarkeit einer temporallogischen CTL Formel f werden alle Zustände nach und nach mit den „von innen nach außen“ berechneten Teilformeln von f markiert, die in den entsprechenden Zuständen wahr sind. Jeder Zustand s wird somit mit einer Menge $label(s)$ markiert, die beschreibt, welche Teilformeln von f in diesem Zustand erfüllt sind. Nach der Terminierung des Verfahrens gilt für alle Zustände $s \in S$: $M, s \models f$ g. d. w. $f \in label(s)$. Als Zeitkomplexität ergibt sich für diesen Algorithmus: $O(|f| \cdot (|S| + |R|))$, wobei $|f|$ für die Anzahl der Teilformeln in f steht.

Zur Überprüfung, ob ein System einer LTL Spezifikation entspricht, wird häufig auf Büchi-Automaten [Büc62] zur Erkennung von Sprachen über unendlichen Wörtern (ω -reguläre Sprachen) zurückgegriffen. Dabei wird der Umstand ausgenutzt, dass für jede LTL Formel ϕ ein ω -regulärer Ausdruck angegeben werden kann, der die ϕ erfüllenden Ausführungspfade beschreibt.

Zur Berechnung wird ein Büchi-Automat konstruiert, der exakt die Ausführungen erkennt, die *nicht* ϕ erfüllen. Die Größe dieses Automaten ergibt sich im schlechtesten Fall mit $O(2^{|\phi|})$. Das Model Checking Problem reduziert sich dann auf die Frage, ob der Ursprungsautomat geschnitten mit dem konstruierten Büchi-Automat die leere Sprache beschreibt. Ist dies der Fall, erfüllt das Modell die Spezifikation.

CTL*-Formeln können nun durch eine Kombination dieser beiden Methoden überprüft werden. Interessanterweise konnte gezeigt werden [EL85, CES86], dass das Model Checking Problem für CTL* im Wesentlichen die gleiche Komplexität wie das LTL Model Checking, nämlich $O((|S| + |R|) \cdot 2^{O(|f|)})$, hat. Der Faktor $2^{O(|f|)}$ beschreibt dabei die maximale Größe des aus der temporallogischen Formel zu erstellenden Büchi-Automaten. Da diese aber typischerweise relativ klein sind, ist das Model Checking in der Praxis möglich.

Um dem Hauptnachteil des großen Speicherbedarfs, der notwendig ist, um den kompletten Zustandsübergangsgraph zu speichern, zu begegnen, werden Reduktionstechniken eingesetzt.

Zur Verifikation von asynchronen nebenläufigen Systemen eignet sich die *Partial Order Reduction*. Bei diesem Verfahren wird ausgenutzt, dass die parallele Ausführung unterschiedlicher Transitionen unabhängig von der Reihenfolge im letztendlich gleichen Zustand resultiert. Daher kann der Zustandsraum dahingehend reduziert werden, dass sich gleich verhaltende Transitionen durch einen Stellvertreter der entsprechenden Äquivalenzklasse ersetzt werden.

2.1.3.2 Symbolisches Model Checking

Verfahren, bei denen die komplette Berechnung des Zustandsraums vermieden wird, nennt man symbolisches Model Checking. Dabei werden Kripke-Strukturen durch *binäre Entscheidungsdiagramme* [Bry86] (engl. *Binary Decision Diagram* (BDD)) dargestellt. Im Folgenden wird diese Diagrammart kurz erläutert. Für eine ausführliche Beschreibung der BDDs wird auf [And99] verwiesen.

BDDs repräsentieren boolesche Funktionen in der **if-then-else**-Normalform (INF). Ein boolescher Ausdruck ist in der **if-then-else**-Normalform, wenn er nur aus **if-then-else**-Operatoren ($x \rightarrow y_0, y_1 = (x \wedge y_0) \vee (\neg x \wedge y_1)$) besteht. Dieser Operator zusammen mit den Konstanten **true** und **false** reicht aus, um alle anderen Operatoren darzustellen (z. B. $\neg x \equiv (x \rightarrow \text{false}, \text{true})$, $x \Leftrightarrow y \equiv x \rightarrow (y \rightarrow \text{true}, \text{false})$, $(y \rightarrow \text{false}, \text{true})$). Terme in INF können durch ein binäres Entscheidungsdiagramm dargestellt werden.

Ein binäres Entscheidungsdiagramm ist ein gerichteter azyklischer Graph mit

- einem oder zwei Terminalknoten ohne ausgehende Kanten mit der Bezeichnung 0 (**false**) oder 1 (**true**) und
- einer Menge von Variablenknoten v mit genau zwei ausgehenden Kanten. Jedem Variablenknoten ist eine Variable $var(v)$ zugeordnet.

Die beiden ausgehenden Kanten werden *Low*- und *High*-Kanten genannt und typischerweise mit gestrichelten bzw. durchgezogenen Linien dargestellt. Die *Low*-Kante entspricht dabei dem **else**-Zweig eines **if-then-else**-Operators, die *High*-Kante dem **then**-Zweig.

In der Grundform entspricht ein BDD einem einfachen Entscheidungsbaum. Äquivalente Teilbäume eines einfachen Entscheidungsbaums können zusammengefasst und unnütze Abfragen eliminiert werden. Dadurch erhält man mit linearem Aufwand ein reduziertes Entscheidungsdiagramm (engl. *reduced BDD*, kurz RBDD). Abb. 2.3 zeigt das bereits reduzierte Entscheidungsdiagramm für die Formel $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$.

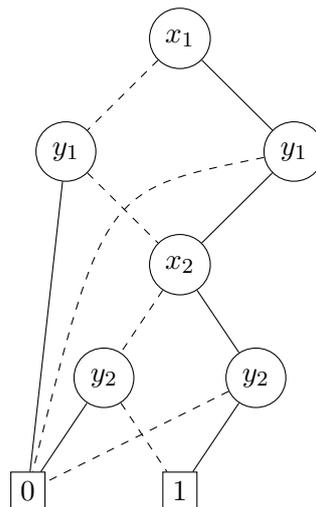


Abbildung 2.3: Ein RBDD für $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$.

Als *Variablenordnung* in einem (R)BDD wird die Reihenfolge der Variablen entlang eines Pfads bezeichnet. Die Variablenordnung für obiges Beispiel ist demnach $x_1 < y_1 < x_2 < y_2$.

Wenn alle Pfade eines (R)BDDs die gleiche Variablenordnung einhalten, spricht man von einem geordnetem (R)BDD (engl. ordered BDD, kurz (R)OBDD). Auf den ROBDDs können verschiedene Operationen (zum Beispiel Erfüllbarkeitsüberprüfung) sehr effizient ausgeführt werden.

Leider hängt die Größe der ROBDDs stark von der gewählten Variablenordnung ab, deren Optimumfindung NP-vollständig ist [Bry92]. Der ROBDD für das Beispiel aus Abb. 2.3 mit einer geänderten Variablenordnung sieht wie in Abb. 2.4 dargestellt aus.

Darüber hinaus gibt es boolesche Funktionen, deren ROBDDs exponentielle Größe für beliebige Variablenordnung haben. Ein Beispiel dafür ist bereits die arithmetische Multiplikation [Bry91]. Dennoch existieren verschiedene Heuristiken, um möglichst kleine ROBDDs zu finden.

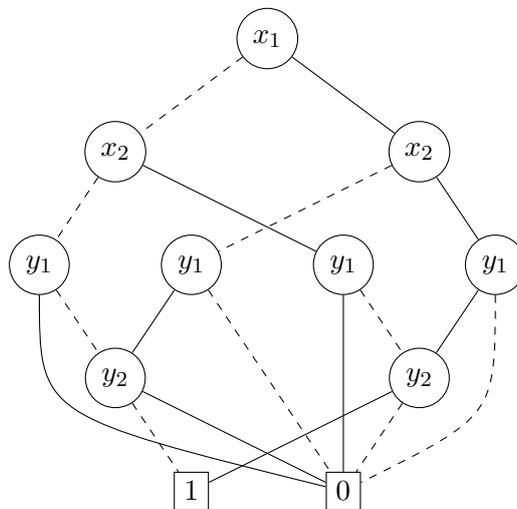


Abbildung 2.4: Das ROBDD für $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$ mit der Variablenordnung $x_1 < x_2 < y_1 < y_2$.

Die temporalen Operatoren müssen bei symbolischem Model Checking durch eine Fixpunktcharakterisierung beschrieben werden, da Operationen nicht mehr auf einzelne Zustände und Transitionen, sondern auf Mengen von Zuständen angewendet werden. Die Fixpunktbeziehung einer temporallogischen Formel auf dem entsprechenden ROBDD liefert dann das Ergebnis der Verifikation (siehe [CGP99]).

Eine verwandte Möglichkeit des symbolischen Model Checkings ist der Einsatz von *SAT-Solvern* [BCCZ99]. Diese Werkzeuge versuchen auf möglichst effiziente Art und Weise, die Erfüllbarkeit (engl. satisfiability) einer booleschen Formel zu zeigen oder zu widerlegen. Ausgehend von der Darstellung der Kripke-Struktur als boolesche Funktion, wie sie auch schon bei BDDs der Fall ist, kann diese Formel dann entsprechenden Werkzeugen zur Berechnung übertragen werden.

Dieser Ansatz wird in den letzten Jahren immer interessanter, da die Effizienz der SAT-Solver gestiegen ist und diese mit Hintergrundtheorien für z.B. Arithmetik und Mengen zu sogenannten SMT-Solvern (Satisfiability Modulo Theories) angereichert wurden [VBR08, AMP09].

Ein Nachteil dieses Ansatzes ist allerdings, dass häufig der zu überprüfende Bereich des

Zustandsraums künstlich beschränkt wird (Bounded Model Checking). Dadurch lässt sich die Fehlerfreiheit des betrachteten Systems nicht mehr vollständig nachweisen.

Auch wenn symbolisches Model Checking die Grenzen der automatischen Verifikation weiter verschoben haben, bietet es nur dann wirkliche Vorteile, wenn die zu untersuchenden Systeme damit kompakt dargestellt werden können. In Fällen, in denen dies nicht möglich ist, muss auf andere Techniken zurückgegriffen werden.

Bei der *zusammengesetzten Verifikation* (engl. compositional verification) werden beispielsweise unabhängige Teile eines Systems getrennt verifiziert und erst anschließend die Wechselwirkungen betrachtet.

Abstraktionstechniken reduzieren den möglichen Zustandsraum bereits vor dem Model Checking auf Modellierungsebene. Indem während der Modellierung des Systems von Einheiten oder komplexen Datentypen abstrahiert wird, wird der Zustandsraum klein gehalten.

Zum einen können Variablen, die nicht die in der Eigenschaftsspezifikation auftretenden Variablen beeinflussen, weggelassen werden. Zum anderen kann die Menge von konkreten, in einem System auftretenden Daten durch eine Untermenge von repräsentativen abstrakten Daten ersetzt werden.

Abschließend sei vermerkt, dass auf dem Gebiet des Model Checkings — obwohl es bereits seit einigen Jahrzehnten vor allem im Bereich der Hardwareentwicklung erfolgreich praktiziert wird — immer noch viel Forschungsarbeit geleistet wird. Insbesondere in Richtung automatischer Überprüfung von Software, was sich auf Grund der vielfältigeren Möglichkeiten erheblich komplexer als Hardwareüberprüfung darstellt.

2.2 Abstract State Machines

Der Formalismus der Abstract State Machines (ASM) wird in dieser Arbeit dazu verwendet, die Transformation der Aktivitätsdiagramme in Zustandsübergangssysteme formal darzustellen. Darüber hinaus werden die ASMs in der Semantikdefinition von Sarstedt [Sar06a], auf der die Transformation aufbaut, eingesetzt.

Dieser Abschnitt führt kurz in die wichtigsten, für das Verständnis der Transformation notwendigen Konzepte der ASMs ein. Für eine ausführliche Beschreibung der zugrunde liegenden formalen Semantik wird auf [BS03] verwiesen. Am Schluss werden die verwendeten Operatoren tabellarisch aufgelistet.

2.2.1 Hintergrund der Abstract State Machines

Gurevich begann 1984 [Gur84] abstrakte Maschinen zu suchen, die besser für die Simulation von Algorithmen geeignet sind als Turingmaschinen. Konkret wollte er vermeiden, dass die Simulation eines Algorithmus auf einer weniger abstrakten Ebene abläuft als der Algorithmus selbst. Dies resultiert nämlich darin, dass für die Durchführung eines Schritts des Algorithmus unter Umständen sehr viele Schritte auf der verwendeten Maschine auszuführen sind. Stattdessen wollte er eine Art „Gleichschritt“ (lock-step) des Algorithmus und der Maschine.

Um dieses Ziel zu erreichen, führte er *dynamische Strukturen* [Gur85] ein, deren Bezeichnung sich über *evolving algebras* [Gur95] zu *Abstract State Machines* [BS03, Referenz 191] weiterentwickelte. ASMs kann man als „pseudo-code over abstract data“ [BS03, S. 13] betrachten, der auf verschiedenen Abstraktionsebenen eingesetzt und weiter verfeinert werden kann. Dies ist möglich, da nicht nur die Regeln verfeinert werden, sondern auch die Interpretation der Daten entsprechend angepasst wird.

2.2.2 Sequentielle Abstract State Machines

Obwohl mittlerweile verschiedene Erweiterungen (siehe Abschnitt 2.2.3) verfügbar sind, wird in diesem Kapitel zunächst eine intuitive Beschreibung der sogenannten *sequentellen ASMs* gegeben, die sich an die Beschreibung aus [BS03] anlehnt.

Sequentielle ASMs sind eine endliche Menge von Transitionsregeln die abstrakte Zustände transformieren und folgendermaßen dargestellt werden:

if *Condition* **then** *Updates*

Die Bedingung *Condition*, unter der eine Regel angewendet wird, ist eine beliebige prädikatenlogische Formel ohne freie Variablen. *Updates* ist eine Menge von Zuweisungen der Form

$$f(t_1, \dots, t_n) := t,$$

deren Ausführung eine Änderung oder Neudefinition (falls vorher undefiniert) der Funktionen an den angegebenen Stellen darstellt.

Einen Funktionsnamen f , der durch die Signatur der Funktion eindeutig ist, bezeichnet man zusammen mit konkreten Werten für die Parameter t_1, \dots, t_n als *Location (loc)*. *Location-value*-Paare (loc, v) werden *updates* genannt und bilden die elementare Einheit einer Zustandsänderung. Diese abstrakte Sichtweise auf Speicher und dessen Aktualisierung erlaubt die im vorherigen Abschnitt erwähnte Anpassung an die zu einem bestimmten Zeitpunkt gewünschte Abstraktionsebene.

ASM *Zustände* werden als mathematische Strukturen dargestellt, wobei Daten (Objekte) als Elemente von Mengen beschrieben werden. Diese Mengen, die auch als *Domänen* oder *Universen* bezeichnet werden, sind mit grundlegenden Operationen (partielle Funktionen) und Prädikaten (Attribute und Relationen) verknüpft.

Die Instantiierung einer Relation auf ein Objekt o kann durch die „Punktschreibweise“ $o.f(x)$ beschrieben werden, was einer Parametrisierung von $f(o, x)$ entspricht. 0-stellige Funktionen werden auch als Konstanten bezeichnet. Partielle Funktionen werden durch einen festen vorgegebenen Wert `undef` (der per Definition Element aller Domänen ist) in totale Funktionen überführt: $f(x) = \text{undef}$ für alle Stellen x , an denen $f(x)$ nicht definiert ist. Für die Auswertung der Terme und Formeln in einem ASM Zustand wird die übliche Interpretation der Funktionssymbole verwendet.

Ein Berechnungsschritt von ASMs besteht aus der *gleichzeitigen* Anwendung aller Updates von allen Transitionsregeln, deren Bedingung im aktuellen Zustand wahr ist.

Eine Menge von Updates ist *inkonsistent*, wenn darin zwei Updates enthalten sind, welche die gleiche Location mit unterschiedlichen Werten verändern wollen $((loc, v), (loc, v')$ mit $v \neq v')$. Ist dies der Fall, wird überhaupt kein Update angewendet. In diesem Sinne verhalten sich ASMs also wie Datenbanktransaktionen: Sie werden wie eine atomare Aktion ohne Seiteneffekte durchgeführt.

Im Allgemeinen führen ASMs immer wieder einen Berechnungsschritt durch ohne sich dabei zu beenden. Für den Fall, dass die Terminierung gewünscht wird, können verschiedene Kriterien wie „Es ist keine Regel mehr anwendbar“, „Eine leere Updatemenge wird erzeugt“ oder „der Zustand ändert sich nicht mehr“ dafür verwendet werden.

Obwohl prinzipiell keine Einschränkungen über die Bedeutung der Funktionen gemacht werden, so hat sich im Laufe der Zeit herausgestellt, dass eine Einteilung der Funktionen gemäß deren Nutzung hilfreich ist. Abb. 2.5 zeigt eine typische Klassifikation wie sie in [BS03] vorgestellt wird.

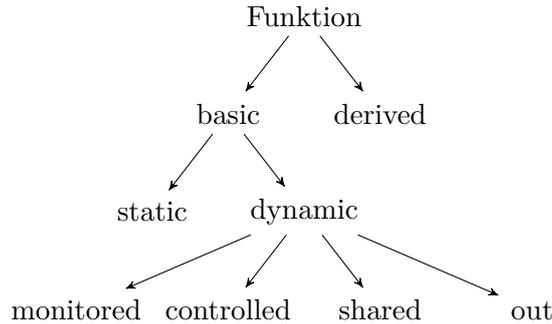


Abbildung 2.5: Klassifikation von ASM Funktionen.

Die Unterscheidung in *elementare (basic)* und *abgeleitete (derived)* Funktionen, beschreibt den Unterschied zwischen grundlegend definierten Funktionen und solchen, die unter Verwendung anderer (elementarer) Funktionen komplexere Rückgabewerte berechnen.

Abgeleiteten Funktionen kann nicht direkt ein neuer Wert zugeordnet werden. Lediglich durch Veränderung der darin verwendeten Funktionen können neue Werte erhalten werden. Auch *statischen (static)* Funktionen kann kein neuer Wert zugewiesen werden. Sie halten ihren Wert während der Ausführung einer ASM immer konstant.

Die *dynamischen (dynamic)* Funktionen werden in vier Unterkategorien unterteilt: Als *überwachte (monitored)* Funktionen werden Funktionen bezeichnet, auf die innerhalb der ASM nur lesend zugegriffen wird. Dennoch kann sich deren Wert z. B. durch die Umgebung (oder einen anderen Agenten, siehe Abschnitt 2.2.3) jederzeit ändern.

Im Gegensatz dazu stehen die *ausgehenden (out)* Funktionen, die nur geschrieben, aber nie gelesen werden. *Kontrollierte (controlled)* Funktionen können sowohl gelesen, als auch geschrieben werden, jedoch nur innerhalb der ASM, in der sie definiert sind. *Geteilte (shared)* Funktionen werden von mehreren Agenten (und/oder der Umgebung) gelesen und geschrieben und daher ist es meist erforderlich, den Zugriff über ein Protokoll zu regeln, um inkonsistente Updatemengen zu vermeiden.

2.2.3 Erweiterungen

Die im vorherigen Abschnitt eingeführten sequentiellen ASMs wurden nach und nach erweitert. Zum einen wurden vereinfachende Schreibweisen für häufig vorkommende Situationen definiert, zum anderen wurden immer mächtigere Konstrukte hinzugefügt und in die ursprüngliche Theorie eingebettet.

Die **skip**-Regel beschreibt die leere Updatemenge. $R_1 \mathbf{par} \dots \mathbf{par} R_n$ erlaubt die Kombination von mehreren parallel ausgeführten Regeln, **forall** $x \mathbf{with} \varphi \mathbf{do} R$ wendet die Regel R für jedes x an, für das φ gilt.

choose $x \mathbf{with} \varphi \mathbf{do} R$ fügt den ASMs Nichtdeterminismus hinzu, indem die Regel R für ein zufällig gewähltes x , welches der Bedingung φ entspricht, ausgeführt wird. Zur Vereinfachung schreibt man häufig z. B. **forall** $x \in X \mathbf{with} \varphi \mathbf{do} R$ statt **forall** $x \mathbf{with} x \in X \wedge \varphi \mathbf{do} R$.

Für die Erweiterung des Speichers einer ASM wird das Konzept der *Reserve* eingeführt. Die Reserve ist eine unendliche Menge aus der beliebige Elemente entnommen werden können, die alle paarweise unterschiedlich sind.

Die Regel **import** $x \mathbf{do} R$ importiert ein neues x aus der Reserve und führt damit R aus.

Eine alternative Schreibweise **let** $x = \text{new}(X)$ **in** R weist dem neuen Element x gleichzeitig die Domäne X zu (x ist damit vom „Typ“ X).

Strukturierte ASMs [BS03, Kapitel 4] fügen die Möglichkeit hinzu, innerhalb eines Berechnungsschritts mehrere „micro-steps“ auszuführen. Daher wird diese Klasse auch Turbo-ASM genannt, da mehrere Mikro-Schritte in einen ASM-Schritt komprimiert werden.

Erreicht wird dies durch Einführung des **seq**-Operators, der eine sequentielle Ausführung mehrerer Regeln erzwingt (der Standard ist die parallele Ausführung). Das **iterate**-Konstrukt erlaubt die Ableitung der klassischen Iterationsoperatoren wie z. B. **while**. Außerdem definieren Turbo-ASMs Rekursion über ASM-Regeln (auch Makros genannt) und ermöglichen die Rückgabe der Ergebnisse.

Zur Darstellung verteilter Algorithmen, die von mehreren sogenannten *Agenten* parallel ausgeführt werden, wurden die *verteilten* (engl. *distributed*) *ASMs* [BS03, Kapitel 5 und 6] entwickelt. Damit lassen sich verschiedene Ausführungen von ASM-Regeln auf mehrere Agenten verteilen und sogar während der Laufzeit neue Agenten erzeugen, denen dann dynamisch neue Regeln zugewiesen werden können.

Die Turbo-ASMs reichen für die Beschreibung der Transformation von Aktivitätsdiagrammen in Zustandsübergangssysteme aus. Verteilte ASMs finden lediglich in der Beschreibung der Semantik von Aktivitätsdiagrammen Verwendung. Daher wird für eine detailliertere Betrachtung der verteilten Multi-Agenten ASMs auf [Sar06a] verwiesen.

2.2.4 ASM-Konstrukte

Zum leichteren Verständnis der in Kapitel 3 beschriebenen ASM-Regeln, werden die verwendeten ASM-Konstrukte mit ihrer Bedeutung in Tabelle 2.2 aufgelistet.

Für die weitere Arbeit werden folgende Schreibkonventionen bezüglich der ASM-Regeln getroffen: **Schlüsselwörter** werden fett, *Domänen* kursiv blau, *Funktionen* kursiv grün, **MAKROS** orange und in Kapitälchen, temporäre Variablen normal, *Parameter* kursiv und Konstanten ohne Serifen gedruckt.

2.3 UML 2 Aktivitätsdiagramme

Dieser Abschnitt führt in die Aktivitätsdiagramme ein. Dabei wird zum einen auf die Syntax dieser, in UML 2 komplett überarbeiteten, Diagrammart eingegangen, zum anderen auch informell deren Semantik dargestellt.

Die ausführliche Beschreibung der Syntax und eine textuelle Beschreibung der Semantik findet sich in der Spezifikation der UML [UML]. Eine ausführliche formale Semantikdefinition gibt Sarstedt in [Sar06a]. Neben diesen eher technischen Dokumenten geben die Bücher [RQZ07] und [Kec06] einen guten Überblick über die Verwendung und Einsatzgebiete der UML.

Wie in Abschnitt 1.1 bereits erwähnt, definiert die UML verschiedene Schichten, die die Darstellungsmöglichkeiten der Aktivitätsdiagramme nach und nach erweitern (siehe [UML, Seite 295f]).

⟨FundamentalActivities⟩ definieren für Aktivitäten lediglich einfache Aktionen. Das Paket ⟨BasicActivities⟩ baut auf den ⟨FundamentalActivities⟩ auf und fügt Kontroll- und Datenflüsse zwischen Aktionen hinzu, die durch die in den ⟨IntermediateActivities⟩ definierten Kontrollknoten gesteuert werden können. Komplexere Konstrukte wie Unterbrechungsbereiche und Streaming wird in dem Paket ⟨CompleteActivities⟩ definiert.

Tabelle 2.2: Für die Arbeit relevante ASM-Konstrukte.

Konstrukt	Bedeutung
true, false, undef	Für alle ASMs vordefiniert.
skip	Führt die leere Regel aus und führt zu einer leeren Updatemenge.
if <i>condition</i> then <i>Q</i> else <i>R</i>	Falls <i>condition</i> wahr ist, wird <i>Q</i> ausgeführt, sonst <i>R</i> .
<i>Q</i> seq <i>R</i>	Führt zunächst <i>Q</i> , dann <i>R</i> mit einer (Zwischen-)Updatemenge von <i>Q</i> aus. Die resultierende Updatemenge für den neuen ASM-Zustand ergibt sich aus der Kombination $Q \oplus R$ der beiden Updatemengen, siehe [BS03].
MACRO (t_1, \dots, t_n)	Ruft die Regel MACRO mit den Parametern t_1, \dots, t_n auf.
result := <i>expr</i>	Liefert <i>expr</i> als Ergebnis des aktuellen Makros zurück.
iterate <i>R</i>	Führt <i>R</i> immer wieder aus, bis die Updatemenge von <i>R</i> leer ist.
domain <i>X</i>	Führt eine neue Domäne <i>X</i> ein.
local $x : X := \textit{expr}$	Definiert eine lokale Variable <i>x</i> aus der Domäne <i>X</i> (optional) und weist ihr <i>expr</i> zu.
let $x = \textit{expr}$ in <i>R</i>	Bindet die Variable <i>x</i> an <i>expr</i> für die Regel <i>R</i> .
let $x = \textit{new}(X)$ in <i>R</i>	Fügt ein neues Element aus der Reservemenge [BS03] in das Universum <i>X</i> ein und bindet es an die Variable <i>x</i> in der Regel <i>R</i> .
forall x with φ <i>R</i>	<i>R</i> wird für alle Elemente <i>x</i> , die φ erfüllen, parallel ausgeführt.
foreach x with φ <i>R</i>	<i>R</i> wird für alle Elemente <i>x</i> , die φ erfüllen, sequentiell ausgeführt.
choose x with φ <i>R</i>	Ein Element <i>x</i> , welches φ erfüllt, wird nichtdeterministisch ausgewählt und damit <i>R</i> ausgeführt. Falls kein solches Element existiert, ist die Updatemenge leer. Falls mindestens ein Element <i>x</i> φ erfüllt, wird dieses auch gewählt (<i>angelic choice Operator</i> , siehe [WM97, BS03]).
add x to <i>S</i> , remove x from <i>S</i>	Das Element <i>x</i> wird der Menge <i>S</i> hinzugefügt bzw. entnommen. Dieser Befehl kann parallel zu anderen add/remove -Befehlen ausgeführt werden ohne zu einer inkonsistenten Updatemenge zu führen (siehe [GT01]).
add <i>X</i> to <i>S</i> , remove <i>X</i> from <i>S</i>	Erweitert das normale add/remove um die Möglichkeit <i>alle Elemente aus der Menge X</i> hinzuzufügen bzw. zu entnehmen.

Orthogonal zu diesen vier Ebenen gibt es `StructuredActivities`, `CompleteStructuredActivities` und `ExtraStructuredActivities`, die die Unterstützung für Konstrukte aus der strukturierten Programmierung und Ausnahmenbehandlung definieren.

In dieser Arbeit werden die Pakete der Aktivitäten unterstützt, die auch in [Sar06a] Verwendung finden. Konkret ist dies das Paket der `IntermediateActivities`, wobei auch einige interessante Konstrukte wie z. B. Unterbrechungsbereiche aus den `CompleteActivities` hinzugenommen wurden. Da für viele weitergehende Elemente (wie z. B. Streams) bisher keine präzise, geschweige denn formale Semantik definiert wurde, ist es derzeit nicht möglich diese beim Model Checking zu berücksichtigen.

Der Einsatz der `StructuredActivities` wird nicht für sinnvoll erachtet, da sie die Ziele Abstraktion, Übersichtlichkeit und bessere Verständlichkeit der modellgetriebenen Softwareentwicklung konterkarieren (siehe dazu auch [SRKS05]).

2.3.1 Syntax

Die Beschreibung der Syntax der Aktivitätsdiagramme beschränkt sich hier auf die Elemente, die auch bei der Transformation berücksichtigt werden. Für die Bezeichner der Elemente werden durchgehend die englischen Begriffe verwendet, da sie geläufiger und meist kürzer sind als deren deutsche Übersetzung.

In der UML Spezifikation wird die abstrakte Syntax von Aktivitätsdiagrammen durch ein Metamodell beschrieben. Abb. 2.6 wurde aus [Sar06a] angepasst und zeigt den Ausschnitt aus dem gesamten Metamodell, der für die Verifikation von Aktivitätsdiagrammen betrachtet wird.

Zur detaillierten Beschreibung der Elemente des Metamodells und deren Bedeutung wird auf [Inf] und [MOF] verwiesen. Zur Erläuterung sei nur angemerkt, dass das Metamodell wie ein UML Klassendiagramm zu lesen ist. Es beschreibt die Beziehung und Kardinalitäten zwischen Elementen.

Eine *Activity* kann z. B. beliebige *ActivityNodes* besitzen, die sich wiederum aufspalten in *ExecutableNodes*, *ObjectNodes* und *ControlNodes*. Außerdem werden die Vererbungshierarchie und die Attribute der Elemente definiert.

Die graphische Darstellung der Elemente in einem Diagramm wird in der Spezifikation nicht näher beschrieben. Die UML lässt diesen Aspekt bewusst flexibel, damit eine beliebige (vielleicht in der entsprechenden Domäne bereits etablierte) Notation verwendet werden kann.

Allerdings macht sie durchaus Vorschläge wie die Elemente aussehen könnten und auf diese Beschreibung wird in dieser Arbeit zurückgegriffen. Die vorgeschlagenen Notationen finden sich in den folgenden Auflistungen immer neben den Bezeichnungen.

In Aktivitätsdiagrammen können sowohl Kontroll- als auch Objekt- bzw. Datenflüsse zwischen Aktionen beschrieben werden. In einer Aktion wird Verhalten zusammengefasst, welches in diesem Diagramm nicht weiter verfeinert wird. Zusätzlich zu dieser allgemeinen Art von Aktionen stehen Spezialisierungen zur Verfügung, die z. B. Signale empfangen oder senden können.

Alle Aktionen können mit einem oder mehreren *Input-* oder *OutputPins* als Ein- bzw. Ausgänge für Objektflüsse versehen werden. Alternativ dazu können Objektflüsse auch über Objektknoten, die auf den Kanten zwischen Aktionen liegen und den entsprechenden Datentyp spezifizieren, modelliert werden. Eine Spezialisierung davon ist der *CentralBufferNode*, der die Daten puffert und es dadurch ermöglicht, asynchron auf die eingehenden Daten zuzugreifen.

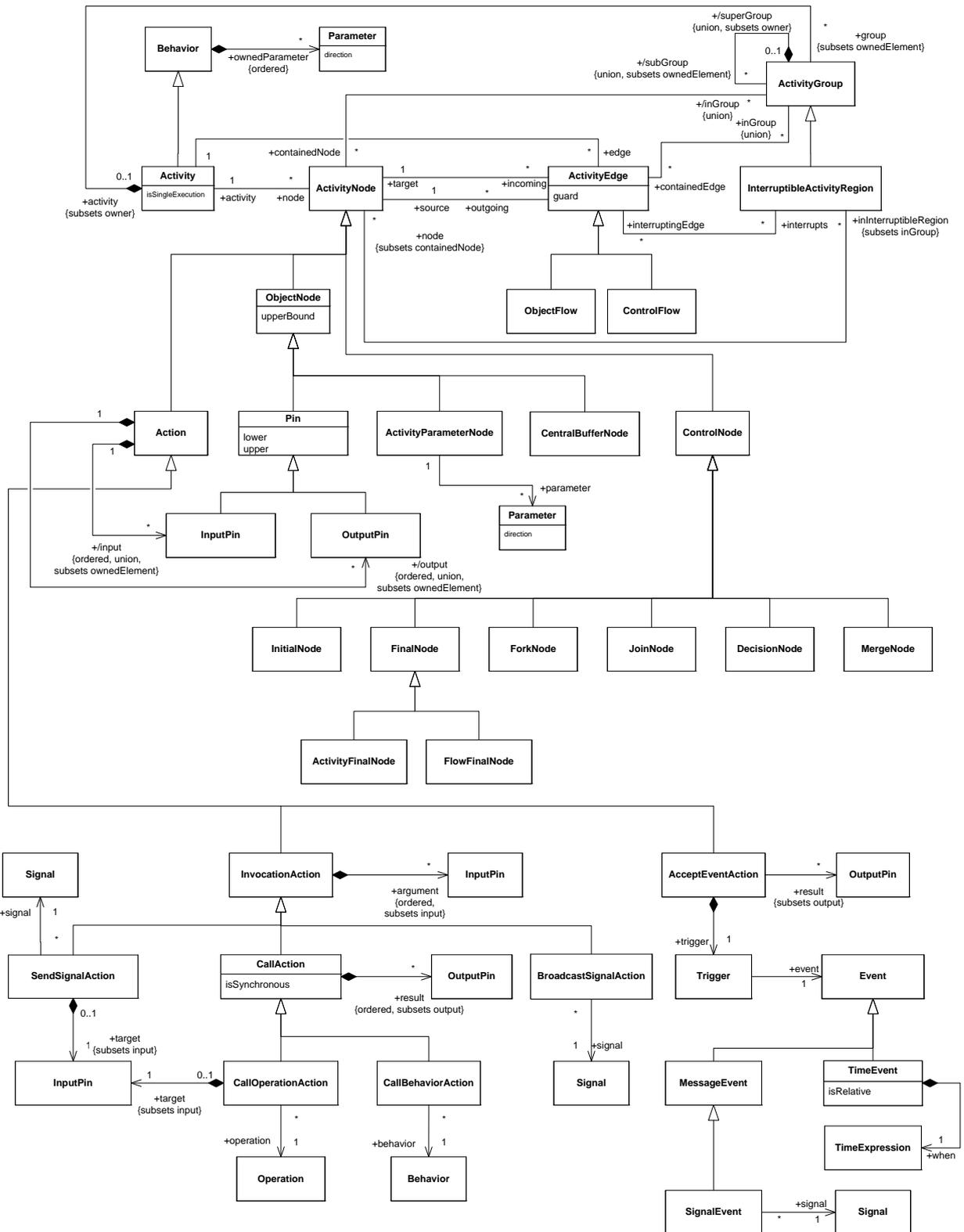


Abbildung 2.6: Ausschnitt aus dem Metamodell der UML 2, der in dieser Arbeit betrachtet wird (angepasst aus [Sar06a]).

Folgende Arten von Aktionen werden unterstützt:

- *CallOperationAction* 

Dies stellt eine elementare Aktion dar, die entweder nicht näher spezifiziert oder durch eine plattformsspezifische Implementierung realisiert wird.
- *CallBehaviorAction* 

Mit einer *CallBehaviorAction* kann ein anderes Aktivitätsdiagramm aufgerufen werden, welches das Verhalten dieser Aktion verfeinert. Die Darstellung unterscheidet sich von einer *CallOperationAction* dadurch, dass eine „Gabel“ rechts unten eingefügt wird, die eine Hierarchisierung andeuten soll. Die *Input-* bzw. *OutputPins* werden dabei auf *ActivityParameterNodes* abgebildet.
- *SendSignalAction* 

Mit dieser Aktion können Signale (*SignalEvents*) versendet werden. Signale sind einfache Ereignisse, die auch mit angehängten Daten versehen werden können. Um ein Signal an mehrere Empfänger gleichzeitig zu senden, stehen *BroadcastSignalActions* zur Verfügung. Die Bestimmung der bzw. des Empfänger(s) ist ein expliziter Variationspunkt der UML und wird in der Spezifikation offen gelassen.
- *AcceptEventAction* 

Diese Aktionsspezialisierung kann die von *SendSignalActions* ausgehenden Signale empfangen. Falls Daten übertragen wurden, können diese über einen *OutputPin* an die nächste Aktion weitergereicht werden. *AcceptEventActions*, die, statt auf Signale, auf absolute oder relative Zeitereignisse (*TimeEvents*) reagieren, werden durch eine stilisierte Sanduhr dargestellt.

Neben diesen ausführbaren Knoten (*ExecutableNodes*) gibt es Kontrollknoten (*ControlNodes*), die den Kontroll- bzw. Objektfluss zwischen Aktionen steuern bzw. Start- und Endpunkte für Aktivitäten vorgeben. Folgende Kontrollknoten sind für Aktivitätsdiagramme definiert:

- *InitialNode* 

Der Startpunkt einer Aktivität wird mit einem (oder mehreren) *InitialNodes* markiert.
- *FlowFinalNode* 

FlowFinalNode beendet nur den aktuellen Fluss. Das anliegende Token wird vernichtet.
- *ActivityFinalNode* 

FlowFinalNode beendet nicht nur den aktuellen Fluss, sondern die komplette Aktivität. Falls diese von einer *CallBehaviorAction* aufgerufen wurde, wird dorthin zurückgesprungen.
- *DecisionNode* 

An einem *DecisionNode* wird zwischen den ausgehenden Kanten in Abhängigkeit der an den Kanten annotierten Bedingungen (*Guards*) gewählt. Der Kontroll- bzw. Datenfluss wird nur über die eine ausgewählte Kante weitergereicht.
- *MergeNode* 

Das Gegenstück zum *DecisionNode* fügt mehrere Flüsse zu einem zusammen. Alle eingehenden Flüsse werden sequentiell an der ausgehenden Kante weitergeführt.

- *ForkNode* 

Der *ForkNode* spaltet den Kontroll- bzw. Datenfluss in mehrere parallel ablaufende Flüsse auf.
- *JoinNode* 

Die, z. B. durch einen *ForkNode*, aufgeteilten, parallelen Flüsse werden an diesen Knoten synchronisiert. Das heißt, der Fluss kann erst weiter fließen, wenn alle dem *JoinNode* vorgelagerten Aktionen beendet sind.

Unterbrechungsbereiche (*InterruptibleActivityRegions* ) markieren Bereiche, die beliebige Knoten beinhalten können. Falls dieser Bereich mit einer Unterbrechkante (*interrupting edge* ) verlassen wird, werden alle Aktionen die sich innerhalb dieses Bereichs befinden, abgebrochen.

Die aus [Sar06a] übernommene Abb. 2.7 gibt einen Überblick, wie die beschriebenen Elemente in einem Diagramm verwendet werden.

2.3.2 Semantik

Auch wenn im vorherigen Abschnitt teilweise schon die Semantik der einzelnen Elemente oberflächlich dargestellt wurde, wird nun etwas ausführlicher die Semantik der Aktivitätsdiagramme, wie sie in [Sar06a] formal definiert wird, beschrieben. Besondere Details werden allerdings erst in Kapitel 3 während der Erläuterung der Transformationsregeln behandelt.

Gemäß der UML Spezifikation ist die Semantik der Aktivitätsdiagramme „Petri-like“ [UML, Seite 324], was sich im Wesentlichen darin widerspiegelt, dass *Tokens* (Kontroll- bzw. Datentokens) durch das Diagramm fließen und das Zusammenspiel der einzelnen Aktionen steuern.

Beim Start einer Aktivität werden Kontrolltokens an allen ausgehenden Kanten der *InitialNodes* und an Aktionen, die keine eingehenden Kanten besitzen (und sich nicht innerhalb einer *InterruptibleActivityRegion* befinden, siehe unten), platziert. Die Tokens fließen entlang der Kanten im Diagramm und werden, sobald sie auf einen Kontrollknoten stoßen, entsprechend dem im vorherigen Abschnitt beschriebenen Verhalten weitergeleitet.

Wichtig ist auch die Beachtung der <Guards>, die an allen Kanten notiert sein können. Nur falls diese Bedingungen wahr sind, können die Tokens entsprechend fließen. Sobald sie auf eine Aktion stoßen und für diese Aktion an allen eingehenden Kanten Tokens anliegen, kann diese Aktion die Tokens konsumieren und starten.

Sobald die Aktion beendet ist, werden an allen ausgehenden Kanten wiederum Tokens angelegt und der Fluss entsprechend weiter berechnet. Stößt ein Token auf einen *FlowFinalNode*, wird nur dieses eine Token vernichtet. Im Gegensatz dazu vernichtet ein *ActivityFinalNode*, sobald ihn ein Token erreicht, alle noch im Diagramm vorhandenen Tokens, bricht alle aktiven Aktionen ab und beendet somit die komplette Aktivität.

Ähnlich zum Verhalten von Marken in Petri-Netzen besagt das Prinzip der „*traverse-to-completion*“-Regel, dass Kontrollknoten keine Tokens puffern können. Wenn eine nachfolgende Aktion nicht bereit ist ein Token in Empfang zu nehmen, dann bleibt das entsprechende Token an der ausgehenden Kante der vorherigen Aktion. Eine Ausnahme dieser Regel (und gleichzeitig eine Abweichung von der in [Sar06a] beschriebenen Semantik) bilden *ForkNodes*.

Für den Fall, dass eine ausgehende Kante eines *ForkNode* das angebotene Token nicht annehmen kann, spezifiziert die UML, dass es an der ausgehenden Kante des *ForkNode* gepuffert wird. Auch wenn das eine Aufweichung der „*traverse-to-completion*“-Regel bedeutet, entspricht dies doch der intuitiven Vorstellung.

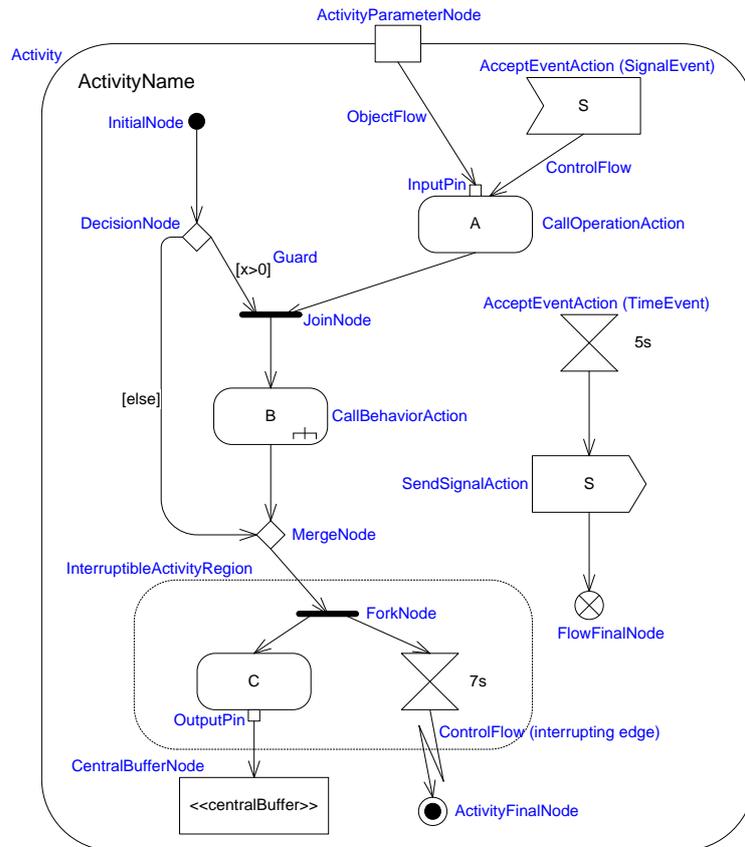


Abbildung 2.7: Syntax der UML 2 Aktivitätsdiagramme (entnommen aus [Sar06a]).

Abb. 2.8, welche einen Ausschnitt aus einem Aktivitätsdiagramm darstellt, verdeutlicht diesen Zusammenhang. In diesem Beispiel wurde die Aktion A gerade beendet (symbolisiert dadurch, dass sich ein Token (grüner Punkt) unmittelbar *unter* der Aktion befindet), aber Aktion B ist noch aktiv (Token *neben* der Aktion). Es wird weiterhin angenommen, dass die Bedingung $[x=1]$ wahr ist.

Die Flussberechnung ergibt nun, dass das Token am *ForkNode* dupliziert wird und einerseits an die linke ausgehende Kante weitergereicht und damit die Aktion C gestartet wird. Andererseits könnte die rechte ausgehende Kante das Token transportieren (da der \langle Guard \rangle wahr ist). Allerdings kann der dahinter liegende *JoinNode* das Token nicht weiterleiten, da nicht an allen eingehenden Kanten Tokens anliegen (das Token der Aktion B fehlt). Gemäß der „*traverse-to-completion*“-Regel müsste nun das Token hinter A bleiben und nicht einmal die Aktion C könnte starten.

Dem intuitiven Verständnis entspricht allerdings eher, dass die Aktion C startet und D startet, sobald B beendet ist. Dies wird durch eine Pufferung des duplizierten Tokens an der rechten ausgehenden Kante des *ForkNodes* realisiert.

Dabei muss allerdings beachtet werden, dass der \langle Guard \rangle einmal beim Vorgang der Pufferung ausgewertet werden muss (denn entsprechend [UML, Seite 376] erhalten nur die Kanten ein Token, deren \langle Guard \rangle wahr ist) und sinnvollerweise noch einmal bevor das Token die Kante tatsächlich traversiert. Anderenfalls würde man eigenartige Effekte erhalten, wie beispielswei-

se, dass in manchen Fällen Tokens über Kanten geleitet werden, obwohl der entsprechende \langle Guard \rangle falsch ist, je nachdem, ob das Token gepuffert war oder nicht.

Sarstedt hat dies in seiner Arbeit so nicht berücksichtigt, da er Probleme bei der notwendigen doppelten Auswertung des \langle Guards \rangle sieht [Sar06a, Seite 26]. Seine weniger intuitive Interpretation führt z. B. dazu, dass ausgehende Kontrollkanten einer Aktion sich nicht durch einen *ForkNode* ersetzen lassen, da das Verhalten damit verändert werden würde.

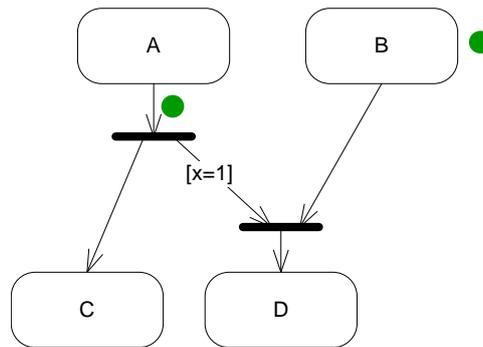


Abbildung 2.8: Beispiel für Tokenpufferung an *ForkNodes* (Erläuterung siehe Text).

Betritt ein Token eine *InterruptibleActivityRegion*, werden (zusätzlich zu den auf Grund des Tokenflusses zu startenden Aktionen) auch Signalempfänger gestartet, die keine eingehenden Kanten haben, sich aber innerhalb des betretenen Bereichs befinden. Wird die *InterruptibleActivityRegion* über eine normale Kante verlassen, laufen alle darin aktiven Aktionen weiter wie bisher. Nur wenn ein Unterbrechungsbereich durch eine Unterbrechkungskante verlassen wird, werden alle Tokens, die sich gerade innerhalb des Bereichs befinden, vernichtet und damit alle aktiven Aktionen abgebrochen.

In der formalen Definition aus [Sar06a] werden die Tokenflüsse auf folgende Art und Weise berechnet (vgl. dazu auch [SG07]): Falls eine Aktion terminiert und somit an ihren ausgehenden Kanten Tokens anliegen, wird ein Token-Angebot (*TokenOffer*) generiert. Dieses Angebot wird dann über alle Kanten und Kontrollknoten propagiert und dabei gegebenenfalls mit anderen vorkommenden Angeboten kombiniert.

Sobald für alle Knoten die anliegenden Tokens berechnet sind, wird aus den pro Aktion anliegenden Angeboten eines (oder mehrere) ausgewählt. Dies erfordert gegebenenfalls eine Neuberechnung der Angebote, da sich z. B. zwei aus einem *OutputPin* ausgehende Angebote gegenseitig ausschließen.

Die Berechnung lässt allerdings offen, ob *alle* möglichen Angebote oder immer nur eine Teilmenge davon in einem Schritt ausgewählt werden. Anschließend werden auf alle Fälle alle Aktionen, deren Startbedingungen nun erfüllt sind, gestartet.

In der Arbeit von Sarstedt sind verschiedene Variationspunkte definiert, die es dem Anwender ermöglichen, die Semantik an einigen Stellen anzupassen. Dies ist notwendig, da die UML Spezifikation einige Punkte nicht präzise genug beschreibt bzw. sogar komplett un spezifiziert lässt.

Im Folgenden werden nur die Variationspunkte vorgestellt, die bei der Verifikation der Aktivitätsdiagramme berücksichtigt werden. Gründe, warum nicht alle Variationspunkte be-

handelt werden können, finden sich in Abschnitt 3.1. Dort werden die für die Verifikation der Aktivitätsdiagramme notwendigen Einschränkungen beschrieben.

Der Mechanismus, der zur Einführung der Variationspunkte bei Sarstedt verwendet wurde, nennt sich `<tagged values>`. `<Tagged values>` sind ein Standarderweiterungsmechanismus der UML (siehe [UML, Seite 672f]). Dabei werden UML Metaklassen mit zusätzlichen Schlüssel-Werte-Paaren, die mit Hilfe von `<Stereotypen>` definiert sind, versehen.

Wie in Abb. 2.9 zu sehen, werden die entsprechenden Metaklassen über eine bestimmte Pfeilnotation mit Stereotypklassen erweitert und darin `<tagged values>` als Felder definiert. Im Diagramm können dann die entsprechenden Elemente mit einer Notiz versehen werden, in der den `<Tag>`s die gewünschten Werte zugewiesen werden. Auf diese Art und Weise kann in einem Diagramm z. B. jedem Unterbrechungsbereich eine eigene Semantik zugewiesen werden.

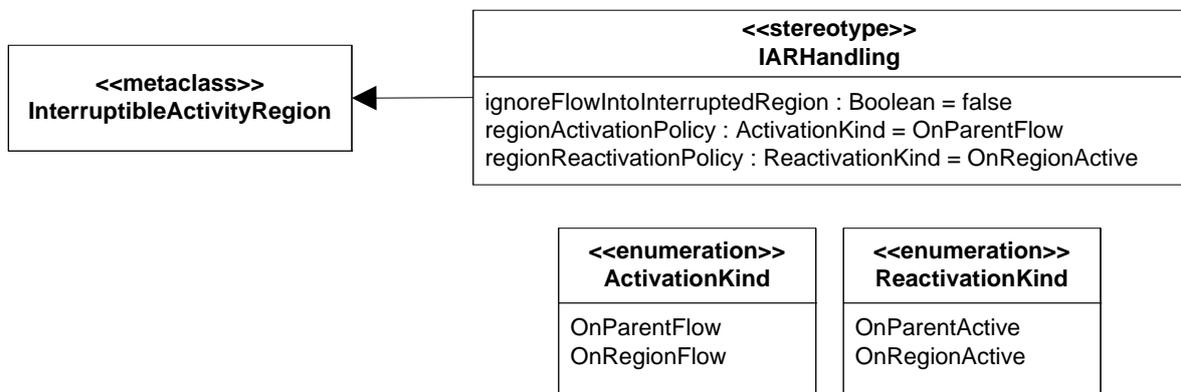


Abbildung 2.9: Definition der Variationspunkte für *InterruptibleActivityRegions*.

Für Unterbrechungsbereiche definiert die UML nicht, ob Tokenflüsse, die in einen in diesem Schritt abgebrochenen Unterbrechungsbereich münden, ebenfalls mit abgebrochen werden sollen oder nicht. Das `<Tag>` `ignoreFlowIntoInterruptedRegion` regelt dies, indem alle eingehenden Flüsse standardmäßig nicht abgebrochen werden, bei Bedarf aber auch ignoriert werden können.

Wie oben beschrieben, werden *AcceptEventActions* ohne eingehende Kanten, die sich innerhalb einer *InterruptibleActivityRegion* befinden, gestartet, sobald der unmittelbar umgebende Bereich betreten wird. Unterbrechungsbereiche können allerdings auch geschachtelt sein. Daher kann mit Hilfe von `regionActivationPolicy` definiert werden, dass eine Aktion bereits gestartet wird, wenn einer der verschachtelten umgebenden Bereiche betreten wird.

Außerdem kann über das `<Tag>` `regionReactivationFlow` noch eingeschränkt werden, in welchen Fällen ein Signalempfänger nach der Verarbeitung eines Ereignisses reaktiviert werden soll. `OnRegionActive` gibt an, dass dies nur der Fall ist, wenn sich noch mindestens ein Token innerhalb der unmittelbar umgebenden Region befindet. Wenn das `<Tag>` den Wert `OnParentActive` hat, reicht es zur Reaktivierung auch, wenn sich noch mindestens ein Token in einer der umgebenden geschachtelten Regionen befindet.

Ein anderer Variationspunkt betrifft die Pufferung von auftretenden Ereignissen. Normalerweise werden alle eingehenden Signale in einem Puffer gespeichert, bis sie von einem Ereignisempfänger abgearbeitet werden. In manchen Fällen ist es aber passender, dass eingehende Signale entweder sofort verarbeitet werden müssen oder aber sonst verloren gehen.

Ebenso ist es denkbar, dass ein wiederholt auftretendes Ereignis im Puffer bereits vorhandene überschreibt. Abb. 2.10 zeigt die Definition der Stereotypen für diese Variationspunkte.

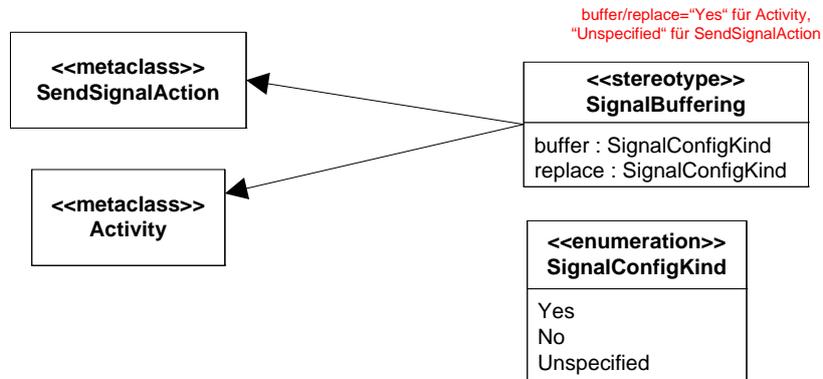


Abbildung 2.10: Definition der Variationspunkte zur Signalpufferung.

Bei Sarstedt wird der Stereotyp `SignalBufferingAndReplacement` statt an die `SendSignalAction` an die `AcceptEventAction` gebunden. Dadurch wird die Behandlung allerdings teilweise unklar:

Angenommen, in einer `Activity` sind zwei `AcceptEventActions` vorhanden, wovon eine durch `<tagged values>` definiert, dass das entsprechende Signal im Puffer ersetzt werden soll. Die andere `AcceptEventAction` definiert genau das Gegenteil, nämlich dass das Signal auch dann in den Puffer eingefügt werden soll, wenn es schon vorhanden ist.

Nun ist beim Versenden des Signals unklar, welches Verhalten angewendet werden soll, da sich beide gegenseitig überschreiben. Dieses Problem kann vermieden werden, indem die `<tagged values>` nicht mit den Konsumenten von Ereignissen, sondern mit den Produzenten — den `SendSignalActions` — verknüpft werden.

Kapitel 3

Transformation von Aktivitätsdiagrammen in Zustandsübergangssysteme

Der Hauptteil dieser Arbeit stellt die Transformation von Aktivitätsdiagrammen in ein Zustandsübergangssystem vor. Dieses Zustandsübergangssystem kann von vorhandenen Model Checkern zur automatisierten Verifikation verwendet werden. Das Verfahren deckt weitgehend den Umfang der formalen Semantikbeschreibung von Sarstedt [Sar06a] ab (siehe auch Abschnitt 3.1).

Es berücksichtigt dabei insbesondere auch schwierigere Aspekte der Aktivitätsdiagramme wie Datenflüsse, Signalbehandlung und Unterbrechungsbereiche. Die für das Verständnis der Übersetzung notwendigen Details der verwendeten Semantik werden an den entsprechenden Stellen erläutert.

Aspekte aus der Arbeit von Sarstedt, die für das Model Checking nicht berücksichtigt werden können, sind in Abschnitt 3.1 beschrieben. Einige dieser Einschränkungen werden im Kapitel 5 wieder aufgegriffen und mögliche Erweiterungen und Anpassungen diskutiert.

Der darauf folgende Abschnitt 3.2 beschreibt ausführlich die einzelnen Schritte der Übersetzung, welche mit Hilfe der Abstract State Machines formalisiert sind. Der Schwerpunkt liegt dabei auf der Berechnung der möglichen Kontroll- und Objektflüsse, die das Kernstück der Transformation darstellen.

In Abschnitt 3.3 wird das Vorgehen für einen Nachweis, dass die Transformation semantikerhaltend ist, beschrieben.

Eine einfachere Version des hier vorgestellten Algorithmus, die nur Kontrollflüsse betrachtet, wurde in [Ras09] veröffentlicht.

3.1 Einschränkungen

Zusätzlich zu den in Abschnitt 2.3.2 beschriebenen Anpassungen sind aus verschiedenen Gründen Einschränkungen der bei Sarstedt beschriebenen Semantik notwendig. Diese Einschränkungen ergeben sich zum einen aus der beim symbolischen Model Checking notwendigerweise statischen Umgebung, zum anderen aus der Tatsache heraus, dass beim Model Checking im Allgemeinen Zeitaspekte unberücksichtigt bleiben.

Keine Schleifen zwischen Kontrollknoten. Der Algorithmus zur Berechnung der möglichen Kontroll- bzw. Objektflüsse würde nicht terminieren, wenn Schleifen zwischen Kontrollknoten erlaubt wären. Abb. 3.1 zeigt ein Beispiel eines solchen Zyklus.

Das Verbot solcher Konstellationen ist aber keine wirkliche Beschränkung, da einerseits die Semantikdefinition von Sarstedt diesen Fall schon ausschließt und andererseits der Sinn eines solchen Zyklus, der keine Zustandsänderung produziert (da diese nur innerhalb von Aktionen möglich ist) sehr fraglich ist. Zyklen zwischen Aktionen, wie sie z. B. in der Abbildung von Aktion A nach Aktion B und wieder zurück möglich sind, sind allerdings schon erlaubt.

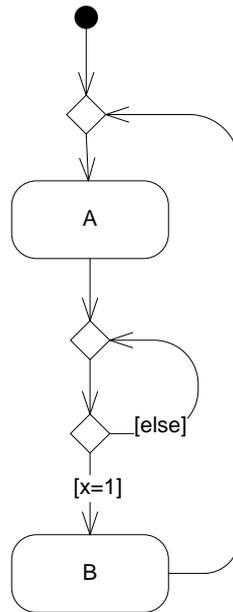


Abbildung 3.1: Beispiel für einen Zyklus zwischen zwei Kontrollknoten.

Keine Unterstützung von Zeitereignissen. *AcceptEventActions*, welche auf Zeitereignisse reagieren, können nicht einfach berücksichtigt werden, da die meisten Model Checker von absoluter Zeit abstrahieren. Stattdessen wird das Zeitereignis wie ein externes Signal behandelt, welches zu jedem Zeitpunkt auftreten kann.

Damit erhält man gegebenenfalls eine Überabschätzung. Es können durch diese Abstraktion Fehler in einem Aktivitätsdiagramm gefunden werden, die auf Grund der Timerkonstellation in der Realität nicht auftreten. Abb. 3.2 zeigt ein solches Beispiel.

Angenommen, in der Aktion *failure* wird eine zu verifizierende Eigenschaft verletzt. Die Verifikation der Aktivität würde diesen Pfad als Gegenbeispiel zurückliefern, obwohl dieser Pfad nie durchlaufen werden kann. Der rechte Timer mit drei Sekunden Wartezeit bricht den anderen Timer stets vor dessen Ablauf ab. Solche *false positives* müssen dann vom Anwender nachvollzogen und eventuell ignoriert werden.

Gemäß UML Spezifikation können Timer mit einem *OutputPin* versehen werden, der dann die aktuelle Uhrzeit als Objektfluss weiterreicht. Durch die besondere Behandlung von Timern macht es wenig Sinn, einen *OutputPin* für Timer zuzulassen.

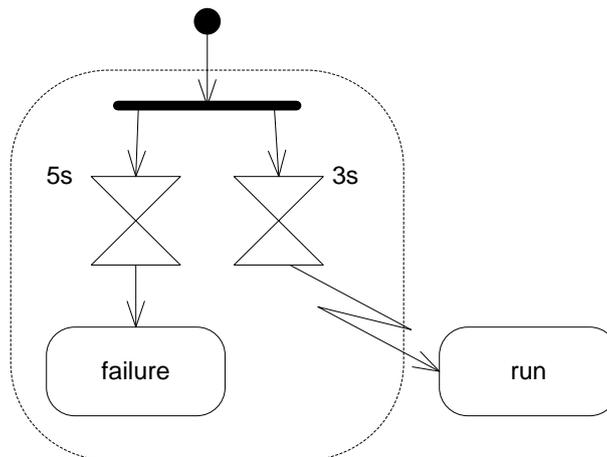


Abbildung 3.2: Beispiel für eine Überabschätzung durch Abstraktion von Zeit.

Verhalten in *CallOperationActions* abhängig von den Möglichkeiten des Model Checkers. Das Verhalten in *CallOperationActions* muss sich auf die Möglichkeiten des eingesetzten Model Checkers beschränken. Nur wenn die Berechnungen und Datenmanipulationen in die Eingabesprache des verwendeten Model Checkers übertragen werden können, kann das Verhalten während der Verifikation berücksichtigt werden.

Auch wenn die meisten (effizienten) Model Checker noch nicht annähernd die Mächtigkeit einer modernen Programmiersprache anbieten (was auch maßgeblich auf die Nichtentscheidbarkeit einiger Konstrukte zurückzuführen ist), lassen sich dennoch bereits viele interessante Eigenschaften sogar nur bei der Betrachtung des Kontroll- bzw. Objektflusses innerhalb einer Aktivität verifizieren.

Dies ist auch ein wesentlicher Aspekt dieser Arbeit, da Aktivitätsdiagramme häufig in sehr frühen Phasen des Softwareentwicklungsprozesses eingesetzt werden. In diesen Phasen sind die Aktionen typischerweise noch nicht mit Algorithmen hinterlegt und dennoch lassen sich bereits verschiedene Eigenschaften des Kontroll- und Objektflusses verifizieren.

Das Fallbeispiel „Zweihandpresse“ in Abschnitt 4.1.3.2 macht von dieser Möglichkeit Gebrauch und zeigt, wie effektiv Verifikation helfen kann, Fehler zu finden.

Eine Möglichkeit, die Beschränkung des Model Checkers zu umgehen, ist die Einführung von Vor- bzw. Nachbedingungen an Aktionen. Der Model Checker überprüft dann lediglich, ob die Vorbedingung zutrifft und kann dann von einer gültigen Nachbedingung ausgehen. Auf diese Art und Weise muss keine explizite Berechnung angegeben werden.

CallOperationActions werden atomar ausgeführt. Das heißt, im spezifizierten Verhalten können keine Seiteneffekte auftreten. Diese wären möglich, wenn die Änderung eines Klassenattributs unmittelbar für alle gerade aktiven Aktionen sofort sichtbar werden würde. Alle Attributänderungen werden aber immer erst nach der Beendigung einer Aktion für die restlichen Aktionen sichtbar.

Nur eine Ausführung pro Aktivität. Die UML sieht vor, dass ein Verhalten einer Klasse zugeordnet wird [UML, Seite 318]. Die Klasse beschreibt dann den „Kontext“ dieses Verhaltens und definiert die Attribute und Methoden, auf die innerhalb des Verhaltens z. B. in *CallOperationActions* oder in Bedingungen an Kanten zugegriffen werden kann.

Für jede Instanz einer Klasse wird auch genau eine Instanz des ihr zugeordneten Verhaltens erzeugt. Diese Instanz eines Verhaltens nennt man Ausführungsinstanz.

Im Rahmen von symbolischen Model Checking ist es durch das zugrundeliegende statische Modell nicht möglich, neue Instanzen von Elementen während der Prüfung zu erzeugen. Daher können mehrere Ausführungsinstanzen nicht realisiert werden.

Stattdessen wird in dieser Arbeit von einer einzigen Instanz pro Aktivität ausgegangen. Wird von der erwähnten Möglichkeit, auf die Klassenattribute innerhalb eines Aktivitätsdiagramms zuzugreifen, Gebrauch gemacht, impliziert dies, dass für die Transformation auch die referenzierten Klassen zugänglich sein müssen und für die benutzten Attribute ein Initialzustand definiert sein muss.

Die UML unterstützt diese Sichtweise mit nur einer Instanz, indem sie die Möglichkeit bietet, die Anzahl der Aktivitätsausführungen auf genau eine zu beschränken. Dies geschieht über das Attribut *isSingleExecution* der Metaklasse *Activity*, welches in dieser Arbeit standardmäßig als *true* angenommen wird.

Da keine Instanzen von Klassen automatisch zu Beginn einer Anwendung erzeugt werden, ist es notwendig, die Aktivitäten, die beim Start des Systems ebenfalls sofort gestartet werden sollen, zu kennzeichnen. Dies geschieht mit Hilfe eines neu definierten <Tags> *initialStart*.

Abb. 3.3 zeigt die Definition des <Tags>, dessen Standardwert *false* ist und somit kennzeichnet, dass die damit verknüpfte Aktivität nicht automatisch gestartet wird, sondern erst bei Aufruf durch eine *CallBehaviorAction*. Alle Aktivitäten, die zu Beginn einer Anwendung gestartet werden sollen, müssen demnach mit einem <Tag> gekennzeichnet sein, der *initialStart* auf *true* setzt.



Abbildung 3.3: Definition des Variationspunkts zur Kennzeichnung von Startaktivitäten.

Auch für die Aktivitäten, die über *CallBehaviorActions* aufgerufen werden, existiert nur eine einzige Ausführungsinstanz. Falls mehrere *CallBehaviorActions* in einem Aktivitätsdiagramm die gleiche Aktivität aufrufen, überlagern sich folglich die verschiedenen Ausführungen und alle Änderungen der Klassenattribute werden sofort für alle anderen Ausführungen sichtbar. Dies muss bei der Erstellung der Aktivitätsdiagramme berücksichtigt werden. Mögliche Lösungen, wie diese Probleme der Instanziierung alternativ behandelt werden könnten, finden sich in Kapitel 5.

Keine Unterstützung von Signalpfaden. In der Arbeit von Sarstedt [Sar05] wurde ein Mechanismus vorgestellt, wie das Ziel einer *SendSignalAction* mit Hilfe von XPath-Ausdrücken [BBC⁺] bestimmt werden kann. Da in der vorliegenden Arbeit aber — wie oben beschrieben — nur von einer Ausführungsinstanz ausgegangen wird und auch keine Objekt-hierarchien betrachtet werden können, macht die Unterstützung dieser XPath-Ausdrücke keinen Sinn.

Eine weitere Vereinfachung ergibt sich dadurch, dass alle *SendSignalActions* als *BroadcastSignalActions* aufgefasst werden, so dass ein Signal immer an alle Signalempfänger gesendet

wird. Auch dies führt zu einer Überabschätzung, die im Einzelfall zu einem *false positive* führen kann.

Nicht alle semantischen Variationspunkte. Wie in Abschnitt 2.3.2 bereits erwähnt, werden einige der bei Sarstedt definierten Variationspunkte nicht behandelt. Die Variationspunkte `CallContext`, `SignalTargeting` und `SignalDistribution` behandeln Probleme, die im Zusammenhang mit Kontextobjekten auftreten. Diese Variationspunkte können nicht berücksichtigt werden, da — wie bereits beschrieben — auf Kontextobjekte verzichtet werden muss.

Der Variationspunkt `singleInterrupt` bestimmt, ob ein Unterbrechungsbereich von mehreren Unterbrechungskanten gleichzeitig verlassen werden darf. Falls dies nicht der Fall sein soll, kann über den Variationspunkt `Interruptpriority` ausgewählt werden, welche der konkurrierenden Kanten gewählt wird.

Dazu werden an den Kanten Prioritäten gesetzt und nur diejenige aller aktiven Kanten gewählt, deren Priorität am höchsten ist. Auf die Umsetzung wird deswegen verzichtet, da diese Funktionalität mit Hilfe von *Merge* - und *DecisionNodes* leicht direkt modelliert werden kann.

Tabelle 3.1 fasst die erläuterten Einschränkungen zusammen. Sie zeigt auch, von welchen Aspekten einige Einschränkungen abhängen und was die Folgen der entsprechenden Einschränkung sind. Außerdem wird dargestellt, welche Anpassungen bzw. Erweiterungen notwendig wären, um diese Aspekte ebenfalls berücksichtigen zu können. Mögliche Lösungen werden in Abschnitt 5.2 detaillierter beschrieben.

Tabelle 3.1: Übersicht der Einschränkungen.

Einschränkung	abhängig von	führt zu	mögliche Lösung
keine Schleifen zwischen Kontrollknoten	—	—	—
keine Zeitereignisse	—	Überabschätzung	Timed Automata
beschränktes Verhalten in Aktionen	Möglichkeiten des Model Checkers	—	explizites MC, Vor- und Nachbedingungen
nur eine Ausführungsinstanz	—	genau eine Ausführung pro Aktivität	explizites MC
keine Signalpfade	mehreren Ausführungsinstanzen	alle Signale werden wie BroadcastSignals behandelt	explizites MC
nicht alle semantischen Variationspunkte aus [Sar06a]	mehreren Ausführungsinstanzen	—	explizites MC

Abschließend sei bemerkt, dass die Möglichkeiten der automatischen Verifikation von Aktivitätsdiagrammen auch von der Mächtigkeit des gewählten Model Checkers abhängen. An welchen Stellen Einschränkungen von Seiten des verwendeten Werkzeugs eine wesentliche Rolle spielen und welche Auswirkungen das auf die Verifikation hat, wird in Abschnitt 4.1.2 bei der Beschreibung der Umsetzung des Zustandsübergangssystems in die Eingabesprache des Model Checkers erläutert.

3.2 Transformation

In diesem Abschnitt wird der Algorithmus zur Übersetzung von Aktivitätsdiagrammen in Zustandsübergangssysteme ausführlich beschrieben und mit ASMs formalisiert. Eine Zusammenfassung aller ASM-Makros inklusive einiger Hilfsmakros finden sich in Anhang A. In den folgenden Abschnitten werden die wichtigeren Makros mit einer detaillierten Erläuterung dargestellt.

3.2.1 Übersicht

Einen Überblick über den prinzipiellen Ablauf des Model Checkings von Aktivitätsdiagrammen gibt Abb. 3.4. Die einzelnen Teilschritte der Transformation sind dick umrandet dargestellt. Die für die automatische Verifikation von Aktivitätsdiagrammen zusätzlich notwendigen Schritte werden nur grob beschrieben und nicht in der Ausführlichkeit behandelt, wie dies für die Transformationsschritte der Fall ist. Die Ziffern in den Kästchen verweisen jeweils auf den Abschnitt, der den entsprechenden Schritt detailliert behandelt.

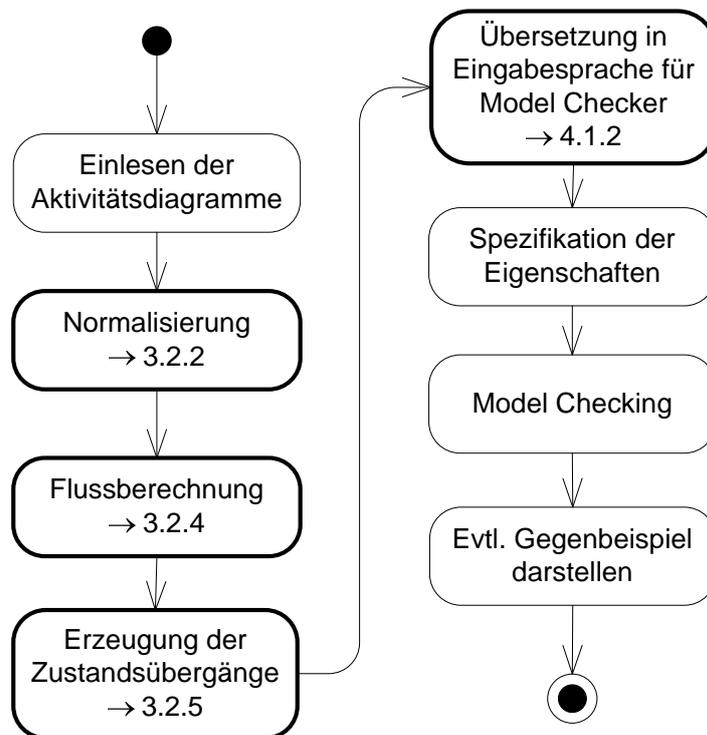


Abbildung 3.4: Übersicht über den Ablauf der Verifikation von Aktivitätsdiagrammen. Die Ziffern in den Kästen verweisen auf den Abschnitt, in dem der entsprechende Schritt beschrieben ist.

Die aus einem beliebigen Datenformat eingelesenen Diagramme werden im ersten Schritt normalisiert. Diese Normalisierung (\rightarrow 3.2.2) sorgt dafür, dass die folgenden Berechnungsschritte auf einer einheitlichen Basis aufbauen können und nicht zusätzliche Fallunterschei-

dungen oder Überprüfungen notwendig sind.

Abschnitt 3.2.3 beschreibt kurz die Elemente eines Zustands des zu erstellenden Zustandsübergangssystem. Diese Zustände werden allerdings (wie beim Model Checking üblich (siehe Abschnitt 2.1)) nicht explizit angegeben, sondern mit Hilfe der Zustandsübergänge im Rahmen des Model Checkings aus dem Initialzustand abgeleitet.

Aus den normalisierten Diagrammen werden die möglichen Kontroll- bzw. Objektflüsse innerhalb eines Diagramms berechnet (\rightarrow 3.2.4). Ein Fluss wird dabei als ein möglicher Pfad eines von einer Aktion angebotenen Tokens durch den Graphen des Aktivitätsdiagramms bis zu einem Knoten, der dieses Token konsumieren kann, gesehen. Dabei sind evtl. mehrere Pfade pro Aktion möglich, da Kontrollknoten den Fluss (und damit auch die Tokens) aufteilen können.

Außerdem können auch mehrere Tokenangebote von verschiedenen Aktionen an einem Fluss beteiligt sein (z. B. fasst ein Join-Node mehrere Tokens zu einem zusammen). Daher werden zunächst die möglichen Flüsse *einer* Aktion (*einfacher Fluss*) und erst in einem weiteren Schritt alle Kombinationen der einfachen Flüsse (*kombinierter Fluss*) berechnet.

Ausgehend von der Menge aller möglichen Flüsse (einfache und kombinierte), werden der Initialzustand und die Zustandsübergänge für das zu erstellende Zustandsübergangssystem bestimmt (\rightarrow 3.2.5). In diesem Schritt werden neben den betretenen und abgebrochenen Unterbrechungsbereichen auch die entsprechenden Variationspunkte berücksichtigt.

Ein wesentliches Ziel dieser Arbeit ist die Flexibilität bezüglich des eingesetzten Model Checkers. Die im vorherigen Schritt erstellten Zustandsübergänge abstrahieren daher von einer realen Eingabesprache. In dieser Arbeit wird allerdings auch auf die Definition einer abstrakten Eingabesprache verzichtet.

Stattdessen wird die Möglichkeit der Abstract State Machine ausgenutzt, einzelne Makros und Funktionen nicht zu spezifizieren und auf diese Art und Weise zu abstrahieren. An Stellen, an denen während der Bestimmung der Zustandsübergänge eine Ausgabe in eine Model Checker Eingabesprache notwendig wird, werden abstrakte ASM-Makros aufgerufen, die für ein bestimmtes Model Checker Werkzeug entsprechend definiert werden müssen.

Insofern wird der letzte und der vorletzte Schritt der Transformation zu einem gewissen Grad vermischt. In Abschnitt 4.1 wird beispielhaft eine konkrete Umsetzung unter Benutzung eines Model Checkers für *Model Programs* [VBR08] beschrieben.

3.2.2 Normalisierung

Das konkrete Eingabeformat, in welchem die Aktivitätsdiagramme vorliegen, ist irrelevant. Es muss lediglich sichergestellt werden, dass das Eingabeformat in eine korrekte und vollständige Instanz des Metamodells für Aktivitätsdiagramme (siehe Abb. 2.6) überführt werden kann. Alle weiteren Berechnungen operieren dann auf dieser Instanz des Metamodells.

Damit auf dieses Modell in den ASM-Regeln zugegriffen werden kann, wird es in Domänen (siehe Abschnitt 2.2) und Funktionen formalisiert. Die prinzipielle Vorgehensweise bei der Formalisierung, die weitgehend aus [Sar06a] übernommen werden konnte, wird an Hand eines Beispiels erläutert. Eine vollständige Auflistung der definierten Domänen findet sich in Anhang A.1.

Abb. 3.5 zeigt einen Ausschnitt aus dem Metamodell, dessen Formalisierung im Folgenden beschrieben wird. Für alle beteiligten Metaklassen werden zunächst entsprechende ASM-Domänen definiert. Generalisierung bzw. Spezialisierung wird dabei durch eine Teilmengenbeziehung zwischen zwei Mengen ausgedrückt.

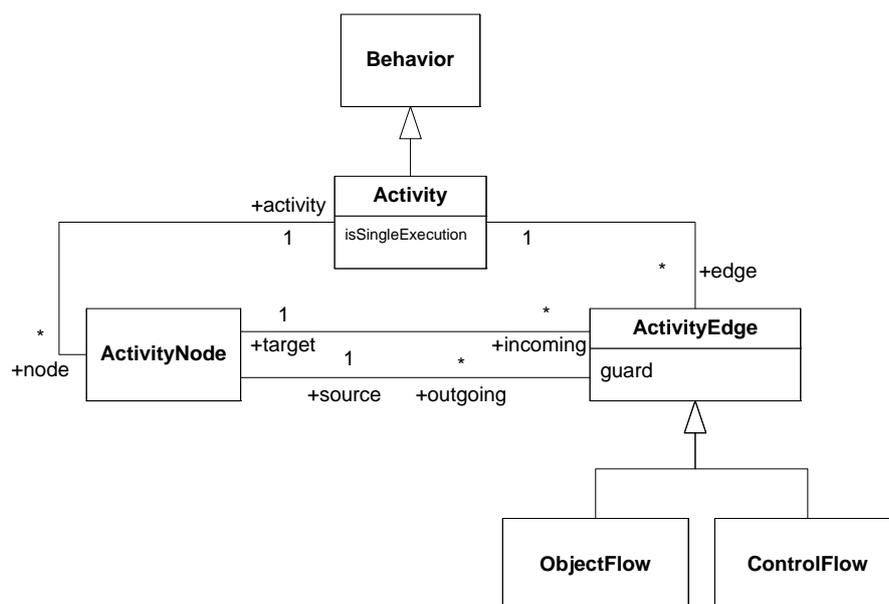


Abbildung 3.5: Ausschnitt aus dem Metamodell für UML 2 Aktivitätsdiagramme.

static domain *Behavior*
static domain *Activity* \subseteq *Behavior*
static domain *ActivityNode*
static domain *ActivityEdge*
static domain *ControlFlow* \subseteq *ActivityEdge*
static domain *ObjectFlow* \subseteq *ActivityEdge*

Anschließend werden für das jeweilige Attribut der Klassen *Activity* und *ActivityEdge* Funktionen definiert:

static *isSingleExecution*: *Activity* \rightarrow *Boolean*
static *guard*: *ActivityEdge* \rightarrow *GuardSpecification*

Diese beiden Funktionen erhalten als Argument ein Element der entsprechenden Menge und bilden diese auf ein Element der Menge *Boolean* oder *GuardSpecification* ab. Ähnlich werden auch die Beziehungen zwischen Klassen abgebildet, wobei die Bezeichner dem Rollennamen der Beziehung entsprechen:

static *node*: *Activity* \rightarrow $\mathcal{P}(\text{ActivityNode})$
static *edge*: *Activity* \rightarrow $\mathcal{P}(\text{ActivityEdge})$
static *incoming*, *outgoing*: *ActivityNode* \rightarrow $\mathcal{P}(\text{ActivityEdge})$
static *activity*: *ActivityNode* \rightarrow *Activity*
static *source*, *target*: *ActivityEdge* \rightarrow *ActivityNode*

Eine Potenzmenge einer Domäne als Rückgabewert realisiert höherwertige Kardinalitäten im Metamodell. Die Funktion *node* beispielsweise liefert eine Teilmenge der Potenzmenge über der Menge der *ActivityNode* zurück, was exakt der Definition von Kardinalitäten entspricht.

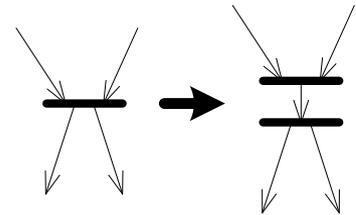
Zur besseren Lesbarkeit werden alle Funktionen, die sich auf das Metamodell beziehen, in Punkt-Notation notiert. Der Ausdruck $n.incoming$ (mit $n \in ActivityNode$) entspricht dabei dem Funktionsaufruf *incoming*(n).

Die Definition der Domänen und Funktionen ist statisch, um zum Ausdruck zu bringen, dass während der Berechnung des Zustandsübergangssystems die zu verarbeitende Instanz des Metamodells nicht mehr verändert wird. Eine Anpassung kann allerdings noch im Rahmen der Normalisierung erfolgen, die im Folgenden beschrieben wird.

Bei der Normalisierung und allen weiteren Schritten wird davon ausgegangen, dass die betrachtete Instanz des Metamodells syntaktisch korrekt, vollständig und konsistent ist. Die Werte von *upper*- bzw. *lower*-Angaben an *Pins*, die eine untere bzw. obere Grenze von eingehenden Datentokens bestimmen, müssen so gewählt sein, dass gilt: $0 < lower \leq upper$.

Syntaktische Korrektheit schließt in diesem Fall auch die korrekte Verwendung von Objektflüssen mit ein, so dass z. B. jedem *OutputPin* ein *InputPin* zugeordnet werden kann. Des Weiteren findet keine Typüberprüfung statt. Es wird daher auch nicht überprüft, ob der Objektfluss von einem *OutputPin* zu einem *InputPin* gültig ist oder der Ausdruck eines *Guard* für die darin verwendeten Typen definiert ist. Stattdessen wird davon ausgegangen, dass diese statischen syntaktischen Überprüfungen schon im Vorfeld stattgefunden haben.

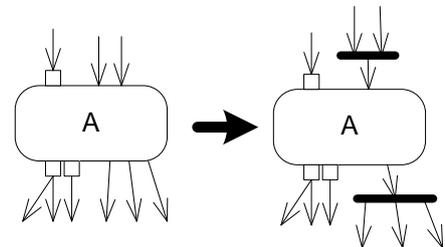
Die UML Spezifikation bietet die Möglichkeit, *Decision*- bzw. *MergeNodes* und *Fork*- bzw. *JoinNodes* zusammenzufassen. Für eine leichtere Beschreibung der Algorithmen ist es von Vorteil, wenn diese „Mischknoten“ in zwei Knoten mit jeweils einer aus- bzw. eingehenden Kante aufgeteilt werden, so dass nur noch „reine“ Kontrollknoten im Diagramm vorhanden sind. Die Abbildung rechts verdeutlicht diesen Zusammenhang.



Anschließend werden an den ausgehenden Kanten von *DecisionNodes* die *else*-Zweige aufgelöst. Dabei wird dem *else*-Zweig eines *DecisionNodes* *d* die Konjunktion der Negation der restlichen Kanten als *Guard* zugewiesen:

$$elseEdge.guard := \bigwedge_{e \in d.edge \wedge e \neq elseEdge} \neg e.guard$$

Für die weitere Verarbeitung erweist es sich auch von Vorteil, wenn mehrere ausgehende Kontrollflüsse einer Aktion oder eines *InitialNodes* durch einen Kontrollfluss und einen *ForkNode* ersetzt werden. Eine ähnliche Normalisierung ist auch mit eingehenden Kontrollflüssen und *JoinNodes* möglich.



Objektflüsse können nicht auf diese Art und Weise zusammengefasst werden, da sich mehrere ausgehende Kanten von *OutputPins* gegenseitig ausschließen und dieses Verhalten mit der Verwendung eines *ForkNodes* aufgehoben wäre. Dieselbe Normalisierung kann auch bei *InitialNodes* durchgeführt werden. Bei *ActivityFinalNodes* ist dies nicht erforderlich, da die Semantik der UML sowieso festlegt, dass die komplette Aktivität beendet wird, sobald *nur ein* Token den Knoten erreicht.

Gleiche Namen von Aktionen in verschiedenen Aktivitäten müssen eindeutig umbenannt werden, da bei der Erzeugung der Eingabe für den Model Checker Variablennamen aus den Bezeichnungen der Aktionen erstellt werden. Anderenfalls würde es bei der Berechnung zu Verwechslungen und damit zu unerwünschtem Verhalten kommen.

Die ursprüngliche Bezeichnung darf allerdings nicht verworfen werden, da sie häufig zusätzliche Informationen bietet (z. B. bezeichnet die Beschriftung einer *SendSignalAction* das damit versendete Signal). Die Speicherung muss somit außerhalb des Metamodells erfolgen, da dort keine zusätzliche Benennung von Elementen vorgesehen ist.

3.2.3 Zustände

Einzelne Systemzustände werden bei der automatischen Verifikation nicht direkt angegeben, sondern aus einem Initialzustand mit Hilfe von Zustandsübergängen berechnet. Dennoch wird in diesem Abschnitt kurz dargestellt wie ein Zustand eines Aktivitätsdiagramms aussieht. Dies erleichtert das Nachvollziehen des Algorithmus, der im Anschluss beschrieben wird.

Eine Aktion einer Aktivität kann sich laut UML Spezifikation [UML, Seite 312] in verschiedenen Modi befinden: Wenn an allen eingehenden Kanten und *InputPins* Tokens anliegen, wird eine Aktionsausführung erzeugt (`<created>`). Sie wird aktiv (`<enabled>`), sobald die Tokens und Daten tatsächlich transferiert sind.

Die Aktion kann dann in den Modus `<running>` übergehen, in welchem sie bleibt, bis die Ausführung beendet (`<completed>`) ist. Schließlich werden die ausgehenden Kanten und *Pins* der beendeten (`<terminated>`) Aktion mit Tokens und Daten versorgt, die wiederum den nachfolgenden Aktionen angeboten werden (`<offering tokens>`).

Da sich einige dieser Modi überschneiden bzw. deren Unterscheidung nicht ganz klar ist (z. B. `<enabled>` vs. `<running>`), wird in dieser Arbeit von nur drei Modi ausgegangen:

- **inactive:** Die Aktion ist nicht aktiv und bietet auch keine Tokens an. Dies ist der Standardmodus für alle Aktionen.
- **running:** Die Aktion läuft und kann, je nach Art der Aktion, Ausgaben berechnen, Signale senden oder Signale empfangen. Insbesondere ist es jederzeit möglich, dass die Aktion beendet wird. Dann geht sie unmittelbar in den Modus `offeringToken` über.
- **offeringToken:** Die Aktion wurde beendet, die Änderungen am Systemzustand wurden sichtbar und die ausgehenden Kanten und *Pins* bieten Kontroll- bzw. Datentokens an.

Der Begriff „Modus“ wird in diesem Zusammenhang verwendet, um von dem Begriff des „Zustands“, der die Menge aller Aktionen mit ihren jeweiligen Modi bezeichnet, zu unterscheiden.

Ein Vorteil der Beschränkung auf drei Modi liegt auch darin, dass die Anzahl der möglichen Zustände deutlich reduziert wird und dies entscheidenden Einfluss auf die Praktikabilität des Model Checkings hat. In Abschnitt 4.3 werden noch weitere Möglichkeiten vorgestellt, wie der Zustandsraum verkleinert werden könnte.

Die Systemkonfiguration der Aktivitäten umfasst neben dem Modus für jede Aktion noch einige weitere Variablen:

- Klassenattribute
- Daten, die an den *In-* bzw. *OutputPins* gehalten werden

- Signalpuffer und Variablen für Spezialfälle wie z. B. die *ForkNode*-Pufferung (siehe Abschnitt 2.3.2)

Ein Systemzustand ist der Zustand, der durch die Variablenbelegung zu einem bestimmten Zeitpunkt beschrieben wird. Zustandsübergänge beschreiben, wie man von einem Systemzustand in einen anderen gelangt.

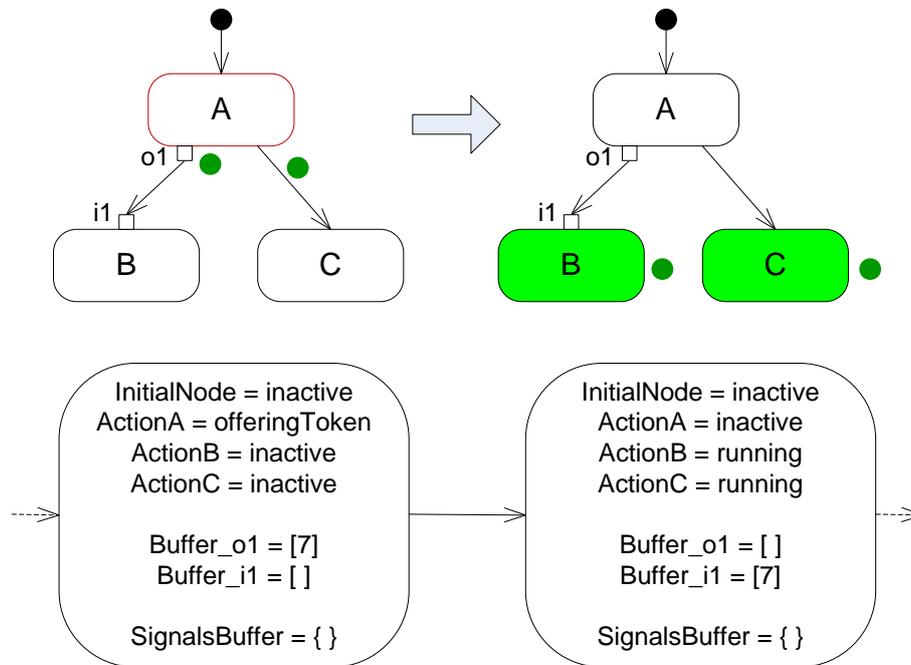


Abbildung 3.6: Beispiel für einen Zustandsübergang.

Abb. 3.6 verdeutlicht die beschriebene Umsetzung von Aktivitätsabläufen in Zuständen. Im linken Aktivitätsdiagramm wurde die Aktion A gerade eben beendet. Sie bietet deshalb am *OutputPin* o1 und an der ausgehenden Kontrollkante jeweils ein Token an. Außerdem ist der *OutputPin* mit dem Ausgabedatum der Aktion A gefüllt (angenommener Wert 7).

Der unterhalb dargestellte Zustand entspricht genau dieser Ausgangssituation. Wenn nun die Tokens fließen, wird Aktion A inaktiv und die Aktionen B und C werden gestartet. Darüber hinaus fließt der Datentoken von *OutputPin* o1 nach *InputPin* i1 und kann dort bei der Berechnung innerhalb der Aktion verwendet werden.

Zu beachten ist dabei, dass logischerweise nicht alle Kombinationen der Variablenbelegung erreicht werden können. Die erreichbaren Systemzustände werden durch die Aktionen und Kontroll- und Datenflüsse in den Aktivitätsdiagrammen beschränkt.

Ein weiterer Zustandsübergang findet beispielsweise statt, wenn eine *SendSignalAction* ein Signal sendet oder eine *AcceptEventAction* ein Signal empfängt. Außerdem kann der Systemzustand in Aktionen geändert werden, spätestens, wenn eine Aktion beendet ist. Schließlich findet eine Zustandsänderung statt, wenn ein Token von einer Aktion zu einer nächsten fließt und diese startet.

Diese Berechnung der möglichen Tokenflüsse in einem Aktivitätsdiagramm stellt somit eine zentrale Komponente bei der Überführung in Zustandsübergangsdigramme dar und ist Inhalt des folgenden Abschnitts.

3.2.4 Flussberechnung

Für die Bestimmung der Zustandsübergänge ist es essentiell, herauszufinden, wie Tokens im Diagramm prinzipiell fließen können und welche Auswirkungen sich daraus ergeben. Der naive Ansatz, zu jedem Knoten einfach die jeweiligen Nachfolger zu bestimmen, führt aus mehreren Gründen nicht zum Ziel:

Sobald z. B. ein *JoinNode* im Pfad von einer Aktion zur nächsten liegt, müssen, damit der Token tatsächlich fließen kann, an allen eingehenden Kanten des *JoinNodes* Tokens anliegen. Daher ist es in diesem Fall notwendig, auch die Vorgänger dieses Knotens in der Flussberechnung einzubeziehen.

Es reicht allerdings auch nicht aus, zu jedem Knoten die unmittelbaren Vorgänger und Nachfolger zu bestimmen. Die in Abschnitt 2.3.2 beschriebene „*traverse-to-completion*“-Regel fordert, dass Tokens immer von einer Aktion zur nächsten fließen und sich dazwischen nicht aufhalten können. Durch diese Regel ist es notwendig, immer komplette Flüsse zwischen allen beteiligten Aktionen zu bestimmen. Dies ist für allgemeine Aktivitätsdiagramme komplex, da neben den Aktionen auch Kontrollknoten, Datenflüsse und Unterbrechungsbereiche berücksichtigt werden müssen.

Ziel der Flussberechnung ist es nun, eine Menge aller möglichen Flüsse im betrachteten Diagramm zurückzuliefern. Entscheidend ist dabei, dass nicht nur die in einem konkreten Systemzustand möglichen, sondern *alle* möglichen Flüsse bestimmt werden. Dies ist notwendig, da vorab nicht entschieden werden kann, welche Zustände erreicht werden und welche nicht. Die Aufgabe des Model Checkings ist es ja, genau diese Mengen zu bestimmen.

Den allgemeinen Ablauf der Flussberechnung zeigt Abb. 3.7. Nach der Initialisierung bestimmter Datenstrukturen (siehe Abschnitt 3.2.4.1), erfolgt die Berechnung der *einfachen Flüsse*. Als *einfache Flüsse* werden in diesem Zusammenhang die Flüsse bezeichnet, die von einem Knoten zu einem anderen Knoten ohne mögliche Zwischenpufferung fließen. Eine ausführlichere Beschreibung findet sich in Abschnitt 3.2.4.3.

Schließlich werden noch die möglichen Kombinationen der einfachen Flüsse bestimmt. Die *kombinierten Flüsse* sind Flüsse, die auf Grund der gegebenen Tokenkonfiguration zumindest teilweise ohne Pufferung auskommen. Details hierzu finden sich in Abschnitt 3.2.4.4.

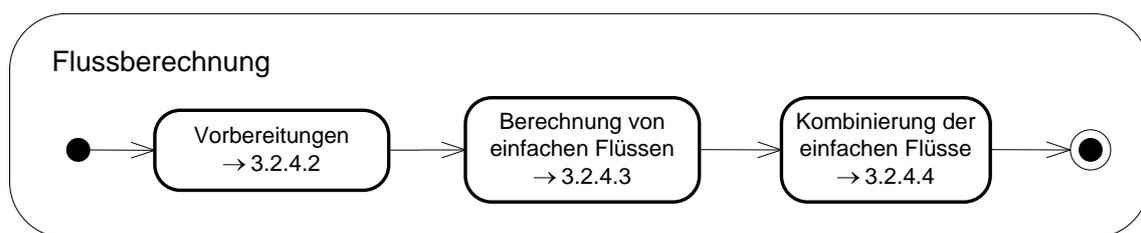


Abbildung 3.7: Einzelne Schritte der Flussberechnung.

3.2.4.1 Datenstruktur

Zunächst wird die Datenstruktur vorgestellt, in der Flüsse gespeichert werden. Die Darstellung als Zusammenfassung mehrerer Funktionen in der ASM-Domäne *Flow* vermeidet die Wiederholung der Domäne *Flow* als Parameter für jede „innere Funktion“.

```

controlled domain Flow =def {
  actionsOffering:  $\mathcal{P}(\text{Action})$ ;
  forksOffering:  $\mathcal{P}(\text{ActivityNode} \times \text{ActivityEdge})$ ;
  objectNodesOffering:  $\mathcal{P}(\text{ObjectNode})$ ;
  guards:  $\mathcal{P}(\text{GuardSpecification})$ ;

  actionsStart:  $\mathcal{P}(\text{Action})$ ;
  forksBuffering:  $\mathcal{P}(\text{ActivityNode} \times \text{ActivityEdge})$ ;
  objectNodesBuffering:  $\mathcal{P}(\text{ObjectNode})$ ;

  regionsEntered:  $\mathcal{P}(\text{InterruptibleActivityRegion})$ ;
  regionsInterrupted:  $\mathcal{P}(\text{InterruptibleActivityRegion})$ ;

  dataFlows:  $\mathcal{P}(\text{ActivityNode} \times \text{ActivityNode})$ 
}

```

Die Funktionen lassen sich in vier Abschnitte aufteilen. Der erste Teil (bis *guards*) definiert die Bedingungen, die erfüllt sein müssen, damit dieser Fluss fließen kann und wird daher auch „Flussvorbereitung“ genannt.

Die restlichen Teile beschreiben die Veränderungen am Systemzustand, die durch einen Fluss ausgelöst werden. Sie werden daher auch mit dem Begriff „Flusseffekt“ zusammengefasst. Die zur Definition der \langle Guards \rangle verwendete Domäne *GuardSpecification* ist in Anhang A.1.1 definiert und beschreibt abstrakt die Spezifikation einer Bedingung.

Die Flussvorbereitung enthält einerseits Knoten, die Tokens anbieten müssen (Aktionen \rightarrow *actionsOffering*, *ForkNodes* \rightarrow *forksOffering* und *ObjectNodes* \rightarrow *objectNodesOffering*), andererseits aber auch die Menge der Bedingungen, die an den Kanten entlang des Pfades notiert sind (*guards*) und zu *true* evaluieren müssen.

Der zweite Teil (bis *objectNodesBuffering*) beschreibt die Knoten, die entsprechend des Flusses gestartet werden können oder Tokens puffern müssen. Wiederum sind dies Aktionen (*actionsStart*), *ForkNodes* (*forksBuffering*) und *ObjectNodes* (*objectNodesBuffering*).

Der dritte Teil speichert die *InterruptibleActivityRegions*, die entlang des Pfades unterbrochen (*regionsInterrupted*) und betreten (*regionsEntered*) werden. Eine Zuordnung von Datenquellen auf Datensinken wird im letzten Teil (*dataFlows*) gespeichert.

Für den Fluss aus Abb. 3.6 aus dem vorherigen Abschnitt 3.2.3 würde der Inhalt der eben vorgestellten Datenstruktur wie in Tabelle 3.2 dargestellt aussehen.

Als kompaktere Darstellung wird folgende Schreibweise eingeführt, bei der zwischen der Flussbedingung und dem Flusseffekt ein Pfeil geschrieben wird und die Mengen entsprechend ihrer Definitionsreihenfolge in Tupeln zusammengefasst werden:

$$(\{A\}, \emptyset, \emptyset, \emptyset) \rightarrow (\{B, C\}, \emptyset, \emptyset, \emptyset, \emptyset, \{(o1, i1)\})$$

3.2.4.2 Vorbereitungen

Vor der eigentlichen Berechnung werden einige Informationen aus den Diagrammen gesammelt, die innerhalb der Berechnung wieder verwendet werden. Außerdem werden zusätzliche Knoten erzeugt, die die Berechnung vereinfachen bzw. für das Model Checking notwendig sind.

Laut UML Spezifikation werden an den ausgehenden Kanten von *ForkNodes* angebotene, aber nicht verwendete Tokens gepuffert (siehe auch Abschnitt 2.3.2). Dies gilt insbesondere

Tabelle 3.2: Wertebelegung der *Flow*-Datenstruktur für Beispiel aus Abb. 3.6.

Funktion	Wert
<i>actionsOffering</i>	{A}
<i>forksOffering</i>	\emptyset
<i>objectNodesOffering</i>	\emptyset
<i>guards</i>	\emptyset
<i>actionsStart</i>	{B, C}
<i>forksBuffering</i>	\emptyset
<i>objectNodesBuffering</i>	\emptyset
<i>regionsEntered</i>	\emptyset
<i>regionsInterrupted</i>	\emptyset
<i>dataFlows</i>	{(o1, i1)}

auch für Objekttokens und daher kann eine ausgehende *ForkNode*-Kante auch als Datenquelle auftreten.

Das Makro **CREATEFORKEDGEBUFFERS** erzeugt für jede ausgehende Kante eines *ForkNodes* einen *ObjectNode*. Dieser neue Knoten wird dann der Funktion *forkEdgeToObjectNode* an der entsprechenden Stelle zugewiesen. Außerdem wird der neue *ObjectNode* innerhalb des gleichen Unterbrechungsbereichs angesiedelt wie der ursprüngliche Knoten. Diese Information ist im weiteren Verlauf für die Zustandsübergangsberechnung wichtig.

Wenn eine Aktion mehrere ausgehende Kanten besitzt, wird ein implizites Fork-Verhalten angenommen (siehe unten). Daher wird auch für den einen (siehe Normalisierung in Abschnitt 3.2.2) ausgehenden Kontrollfluss ein *ObjectNode* erstellt, der dann sogenannte Nulltokens puffern kann.

controlled *forkEdgeToObjectNode* : *ActivityNode* \times *ActivityEdge* \rightarrow *ObjectNode*

CREATEFORKEDGEBUFFERS \equiv

```

forall n with n  $\in$  ForkNodes
  forall e with e  $\in$  n.outgoing
    let
      o = new(ObjectNode)
      o.inInterruptibleRegion = n.inInterruptibleRegion
    in
      forkEdgeToObjectNode(n, e) := o
forall n with n  $\in$  Action
  if multipleOutFlows(n)  $\wedge$  |n.outgoing| = 1 then
    let
      o = new(ObjectNode)
      o.inInterruptibleRegion = n.inInterruptibleRegion
    in
      forkEdgeToObjectNode(n, outgoing(n, 1)) := o

```

Für die Berücksichtigung der *lower* und *upper* Werte, die eine untere und eine obere

Grenze für die gleichzeitig zu verarbeitenden Tokens angeben, ist es hilfreich, die Menge der möglichen Datenquellen für eine Datensenke zu wissen. Diese Menge pro Knoten berechnet das Makro `CALCULATEPOSSIBLEDATASOURCES`.

Es ruft dazu für jede mögliche Datensenke, die auch *lower* oder *upper* Werte besitzen kann, für jede eingehende Kante das Makro `CALCULATEPOSSIBLEDATASOURCESREC` auf. Darin werden die möglichen Pfade entgegen der Pfeilrichtung rekursiv verfolgt und die Ergebnismenge zurückgeliefert. Diese Menge wird dann der Funktion `possibleDataSources` an der Stelle des Knotens zugewiesen.

controlled `possibleDataSources` : $ActivityNode \rightarrow \mathcal{P}(ObjectNode)$

```

CALCULATEPOSSIBLEDATASOURCES  $\equiv$ 
forall p with p  $\in$  InputPin  $\cup$  CentralBufferNode  $\cup$  ActivityParameterNode
forall e with e  $\in$  p.incoming
let
  r = CALCULATEPOSSIBLEDATASOURCESREC(e)
in
  possibleDataSources(p) := possibleDataSources(p)  $\cup$  r

```

Die Rekursion in `CALCULATEPOSSIBLEDATASOURCESREC` wird beendet, sobald auf eine Datenquelle gestoßen wird (Zeilen 7–8). Ein Puffer eines *ForkNodes* wird ebenfalls als Datenquelle betrachtet (Zeilen 9–13). Nur für *Merge*-, *Join*- und *DecisionNodes* wird das Makro für jede eingehende Kante rekursiv aufgerufen.

Die Terminierung der Rekursion ist sichergestellt, da keine Zyklen zwischen Kontrollknoten erlaubt sind. Da außerdem von einem syntaktisch korrekten Diagramm ausgegangen wird, trifft der Algorithmus immer auf einen *ObjectNode*. Diese beiden Voraussetzungen stellen auch für die restlichen Makros deren Terminierung sicher.

```

1 CALCULATEPOSSIBLEDATASOURCESREC :  $ActivityEdge \rightarrow \mathcal{P}(ObjectNode)$ 
2 CALCULATEPOSSIBLEDATASOURCESREC(edge)  $\equiv$ 
3 local r :  $\mathcal{P}(ObjectNode)$ 
4 let
5   node = edge.source
6 in
7   if node  $\in$  OutputPin  $\cup$  CentralBufferNode  $\cup$  ActivityParameterNode then
8     add node to r
9   if node  $\in$  ForkNode then
10    let
11      res = forkEdgeToObjectNode(node, edge)
12    in
13      add res to r
14   if node  $\in$  MergeNode  $\cup$  JoinNode  $\cup$  DecisionNode then
15     forall e with e  $\in$  node.incoming
16     let
17       res = CALCULATEPOSSIBLEDATASOURCESREC(e)
18     in
19       add res to r

```

20 seq
 21 result := r

3.2.4.3 Berechnung der einfachen Flüsse

Nach den im vorherigen Abschnitt vorgestellten Vorarbeiten, kann die Flussberechnung durchgeführt werden. Es werden zunächst „einfache“ Flüsse berechnet. Unter „einfachen“ Flüssen werden Flüsse betrachtet, die unmittelbar von einer Tokenquelle zu einer Tokensenke führen. Insbesondere sind damit Flüsse ausgeschlossen, die über einen internen Puffer (wie z. B. bei *ForkNodes*) fließen.

Abb. 3.8 zeigt ein Aktivitätsdiagramm, das im Folgenden zur Verdeutlichung einiger Zusammenhänge herangezogen wird. Für eine eindeutige Zuordnung wurden die beiden *ForkNodes* jeweils mit einem Bezeichner (F1 bzw. F2) versehen.

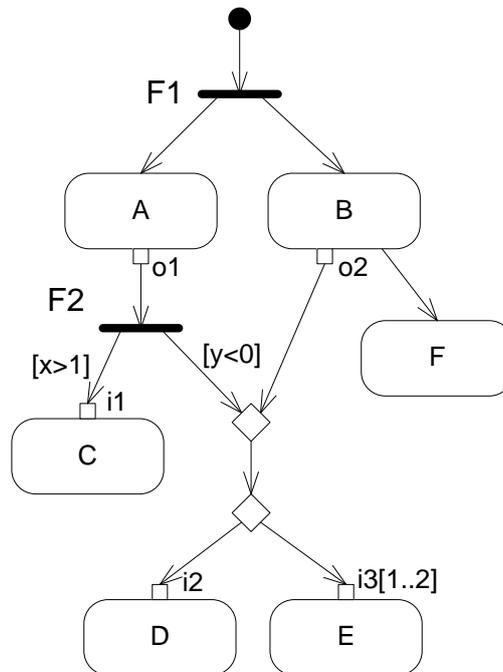


Abbildung 3.8: Aktivitätsdiagrammbeispiel zur Erläuterung der Flussberechnung.

Einfache Flüsse sind in diesem Beispiel der Fluss vom *InitialNode* zum *ForkNode* F1 und der Fluss vom *ForkNode* F1 zur Aktion A oder Aktion B. Der Fluss vom *InitialNode* zu den Aktionen A und B allerdings nicht, da er über die *ForkNode*-Pufferung hinweggeht.

Diese sogenannten „kombinierten“ Flüsse werden erst in einem nächsten Schritt durch Verknüpfung der verschiedenen *ForkNodes* bestimmt (siehe Abschnitt 3.2.4.4). Diese Vorgehensweise vermeidet zum einen die mehrfache Berechnung von Flüssen (insbesondere in Zusammenhang mit *JoinNodes*, siehe unten), zum anderen eine im Nachhinein komplizierte Bestimmung der über Zwischenpuffer möglichen Flüsse.

CompSimpleFlows. Das Makro **COMP SIMPLE FLOWS** übernimmt die Berechnung der einfachen Flüsse. Es ruft dazu für jeden Knotentyp, der überhaupt ausgehende Kanten haben kann, jeweils wieder spezielle Makros auf.

```

COMP SIMPLE FLOWS : void →  $\mathcal{P}(\text{Flow})$ 
COMP SIMPLE FLOWS ≡
  local r:  $\mathcal{P}(\text{Flow})$  := ∅
  forall a with a in Activity
    forall n with n in a.node
      if n ∈ InitialNode then
        let
          simpleFlows = COMP SIMPLE FLOW FOR INITIAL NODE(n)
        in
          add simpleFlows to r
      if n ∈ Action then
        let
          simpleFlows = COMP SIMPLE FLOW FOR ACTION(n)
        in
          add simpleFlows to r
      if n ∈ CentralBufferNode ∪ ActivityParameterNode then
        let
          simpleFlows = COMP SIMPLE FLOW FOR OBJECT NODE(n)
        in
          add simpleFlows to r
  seq
  result := r

```

Zur Übersicht zeigt Abb. 3.9 einen Ausschnitt aus dem Aufrufgraph der Makros. Ein Pfeil bedeutet dabei, dass ein Makro (Pfeilanzug) ein anderes (Pfeilende) aufruft. Es wird keine Aussage über die Anzahl oder Reihenfolge der Aufrufe getroffen. Auch Rückgabewerte oder ähnliches bleiben unberücksichtigt.

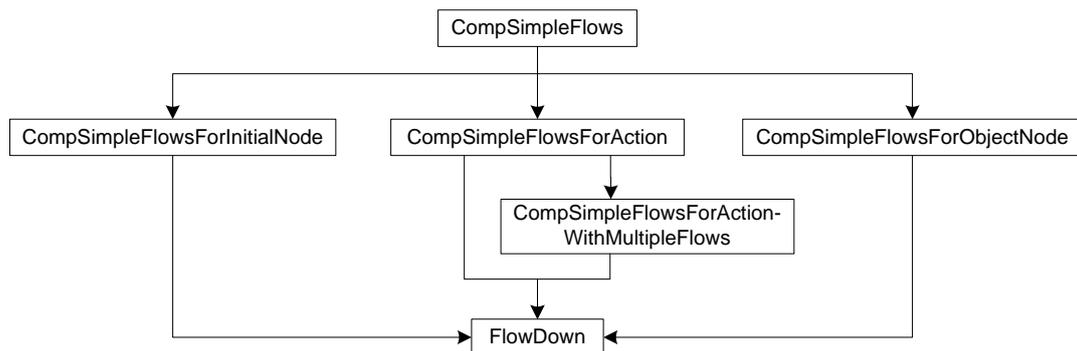


Abbildung 3.9: Ausschnitt aus dem Aufrufgraphen für das Makro **COMP SIMPLE FLOWS**.

Wie man sieht, rufen alle Makros, die von **COMP SIMPLE FLOWS** aufgerufen werden, schließlich das Makro **FLOW DOWN** auf. Dieses Makro bildet das Kernstück der Flussberechnung und wird ausführlich ab Seite 49 beschrieben. Zunächst werden die Submakros des Makros **COMP**

SIMPLEFLOWS erläutert.

CompSimpleFlowForInitialNode. Für *InitialNodes* ist die Berechnung einfach, da durch die Normalisierung (siehe Abschnitt 3.2.2) sichergestellt ist, dass es nur eine ausgehende Kante gibt. Ein neu erzeugter Fluss (Zeile 4) wird mit der Startinformation gefüllt (Zeile 7).

Dieser wird dann zusammen mit der einzigen ausgehenden Kante dem Makro **FLOWDOWN** übergeben, welches dieser Kante in Pfeilrichtung folgt und die Fluss-Datenstruktur weiter füllt.

```

1 COMPSIMPLEFLOWFORINITIALNODE : InitialNode →  $\mathcal{P}(\text{Flow})$ 
2 COMPSIMPLEFLOWFORINITIALNODE(node) ≡
3 let
4   flow = new(Flow)
5   edge = outgoing(node,1)
6 in
7   add node to flow.actionsOffering
8   seq
9   result := FLOWDOWN(edge, flow)

```

CompSimpleFlowForObjectNode. Für *ObjectNodes* ist die Berechnung nur unwesentlich komplizierter als für *InitialNodes*. Es wird diesmal allerdings für jede ausgehende Kante ein eigener Fluss erzeugt (Zeile 6), da sich die Tokenangebote an ausgehenden Kanten von *ObjectNodes* gegenseitig ausschließen (siehe auch [Sar06a, Seite 62]). Außerdem wird der Knoten nicht nur der Menge der Knoten, die ein Token anbieten müssen, hinzugefügt (Zeile 8), sondern auch in der Menge der Datenflüsse als Datenquelle eingetragen (Zeile 9).

```

1 COMPSIMPLEFLOWFOROBJECTNODE : ObjectNode →  $\mathcal{P}(\text{Flow})$ 
2 COMPSIMPLEFLOWFOROBJECTNODE(node) ≡
3 local r :  $\mathcal{P}(\text{Flow})$  :=  $\emptyset$ 
4 forall e with e ∈ node.outgoing
5   let
6     flow = new(Flow)
7   in
8     add node to flow.objectNodesOffering
9     add (node, undef) to flow.dataFlows
10    seq
11    let
12      s = FLOWDOWN(e, flow)
13    in
14      add s to r
15    seq
16    result := r

```

CompSimpleFlowForAction. Für Aktionen gestaltet sich die Berechnung des Flusses schwieriger, da unterschieden werden muss, ob die Aktion nur einen oder mehrere ausgehende

Flüsse besitzt, wie es im obigen Beispiel (Abb. 3.8) für Aktion B der Fall ist.

Wie Abb. 3.10 symbolisiert, werden mehrere ausgehende Kanten wie ein *ForkNode* (vgl. [Sar06a, Seite 47]) behandelt. Die Verarbeitung unterscheidet sich allerdings ein wenig von diesen, da an dieser Stelle Kontroll- und Datenflüsse gemischt auftreten können. Dies ist bei „normalen“ *ForkNodes* nicht möglich. Zur Unterscheidung werden mehrere ausgehende Kanten im Folgenden als „impliziter *ForkNode*“ bezeichnet. Mehrere eingehende Kanten werden analog als „impliziter *JoinNode*“ bezeichnet.

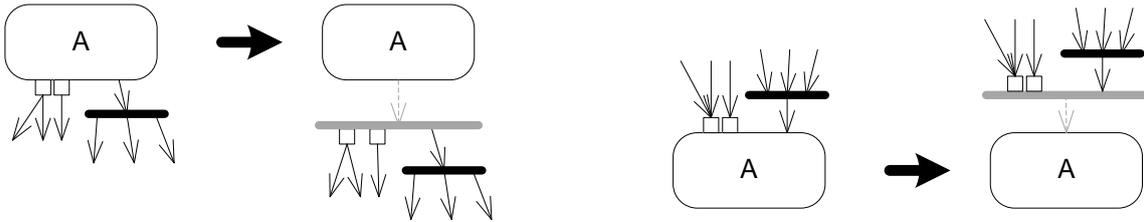


Abbildung 3.10: Mehrere aus- bzw. eingehende Flüsse werden implizit wie ein *Fork-* bzw. *JoinNode* behandelt.

Bei impliziten *ForkNodes* werden die ausgehenden Flüsse getrennt voneinander betrachtet. Es kann zu einer Pufferung an einer Kante kommen, falls eine nachfolgende Aktion noch nicht bereit ist, das Token entgegenzunehmen.

Das Makro `COMPSIMPLEFLOWFORACTION` prüft zunächst, ob ein impliziter *ForkNode* vorliegt und ruft gegebenenfalls das Makro `COMPSIMPLEFLOWFORACTIONWITHMULTIPLEFLOWS` auf.

Ist dies nicht der Fall, wird im weiteren Verlauf unterschieden, ob die eine ausgehende Kante ein Kontroll- oder ein Objektfluss ist. Hier reicht eine Überprüfung auf die Menge der ausgehenden Kanten, da Objektflüsse nur von *OutputPins* ausgehen können (Zeile 6).

Ist die Kante Teil eines Kontrollflusses, wird der Fluss ähnlich wie bei *InitialNodes* bestimmt. Falls die Kante allerdings Teil eines Objektflusses ist (ab Zeile 14), wird zur Flussberechnung wie bei Objektknoten vorgegangen, wobei der entsprechende *OutputPin* als Datenquelle vermerkt wird (Zeile 17).

```

1 COMPSIMPLEFLOWFORACTION : Action →  $\mathcal{P}(\textit{Flow})$ 
2 COMPSIMPLEFLOWFORACTION(node ≡
3   if multipleOutFlows(node) then
4     result ← COMPSIMPLEFLOWFORACTIONWITHMULTIPLEFLOWS
5   else
6     if node.outgoing ≠ ∅ then
7       let
8         flow = new(Flow)
9         edge = outgoing(node,1)
10      in
11        add node to flow.actionsOffering
12        seq
13          result := FLOWDOWN(edge, flow)
14      if |node.output| = 1 then
15        local r :  $\mathcal{P}(\textit{Flow})$  := ∅

```

```

16  let
17    o = output(node, 1)
18  in
19    forall e with e ∈ o.outgoing
20      let
21        flow = new(Flow)
22      in
23        add node to flow.actionsOffering
24        add (o, undef) to flow.dataFlows
25      seq
26      let
27        s = FLOWDOWN(e, flow)
28      in
29        add s to r
30    seq
31    result := r

```

CompSimpleFlowForActionWithMultipleFlows. Die Flüsse von *impliziten ForkNodes* werden sehr ähnlich zu einem „normalen“ *ForkNode* bestimmt. Auch dafür werden in dieser Phase nur die einfachen Flüsse berechnet.

Einfache Flüsse sind in diesem Fall der Fluss von der Aktion auf die *OutputPins* und den Kontrollkantenpuffer und dann von den *OutputPins* bzw. dem Kontrollkantenpuffer ausgehend alle möglichen Flüsse zu nachfolgenden Aktionen.

Der erste Fluss wird im Makro **COMPSIMPLEFLOWFORACTIONWITHMULTIPLEFLOWS** in den Zeilen 4–14 erzeugt. Im restlichen Teil wird wieder zwischen Kontrollfluss (Zeilen 15–24) und Objektfluss (Zeilen 25–39) unterschieden und die entsprechenden Flüsse mit Hilfe des Makros **FLOWDOWN** berechnet.

```

1  COMPSIMPLEFLOWFORACTIONWITHMULTIPLEFLOWS : Action → P(Flow)
2  COMPSIMPLEFLOWFORACTIONWITHMULTIPLEFLOWS(node) ≡
3  local res : P(Flow) := ∅
4  let
5    bufferFlow = new(Flow)
6  in
7    add node to bufferFlow.actionsOffering
8    if node.outgoing ≠ ∅ then
9      add (node, outgoing(node,1)) to bufferFlow.forksBuffering
10   forall o with o in node.output
11     forall e with e in o.outgoing
12       add (o, e) to bufferFlow.forksBuffering
13   seq
14   add bufferFlow to res
15  if node.outgoing ≠ ∅ then
16   let
17     flow = new(Flow)
18   in

```

```

19   add (node, outgoing(node,1)) to flow.forksBuffering
20   seq
21   let
22     r = FLOWDOWN(outgoing(node,1), flow)
23   in
24     add r to res
25 forall o with o in node.output
26   local r :  $\mathcal{P}(\textit{Flow}) := \emptyset$ 
27   forall e with e in o.outgoing
28     let
29       flow = new(Flow)
30     in
31       add (o, e) to flow.forksOffering
32       add (o, undef) to flow.dataFlows
33       seq
34       let
35         s = FLOWDOWN(e, flow)
36       in
37         add s to r
38     seq
39     add r to res
40   seq
41   result := res

```

FlowDown. Eine zentrale Rolle bei der Berechnung der einfachen Flüsse spielt das Makro **FLOWDOWN**, das von allen bisher vorgestellten Makros verwendet wird. Es übernimmt die Aufgabe, ausgehend von der übergebenen Fluss-Datenstruktur und einer aktuellen Kante, das Aktivitätsdiagramm rekursiv zu durchlaufen. Dabei wird die ursprüngliche Datenstruktur entsprechend gefüllt bzw. kopiert und schließlich eine Menge der möglichen Flüsse zurückgeliefert. Das Rekursionsende bestimmt sich dabei nicht einfach, da z. B. bei *JoinNodes* auch die eingehenden Flüsse entgegen der Pfeilrichtung betrachtet werden.

Die vereinfachte Aufrufstruktur für den folgenden Teil der Flussberechnung zeigt Abb. 3.11. Zur besseren Nachvollziehbarkeit ist auch dieses Makro in mehrere Makros aufgeteilt, die das Verhalten beim Erreichen von unterschiedlichen Knotentypen beschreiben. Für das Makro **FLOWDOWN** sind das **DECISIONFLOWDOWN**, **JOINFLOWDOWN**, **FORKFLOWDOWN**, **ACTIONFLOWDOWN** und **OBJECTNODEFLOWDOWN**.

Die Behandlung des *MergeNodes* besteht lediglich aus dem rekursiven Aufruf des Makros **FLOWDOWN** und wurde daher nicht in ein eigenes Makro ausgelagert. Die Submakros der drei restlichen Kontrollknoten (*Join*-, *Fork*- und *DecisionNode*) rufen wiederum das Makro **FLOWDOWN** auf. **FORKFLOWDOWN** benötigt außerdem das Makro **CREATEFLOWCOMBINATIONSFORFORKNODE** zur Bestimmung der Flusskombinationen (siehe bei der Beschreibung weiter unten).

Die Makros zur Behandlung von *JoinNodes*, Aktionen und *ObjectNodes* rufen das Makro **FLOWUP** auf, da in diesen Situationen auch die eingehenden Kanten in die Berechnung einbezogen werden müssen. **FLOWUP** wiederum wird in den meisten Fällen (*Join*-, *Merge*-, *DecisionNode*) direkt oder indirekt rekursiv aufgerufen. Nur eine Aktion, ein *OutputPin* oder

ein *ForkNode* markieren das Rekursionsende.

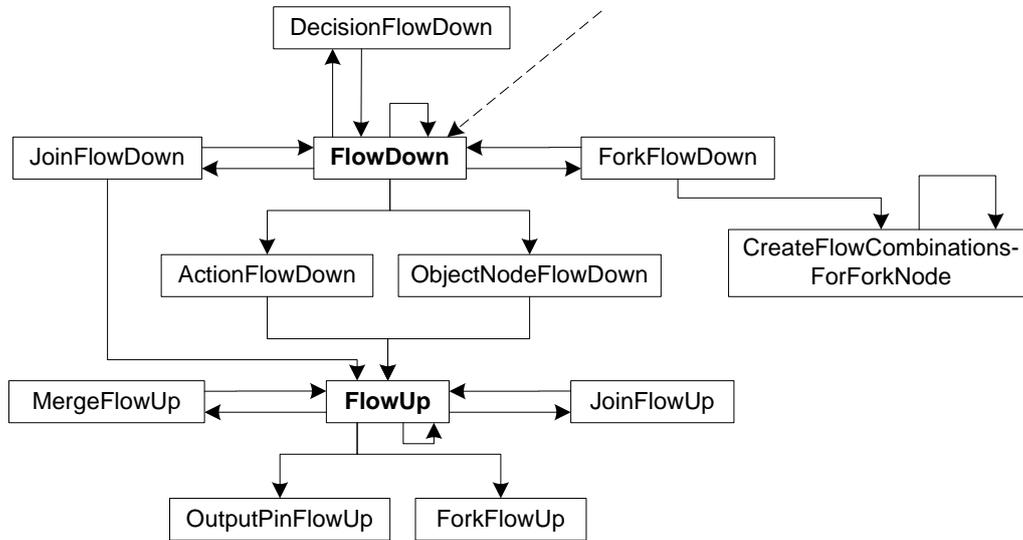


Abbildung 3.11: Ausschnitt aus dem Aufrufgraphen für die Makros **FLOWDOWN** und **FLOWUP**. Der gestrichelte Pfeil deutet an, dass zunächst das Makro **FLOWDOWN** aufgerufen wird.

Im Folgenden wird zunächst auf das Makro **FLOWDOWN** detailliert eingegangen.

Vor der Aufteilung in Submakros werden einige Punkte behandelt, die für alle Knotentypen gleich sind, da sie die verfolgte Kante betreffen. Zum einen werden in Zeile 9 an der Kante notierte Bedingungen in den aktuellen Fluss eingefügt, zum anderen die betretenen und unterbrochenen *InterruptibleActivityRegions* der Datenstruktur hinzugefügt (Zeilen 10–12).

Unterbrochene *InterruptibleActivityRegions* sind direkt mit der Kante verknüpft. Die Menge der betretenen Bereiche muss aus der Differenzmenge der Bereiche, in denen der Anfang der Kante liegt und den Bereichen, in welchen sich das Ziel der Kante befindet, berechnet werden (Zeilen 6–7). In Abb. 3.12 werden beispielsweise zwei Unterbrechungsbereiche abgebrochen und einer betreten. Der größte Unterbrechungsbereich wird nicht neu betreten, da die Quelle der Kante sich schon in diesem befindet.

Daran schließt sich die Fallunterscheidung in Abhängigkeit des Zielknotens der aktuellen Kante an. Für die meisten Knotentypen werden wiederum neue Makros aufgerufen, nur zwei einfache Fälle werden unmittelbar behandelt.

Einerseits bildet ein *FinalNode* ein Ende der Rekursion. Der Knoten wird der Funktion *actionsStart* der Fluss-Datenstruktur zugewiesen und der komplettierte Fluss zurückgeliefert (Zeilen 13–15). Andererseits wird der Fluss bei einem *MergeNode* lediglich an die eine ausgehende Kante durchgereicht (Zeilen 20–21).

Die Verarbeitung eines *DecisionNodes* ist ähnlich einfach. Es wird für jede ausgehende Kante für jeden Datenfluss das Makro **FLOWDOWN** aufgerufen. Daher wird hier auf eine nähere Erläuterung verzichtet und auf Anhang A.2 verwiesen. Die restlichen Submakros werden in den folgenden Absätzen genauer erläutert.

- 1 **FLOWDOWN** : $ActivityEdge \times Flow \rightarrow \mathcal{P}(Flow)$
- 2 **FLOWDOWN**($edge, flow$) \equiv
- 3 **local** $r : \mathcal{P}(Flow) := \emptyset$

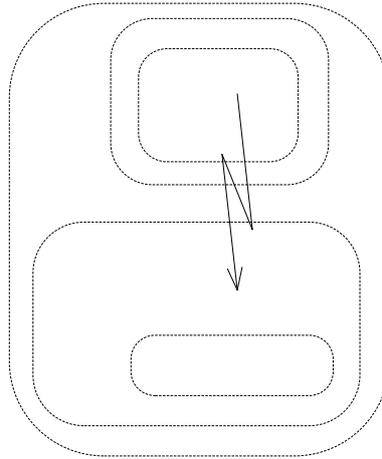


Abbildung 3.12: Beispiel von unterbrochenen und betretenen *InterruptibleActivityRegions*.

```

4  let
5    node : ActivityNode = edge.target
6    diffRegions :  $\mathcal{P}(\text{InterruptibleActivityRegion})$  =
7      edge.target.inInterruptibleRegion \ edge.source.inInterruptibleRegion
8  in
9    add edge.guard to flow.guards
10   forall region with region ∈ edge.interrupts
11     add region to flow.regionsInterrupted
12   add diffRegions to flow.regionsEntered
13   if node ∈ FinalNode then
14     add node to flow.actionsStart
15     add flow to r
16   if node ∈ Action then
17     r := ACTIONFLOWDOWN(node, flow)
18   if node ∈ ObjectNode then
19     r := OBJECTNODEFLOWDOWN(node, flow)
20   if node ∈ MergeNode then
21     r := FLOWDOWN(outgoing(node,1), flow)
22   if node ∈ DecisionNode then
23     r := DECISIONFLOWDOWN(node, flow)
24   if node ∈ ForkNode then
25     r := FORKFLOWDOWN(node, flow)
26   if node ∈ JoinNode then
27     r := JOINFLOWDOWN(node, edge, flow)
28   seq
29   result := r

```

ForkFlowDown. Ein *ForkNode* erfordert mehr Aufwand, da die UML vorschreibt (vgl. [Sar06a, Seite 67]), dass Kanten, deren \langle Guard \rangle nicht wahr ist, keine Tokens angeboten be-

kommen. Daher dürfen für diesen Fall auch keine Tokens an den entsprechenden Kanten gepuffert werden. Diese Sonderbehandlung ist bei impliziten Forks nicht erforderlich, da dort keine Bedingungen angegeben werden können.

Um diesen Aspekt im Rahmen des Model Checkings zu erfassen, müssen für jede mögliche Kombination an wahren und nicht wahren \langle Guards \rangle eigene Flüsse erzeugt werden. Dies erledigt das Makro `CREATEFLOWCOMBINATIONSFORFORKNODE`. Es bekommt den *ForkNode*, den bisherigen Fluss und die Nummer der ersten ausgehenden Kante übergeben.

Das letzte Argument ist notwendig, da das Makro rekursiv arbeitet, um alle Kombinationsmöglichkeiten zu erstellen. In diesem Argument wird die Position innerhalb der Liste der ausgehenden Kanten vermerkt. Eine genaue Beschreibung dieses Makros findet sich in Anhang A.4.

Im Folgenden wird die Funktionsweise für einen Ausschnitt aus Abb. 3.8 mit dem *ForkNode* F2 beispielhaft dargestellt. Abb. 3.13 zeigt die zu erstellenden unterschiedlichen Flüsse für dieses Beispiel. Die grün gezeichneten Tokens symbolisieren eine Pufferung des Tokens an den entsprechenden Kanten.

Es werden insgesamt vier Flüsse von Aktion A aus erzeugt: ein Fluss, bei dem die Bedingungen $[x > 1]$ und $[y < 0]$ wahr sind und folglich beide Puffer gefüllt werden können (Fall a)), einer mit den zwei negierten Bedingungen und keinem Effekt (Fall d)) und schließlich jeweils ein Fluss, bei dem eine Bedingung wahr und die andere negiert ist und jeweils nur ein Puffer belegt wird (Fall b) und c)).

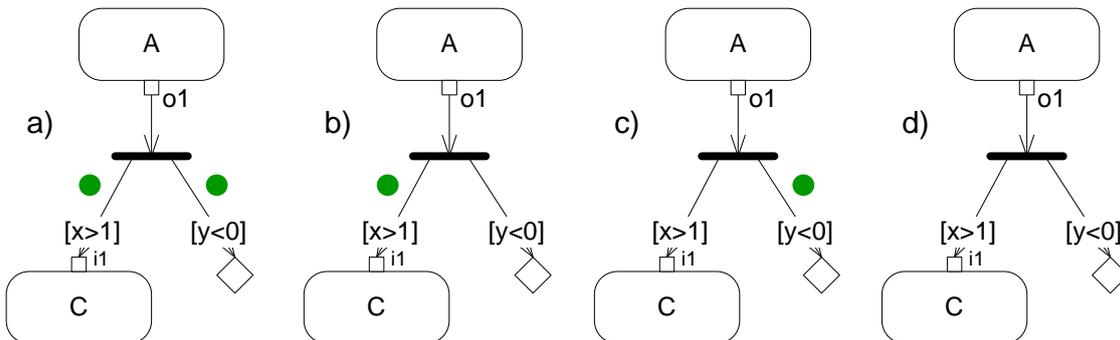


Abbildung 3.13: Ausschnitt aus Abb. 3.8 mit Darstellung der möglichen Flüsse.

Von diesen Puffern ausgehend, werden noch Flüsse von *ForkNode* F2 zu Aktion C und Richtung *MergeNode* erstellt. Der Fluss in Richtung *MergeNode* wird im weiteren Verlauf am *DecisionNode* wieder in zwei Flüsse aufgespalten und führt einmal zu Aktion D und einmal zu Aktion E.

Der restliche Ablauf des Makros `FORKFLOWDOWN` gestaltet sich folgendermaßen: Nach der Berechnung der einfachen Flüsse von einer Tokenquelle zu den Puffern eines *ForkNodes* unter Berücksichtigung der \langle Guards \rangle , werden, ausgehend vom *ForkNode*, für jede ausgehende Kante ein neuer Fluss kreiert und der Kante entsprechend gefolgt (ab Zeile 8). Falls es sich bei der Kante um einen Objektfluss handelt, muss auch noch die Menge der Datenflüsse auf einen Startwert gesetzt werden (Zeilen 14f).

¹ `FORKFLOWDOWN` : $ActivityNode \times Flow \rightarrow \mathcal{P}(Flow)$

² `FORKFLOWDOWN`(*node*, *flow*) \equiv

```

3  local  $r : \mathcal{P}(\text{Flow}) := \emptyset$ 
4  let
5    bufferFlows = CREATEFLOWCOMBINATIONSFORFORKNODE( $node$ , flow, 1)
6  in
7    add bufferFlows to  $r$ 
8  foreach  $e$  with  $e \in node.outgoing$ 
9    let
10     newFlow : Flow = new(Flow)
11     onode : ObjectNode = forkEdgeToObjectNodes( $node$ ,  $e$ )
12   in
13     add ( $node$ ,  $e$ ) to newFlow.forksOffering
14     if  $e \in \text{ObjectFlow}$  then
15       add ( $onode$ , undef) to newFlow.dataFlows
16     seq
17     let
18       s = FLOWDOWN( $e$ , newFlow)
19     in
20       add  $s$  to  $r$ 
21   seq
22 result :=  $r$ 

```

JoinFlowDown. Ein *JoinNode* bedeutet eine Synchronisierung der eingehenden Flüsse. Der *JoinNode* kann also nur dann Tokens weiterleiten, wenn an allen eingehenden Kanten jeweils mindestens ein Token anliegt. In diesem Fall werden *alle* Objekttokens oder — falls nur Kontrolltokens anliegen — *ein* Kontrolltoken weitergeleitet. Zur Erläuterung der Flussberechnung bei *JoinNodes* wird ein weiteres Beispiel in Abb. 3.14 eingeführt.

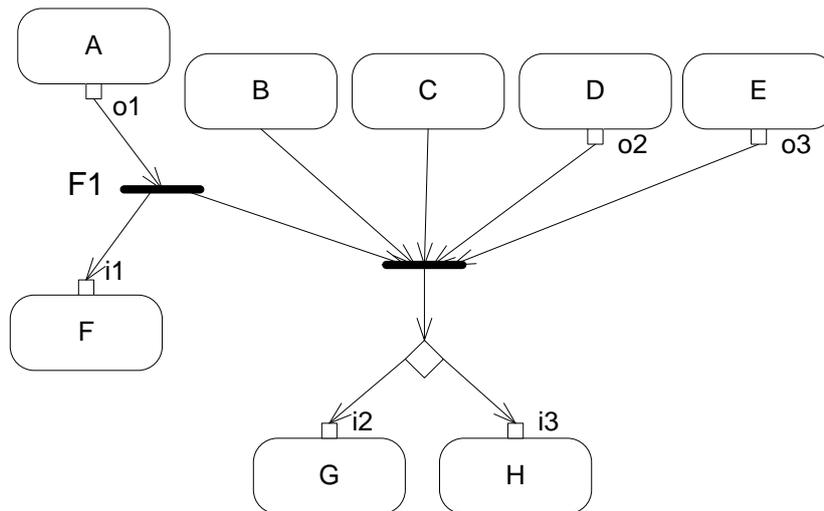


Abbildung 3.14: Aktivitätsdiagrammausschnitt mit Join- und *ForkNodes*.

Damit der *JoinNode* in diesem Beispiel schalten kann, muss von Aktion B und C jeweils

ein Kontrolltoken anliegen, von den Aktionen D und E jeweils ein Objekttoken und, da in dieser Phase der Flussberechnung nur einfache Flüsse betrachtet werden, ein Objekttoken aus dem *ForkNode*-Puffer. Die beiden Kontrolltokens werden ignoriert, aber alle drei Objekttokens weitergereicht.

Nach dem *DecisionNode* können diese Tokens dann entweder der Aktion G oder H angeboten werden. Es ist nicht möglich, dass beispielsweise das Token von o2 der Aktion G und das Token von o3 der Aktion H zugeordnet wird und somit beide Aktionen starten.

Die Semantik der UML sieht folgende Betrachtungsweise vor: Sobald eine Aktion ein Token konsumiert hat, liegt es nicht mehr am Eingang des *JoinNodes* an und demnach kann der *JoinNode* nicht „ein weiteres Mal“ schalten. In der formalen Semantik von Sarstedt [Sar06a, Seite 74ff] wird dieser Fall behandelt, indem nach einer Tokenauswahl die Tokenangebote insbesondere an *JoinNodes* komplett neu berechnet werden.

Für die Flussberechnung in dieser Arbeit bedeutet dies, dass, sobald während der Flussberechnung auf einen *JoinNode* gestoßen wird, alle anderen Kanten entgegengesetzt ihrer Richtung verfolgt werden (**FLOWUP**), bis wieder auf eine Aktion, einen ObjektNode oder einen *ForkNode* gestoßen wird.

Die Ergebnisflüsse der einzelnen Berechnungen werden mit Hilfe des Makros **ELEMENTWISEUNION** elementweise kombiniert. Durch diese Kombination werden die notwendigen Voraussetzungen zur Schaltung des betrachteten *JoinNodes* erfasst. Für eine ausführliche Beschreibung dieses Makros wird auf Anhang A.4 verwiesen.

```

1 JOINFLOWDOWN : ActivityNode × ActivityEdge × Flow →  $\mathcal{P}(\textit{Flow})$ 
2 JOINFLOWDOWN(node, edge, flow) ≡
3   local r :  $\mathcal{P}(\textit{Flow})$  := ∅
4   local upResults :  $\mathcal{P}(\textit{Flow})$  := ∅
5   foreach e with e ∈ node.incoming
6     if e ≠ edge
7       let
8         newFlow = CLONEFLOW(flow)
9         joinResult = FLOWUP(e, newFlow)
10      in
11        upResults := ELEMENTWISEUNION(upResults, joinResult)
12  seq
13  foreach downFlow with downFlow ∈ CLONEFLOWS(upResults)
14    let
15      joinResult = FLOWDOWN(outgoing(node,1), downFlow)
16    in
17      add joinResult to r
18  seq
19  result := r

```

In obigem Beispiel aus Abb. 3.14 ergibt sich für die Flussberechnung ausgehend von Aktion E zunächst folgender Fluss:

$$(\emptyset, \emptyset, \{\text{o3}\}, \emptyset) \rightarrow (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \{\text{o3}, \text{undef}\})$$

Nach der Berechnung der Voraussetzungen (Zeilen 5–11) sind die Aktionen B, C, D und E und die rechte ausgehende Kante des *ForkNodes* F1 in der Flussvorbedingung enthalten. Außerdem

sind die Datenflüsse mit möglichen Quellen gefüllt. Der Fluss sieht dann folgendermaßen aus:

$$(\{B, C, D, E\}, \{F1Right\}, \emptyset, \emptyset) \rightarrow (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \{(o3, undef), (F1Right, undef), (o2, undef)\})$$

Es könnte auch sein, dass statt genau einem Fluss eine Menge von Flüssen bestimmt wird, da bei der Berechnung der Voraussetzung *MergeNodes* im Pfad sein können.

Im nächsten Schritt wird für jedes Element dieser Zwischenergebnismenge der Fluss nach unten bestimmt (Zeilen 13–17) und man erhält für das hier skizzierte Beispiel folgende Ergebnismenge:

$$\left\{ \begin{aligned} &(\{B, C, D, E\}, \{F1Right\}, \emptyset, \emptyset) \rightarrow (\{G\}, \emptyset, \emptyset, \emptyset, \emptyset, \{(o2, i2)\}), \\ &(\{B, C, D, E\}, \{F1Right\}, \emptyset, \emptyset) \rightarrow (\{G\}, \emptyset, \emptyset, \emptyset, \emptyset, \{(o3, i2)\}), \\ &(\{B, C, D, E\}, \{F1Right\}, \emptyset, \emptyset) \rightarrow (\{G\}, \emptyset, \emptyset, \emptyset, \emptyset, \{(F1Right, i2)\}), \\ &(\{B, C, D, E\}, \{F1Right\}, \emptyset, \emptyset) \rightarrow (\{H\}, \emptyset, \emptyset, \emptyset, \emptyset, \{(o2, i3)\}), \\ &(\{B, C, D, E\}, \{F1Right\}, \emptyset, \emptyset) \rightarrow (\{H\}, \emptyset, \emptyset, \emptyset, \emptyset, \{(o3, i3)\}), \\ &(\{B, C, D, E\}, \{F1Right\}, \emptyset, \emptyset) \rightarrow (\{H\}, \emptyset, \emptyset, \emptyset, \emptyset, \{(F1Right, i3)\}) \end{aligned} \right\}$$

Diese Flüsse entsprechen der oben verdeutlichten Semantik von Aktivitätsdiagrammen: Die Aktionen B, C, D und E und der *ForkNode* F1 müssen jeweils ein Token anbieten, damit entweder die Aktion G oder H starten kann. Der *InputPin* i2 oder i3 wird dann von einem der drei möglichen Datenquellen bedient. Wichtig hierbei ist, dass für jeden möglichen Datenfluss ein eigener Fluss erzeugt wird und sich die Flüsse demnach gegenseitig ausschließen.

ActionFlowDown. Das Makro **ACTIONFLOWDOWN** beschreibt die für die Flussberechnung notwendigen Schritte, falls das Ziel der aktuellen Kante eine Aktion ist. Dies ist nur dann der Fall, wenn die Kante Teil eines Kontrollflusses ist, da Objektflüsse in einem *InputPin* und nicht direkt in einer Aktion enden.

Wenn die betrachtete Aktion gemäß der Abfrage in Zeile 4 keine *InputPins* hat, kann folglich die Rekursion beendet werden, da auf Grund der Normalisierung sichergestellt ist, dass es nur eine eingehende Kontrollflusskante gibt. In diesem Fall wird die Aktion der Funktion *actionsStart* hinzugefügt und die ergänzte Datenstruktur zurückgeliefert.

Sollte die Aktion allerdings *InputPins* besitzen, bedeutet dies ein implizites Join-Verhalten. Daher werden wiederum alle eingehenden Kanten entgegen ihrer Pfeilrichtung rekursiv verfolgt, um die weiteren notwendigen Voraussetzungen für den Start der aktuellen Aktion zu erhalten.

Bevor dies erfolgt, wird der Funktion *actionsStart* des aktuellen Flusses die aktuelle Aktion hinzugefügt. Anschließend wird pro *InputPin* jede eingehende Kante rückwärts verfolgt und mit den bisherigen Ergebnissen elementweise vereinigt.

```

1 ACTIONFLOWDOWN : ActivityNode × Flow → P(Flow)
2 ACTIONFLOWDOWN(node, flow) ≡
3   local r : P(Flow) := ∅
4   if node.input = ∅ then
5     add node to flow.actionsStart
6     seq
7     add flow to r

```

```

8  else
9  let
10  newFlow = CLONEFLOW(flow)
11  in
12  add node to newFlow.actionsStart
13  seq
14  r := {newFlow}
15  foreach p with p ∈ node.input
16  foreach e with e ∈ p.incoming
17  let
18  newFlow = CLONEFLOW(flow)
19  in
20  if newFlow.dataFlows = ∅ then
21  add (undef, p) to newFlow.dataFlows
22  else
23  first(asList(newFlow.dataFlows)).Second := p
24  seq
25  let
26  upResult = FLOWUP(e, newFlow)
27  in
28  r := ELEMENTWISEUNION(r, upResult)
29  seq
30  result := r

```

ObjectNodeFlowDown. Das Makro **OBJECTNODEFLOWDOWN** unterscheidet zunächst die Art des *ObjectNodes*. Falls es *kein InputPin* ist, wird lediglich die Datenstruktur angepasst und zurückgeliefert (Zeilen 45–48). Ähnlich wird verfahren, falls die Aktion, zu der der *InputPin* gehört, keine anderen eingehenden Kanten oder *Pins* besitzt (Zeilen 40–43).

Anderenfalls muss wieder ein impliziter Join durchgeführt werden. Dieses Mal werden zunächst die restlichen eingehenden Objektflüsse verfolgt (Zeilen 17–28) und anschließend, falls vorhanden, der einzige Kontrollfluss berechnet (Zeilen 29–38).

```

1  OBJECTNODEFLOWDOWN : ActivityNode × Flow → P(Flow)
2  OBJECTNODEFLOWDOWN(node, flow) ≡
3  local r : P(Flow) := ∅
4  if node ∈ Pin then
5  let
6  owner = node.owner
7  in
8  if multipleInFlows(owner) then
9  let
10  newFlow = CLONEFLOW(flow)
11  in
12  add owner to newFlow.actionsStart
13  first(asList(newFlow.dataFlows)).Second := node
14  seq

```

```

15     r := {newFlow}
16   seq
17   foreach p with p ∈ owner.input
18     if p ≠ node then
19       foreach e with e ∈ p.incoming
20         let
21           newFlow = CLONEFLOW(flow)
22         in
23           first(asList(newFlow.dataFlows)).Second := p
24         seq
25         let
26           upResult = FLOWUP(e, newFlow)
27         in
28           r := ELEMENTWISEUNION(r, upResult)
29     if owner.incoming ≠ ∅ then
30       let
31         newFlow = CLONEFLOW(flow)
32       in
33         first(asList(newFlow.dataFlows)).Second := p
34       seq
35       let
36         upResult = FLOWUP(incoming(owner, 1), newFlow)
37       in
38         r := ELEMENTWISEUNION(r, upResult)
39     else
40       add owner to flow.actionsStart
41       first(asList(flow.dataFlows)).Second := node
42       seq
43       add flow to r
44   else
45     add node to flow.objectNodesBuffering
46     first(asList(flow.dataFlows)).Second := node
47     seq
48     add flow to r
49   seq
50   result := r

```

FlowUp. Nachdem nun das Makro **FLOWDOWN** mit allen „Untermakros“, die spezielles Verhalten für unterschiedliche Aktivitätsknoten spezifiziert haben, vollständig beschrieben ist, fehlt nun zur Berechnung der einfachen Flüsse noch das Makro **FLOWUP**.

Es ist prinzipiell sehr ähnlich zu **FLOWDOWN**, allerdings werden alle Kanten darin eben nicht vorwärts sondern rückwärts verfolgt und insgesamt sind die Teilregeln einfacher, da komplexere Zusammenhänge bereits im Makro **FLOWDOWN** berücksichtigt wurden.

Der Anfang des Makros ist (davon abgesehen, dass nicht *edge.target*, sondern *edge.source* als aktueller Knoten betrachtet wird) identisch mit **FLOWDOWN** bis Zeile 15. Auch bei **FLOWUP** wird anschließend wieder eine Fallunterscheidung bezüglich der verschiedenen Knotenty-

pen gemacht, allerdings sind die einzelnen Regeln weniger kompliziert und daher meist direkt in das Makro integriert.

```

1 FLOWUP : ActivityEdge × Flow →  $\mathcal{P}(\textit{Flow})$ 
2 FLOWUP(edge, flow) ≡
3   local r :  $\mathcal{P}(\textit{Flow})$  := ∅
4   let
5     node : ActivityNode = edge.source
6     diffRegions :  $\mathcal{P}(\textit{InterruptibleActivityRegion})$  =
7       edge.target.inInterruptibleRegion \ edge.source.inInterruptibleRegion
8   in
9     add edge.guard to flow.guards
10    forall region with region ∈ edge.interrupts
11      add region to flow.regionsInterrupted
12    add diffRegions to flow.regionsEntered
13    if node ∈ InitialNode then
14      add node to flow.actionsOffering
15      add flow to r
16    if node ∈ Action then
17      if multipleOutFlows(node) then
18        add (node, edge) to flow.forksOffering
19      else
20        add node to flow.actionsOffering
21      add flow to r
22    if node ∈ CentralBufferNode ∪ ActivityParameterNode then
23      add node to flow.objectNodesOffering
24      first(asList(flow.dataFlows)).First := node
25      add flow to r
26    if node ∈ OutputPin then
27      r := OUTPUTPINFLOWUP(node, edge, flow)
28    if node ∈ DecisionNode then
29      r := FLOWUP(incoming(node,1), flow)
30    if node ∈ MergeNode then
31      r := MERGEFLOWUP(node, flow)
32    if node ∈ ForkNode then
33      r := FORKFLOWUP(node, edge, flow)
34    if node ∈ JoinNode then
35      r := JOINFLOWUP(node, flow)
36    seq
37    result := r

```

Die Behandlung des *InitialNodes* ist wiederum einfach (Zeile 13–15). Auch ein *Central-Buffer-* oder *ActivityParameterNode* kann einfach durch Ergänzung der Flussdatenstruktur abgehandelt werden (Zeilen 22–25).

Etwas komplizierter ist die Behandlung einer Aktion. Falls diese mehrere ausgehende Flüsse hat (Definition der Funktion *multipleOutFlows* siehe Anhang A.4), liegt ein impliziter *ForkNode* vor und es muss, statt auf die Aktion selbst, auf den entsprechenden *ForkNode-*

Puffer zurückgegriffen werden (Zeilen 16–21). Aktion B in Abb. 3.8 ist ein Beispiel dafür.

Die Behandlung eines *OutputPins* ist ähnlich zur Behandlung einer Aktion. Allerdings muss hierbei noch die Funktion *dataFlows* entsprechend angepasst werden. Der *MergeNode* ist in der Rückwärtsbetrachtung ähnlich zum *DecisionNode* vorwärts. Die formalen Spezifikationen der beiden Makros **OUTPUTPINFLOWUP** und **MERGEFLOWUP** finden sich in Anhang A.2.

ForkFlowUp. Wird bei einer Rückwärtsberechnung auf einen *ForkNode* gestoßen, wird die Rekursion abgebrochen und der entsprechende *ForkNode*-Puffer als Flussvorbereitung in die Funktion *forksOffering* eingetragen. Die eingehenden Kanten des *ForkNodes* müssen nicht weiter verfolgt werden, da die Flüsse, die zu diesem *ForkNode* führen und die Tokens entsprechend puffern, an anderer Stelle ausgehend von den Quellen dieser Flüsse bestimmt werden.

Allerdings wird noch beachtet, dass, falls es sich bei der aktuellen Kante um einen Objektfluss handelt, die Funktion *dataFlows* mit dem am Anfang der Flussberechnung erstellten *ObjectNode* für die entsprechende Kante des *ForkNodes* als Datenquelle ergänzt wird.

Damit diese Kante passiert werden kann, muss der evtl. annotierte *Guard* wahr sein. Dies wird berücksichtigt, indem der *Guard* dieser Kante bereits im Makro **FLOWUP** dem Fluss hinzugefügt wurde.

Der *Guard* an ausgehenden Kanten von *ForkNodes* wird also zweimal evaluiert: Einmal, bevor ein Token in den Puffer geschrieben wird und ein weiteres Mal, wenn das Token aus dem Puffer benutzt werden soll. Dies entspricht der gewünschten Semantik wie in Abschnitt 2.3.2 beschrieben, die in diesem Punkt von der Semantik von Sarstedt abweicht.

FORKFLOWUP : $ActivityNode \times ActivityEdge \times Flow \rightarrow \mathcal{P}(Flow)$

FORKFLOWUP(*node*, *edge*, *flow*) \equiv

local *r* : $\mathcal{P}(Flow) := \emptyset$

if *edge* \in *ObjectFlow* then

if *flow.dataFlows* = \emptyset then

add (*forkEdgeToObjectNode*(*node*, *edge*), undef) to *flow.dataFlows*

else

first(asList(flow.dataFlows)).First := *forkEdgeToObjectNode*(*node*, *edge*)

add (*node*, *edge*) to *flow.forksOffering*

result := *r*

JoinFlowUp. Auch bei *JoinNodes* wird jede eingehende Kante rekursiv verfolgt. Allerdings werden die Ergebnisse nicht wie bei *MergeNodes* einfach vereinigt, was in diesem Zusammenhang einer Oder-Verknüpfung entspricht, sondern elementweise verknüpft. Dies bewirkt eine Zusammenfassung aller berechneten (Teil-)Flüsse und somit einer Und-Verknüpfung.

JOINFLOWUP : $ActivityNode \times Flow \rightarrow \mathcal{P}(Flow)$

JOINFLOWUP(*node*, *flow*) \equiv

local *r* : $\mathcal{P}(Flow) := \emptyset$

foreach *e* with *e* \in *node.incoming*

let

newFlow = **CLONEFLOW**(*flow*)

```

    upResult = FLOWUP(e, newFlow)
  in
    r := ELEMENTWISEUNION(r, upResult)
seq
result := r

```

Nach vollständiger Abarbeitung der Regel **COMPSIMPLEFLOWS** sind alle einfachen Flüsse berechnet. Der nächste Abschnitt widmet sich deren Kombination zu kombinierten Flüssen.

3.2.4.4 Berechnung der kombinierten Flüsse

Im Gegensatz zu einfachen Flüssen, können kombinierte Flüsse auch über die Grenze der *ForkNode*-Puffer hinweg fließen. Im Beispiel aus Abb. 3.8 ist der Fluss vom *InitialNode* direkt zu den Aktionen A und B bereits ein kombinierter Fluss. Deren Bestimmung ist wichtig, da laut UML nicht grundsätzlich Daten an *ForkNodes* gepuffert werden sollen:

„The outgoing edges that did not accept the token due to failure of their targets to accept it, keep their copy in an implicit FIFO queue until it can be accepted by the target.“ [UML, Seite 376]

Insbesondere der erste Teil des Satzes begründet die Notwendigkeit von kombinierten Flüssen. Dort wird betont, dass nur Flüsse, deren Ziele das Token nicht akzeptieren können, gepuffert werden. Aktionen können Tokens beispielsweise nicht akzeptieren, falls nicht alle eingehenden Kanten mit Tokens versorgt sind. Für den eben erwähnten Fall des *InitialNodes* und der Aktionen A und B gilt allerdings, dass die Aktionen A und B immer bereit sind und daher das Puffern nicht notwendig ist.

Um diese Fälle ebenfalls zu erfassen und damit das Prinzip der „*traverse-to-completion*“-Regel (siehe Abschnitt 2.3.2) in den entsprechenden Fällen für *ForkNodes* zu erfüllen, werden in diesem Schritt die möglichen Kombinationen, die sich aus den einfachen Flüssen ergeben, berechnet. In einem weiteren Schritt werden dann die einfachen Flüsse überprüft, ob sie überhaupt auftreten können und gegebenenfalls wieder entfernt.

Eine andere Möglichkeit wäre, nur die notwendigen Pufferungen zu berechnen (die in [Ras09] vorgestellte vereinfachte Variante des Algorithmus arbeitet tatsächlich nach diesem Prinzip). Der hier beschriebene Weg wurde jedoch für eine einheitliche Berechnung der kombinierten Flüsse gewählt. Außerdem harmonisiert die andere Variante des Algorithmus nicht mit Objektflüssen, die sehr häufig eine implizite Pufferung erfordern.

Wiederum an Hand des Beispiels aus Abb. 3.8 wird kurz das Prinzip der Berechnung erläutert. Für die Aktion A werden unter anderem folgende einfache Flüsse ermittelt:

$$\left\{ \begin{aligned}
 &(\{A\}, \emptyset, \emptyset, \{x > 1, y < 0\}) \rightarrow (\emptyset, \{\mathbf{F2Left}, \mathbf{F2Right}\}, \emptyset, \emptyset, \emptyset, \{(o1, \mathbf{F2Left}), (o1, \mathbf{F2Right})\}), \\
 &(\{A\}, \emptyset, \emptyset, \{x > 1, \neg(y < 0)\}) \rightarrow (\emptyset, \{\mathbf{F2Left}\}, \emptyset, \emptyset, \{(o1, \mathbf{F2Left})\}), \\
 &\dots \}
 \end{aligned} \right.$$

Das Hauptaugenmerk liegt dabei auf der zweiten Komponente des rechten Tupels (fettgedruckt). Dies beschreibt die Menge der Funktion *forksBuffering* und enthält die *ForkNodes*, die bei diesem Fluss ein Token puffern müssen. Andererseits wurde ausgehend vom *ForkNode*

F2 folgende Menge ermittelt:

$$\left\{ \begin{aligned} &(\emptyset, \{\mathbf{F2Right}\}, \emptyset, \{y < 0\}) \rightarrow (\{D\}, \emptyset, \emptyset, \emptyset, \emptyset, \{(F2Right, i2)\}), \\ &(\emptyset, \{\mathbf{F2Right}\}, \emptyset, \{y < 0\}) \rightarrow (\{E\}, \emptyset, \emptyset, \emptyset, \emptyset, \{(F2Right, i3)\}), \\ &(\emptyset, \{\mathbf{F2Left}\}, \emptyset, \{x > 1\}) \rightarrow (\{C\}, \emptyset, \emptyset, \emptyset, \emptyset, \{(F2Left, i1)\}) \end{aligned} \right\}$$

Wie man hier erkennen kann, sind Elemente der Menge *forksOffering* auch in der Menge *forksBuffering* enthalten (z. B. **F2Right**). Genau darauf setzt die Kombination der Flüsse auf.

Es wird ausgehend von einem Element der Menge *forksOffering* in den restlichen Flüssen nach einem Fluss gesucht, der dieses Element in der Menge *forksBuffering* enthält. Diese beiden Flüssen können dann (vorausgesetzt, die Datenflüsse passen ebenfalls zusammen) zu einem neuen Fluss kombiniert werden. Gleiche Elemente in beiden Mengen werden dabei „weggekürzt“. Kombiniert man die fett gesetzten Elemente, erhält man folgende kombinierte Flüsse:

$$\left\{ \begin{aligned} &(\{A\}, \emptyset, \emptyset, \{x > 1, y < 0\}) \rightarrow (\{D\}, \{F2Left\}, \emptyset, \emptyset, \emptyset, \{(o1, F2Left), (o1, i2)\}), \\ &(\{A\}, \emptyset, \emptyset, \{x > 1, y < 0\}) \rightarrow (\{E\}, \{F2Left\}, \emptyset, \emptyset, \emptyset, \{(o1, F2Left), (o1, i3)\}), \\ &(\{A\}, \emptyset, \emptyset, \{x > 1, y < 0\}) \rightarrow (\{C\}, \{F2Right\}, \emptyset, \emptyset, \emptyset, \{(o1, i1), (o1, F2Right)\}) \end{aligned} \right\}$$

Diese Flüsse könnten dann in einer weiteren Iteration wieder mit entsprechenden Flüssen kombiniert werden, bis schließlich alle *ForkNode*-Puffer Elemente (F2Left, F2Right) eliminiert sind.

CalculateForksCombinedFlows. Eine Formalisierung der Kombination der einfachen Flüsse findet sich im Makro **MIXFORKS**. Dieses Makro wird so oft aufgerufen, bis die Berechnung einen Fixpunkt erreicht hat und keine neuen Flüsse mehr hinzukommen. Dies ist notwendig, da es, wie dargestellt, auch Kombinationen von Kombinationen geben kann. Dies erfüllt das ASM-Konstrukt **iterate** im Makro **CALCULATEFORKSCOMBINEDFLOWS**, welches **MIXFORKS** immer wieder mit den bisher erstellten Kombinationen aufruft.

Der zweite Parameter des Makros **MIXFORKS** enthält die Menge der einfachen Flüsse, die zur Bildung einer Kombination herangezogen werden können. Das Ergebnis des Makros **CALCULATEFORKSCOMBINEDFLOWS** ist dann die Vereinigung der einfachen mit den kombinierten Flüssen.

```

CALCULATEFORKSCOMBINEDFLOWS :  $\mathcal{P}(\text{Flow}) \rightarrow \mathcal{P}(\text{Flow})$ 
CALCULATEFORKSCOMBINEDFLOWS(flows)  $\equiv$ 
  local forkCombinations :  $\mathcal{P}(\text{Flow}) := \mathbf{CLONEFLOWS}(\text{flows})$ 
  iterate
    forkCombinations := forkCombinations  $\cup$  MIXFORKS(forkCombinations, flows)
  seq
  result := forkCombinations  $\cup$  flows

```

MixForks. Das Makro **MIXFORKS** ist symmetrisch aufgebaut. Die ASM-Regeln von Zeile 5 bis 15 wiederholen sich ab Zeile 16 mit dem Unterschied, dass einmal Kombinationen von *forksBuffering* auf *forksOffering* gesucht werden und im zweiten Teil genau umgekehrt.

Die Suche korrespondierender Elemente, wie sie im vorletzten Absatz beschrieben wurde, findet in den Zeilen 5 bis 8 respektive 16 bis 19 statt. Bei Übereinstimmung der Elemente, werden die Flüsse kombiniert und im nachhinein überprüft, ob die vereinigten Datenflüsse noch konsistent sind (**CHECKDATAFLOWS**).

Inkonsistente Datenflüsse können auftreten, falls zwei Datenquellen auf die gleiche Senke zeigen würden, diese aber nur ein Token aufnehmen kann. Falls die Datenflüsse konsistent sind, werden alle Elemente des Flusses reduziert („gekürzt“) und der Ergebnismenge hinzugefügt.

```

1 MIXFORKS :  $\mathcal{P}(\text{Flow}) \times \mathcal{P}(\text{Flow}) \rightarrow \mathcal{P}(\text{Flow})$ 
2 MIXFORKS(flows, poolFlows)  $\equiv$ 
3   local  $r : \mathcal{P}(\text{Flow}) = \emptyset$ 
4   forall f with  $f \in \text{flows}$ 
5     forall n, e with  $(n, e) \in f.\text{forksOffering}$ 
6       forall pool with  $\text{pool} \in \text{poolFlows}$ 
7         forall n2, e2 with  $(n_2, e_2) \in \text{pool}.\text{forksBuffering}$ 
8           if  $n = n_2 \wedge e = e_2$  then
9             let
10              newFlow = COMBINEFLOWS(f, pool)
11            in
12              if CHECKDATAFLOWS(newFlow) then
13                REDUCEELEMENTS(newFlow)
14              seq
15                add newFlow to r
16          forall n, e with  $(n, e) \in f.\text{forksBuffering}$ 
17            forall pool with  $\text{pool} \in \text{poolFlows}$ 
18              forall n2, e2 with  $(n_2, e_2) \in \text{pool}.\text{forksOffering}$ 
19                if  $n = n_2 \wedge e = e_2$  then
20                  let
21                    newFlow = COMBINEFLOWS(f, pool)
22                  in
23                    if CHECKDATAFLOWS(newFlow) then
24                      REDUCEELEMENTS(newFlow)
25                    seq
26                      add newFlow to r
27          seq
28          result := r

```

CalculateMultiDataFlows. Eine zusätzliche Erweiterung der einfachen Flüsse zu kombinierten Flüssen betreffen nicht *ForkNodes* sondern *InputPins*. Diese können laut UML-Spezifikation [UML, Seite 255] lower- und upper-Werte haben, die angeben, wie viele Tokens minimal anliegen müssen und maximal anliegen dürfen.

Der Standard liegt für beide Werte bei eins. Der *InputPin* i3 der Aktion E aus Abb. 3.8 hat einen upper-Wert von zwei. Daher könnten auch zwei Flüsse gleichzeitig von diesem Knoten verarbeitet werden.

Im genannten Beispiel könnten also auch die Aktionen A und B gleichzeitig Tokens anbieten und würden damit (neben den Aktionen C und F) auch Aktion E starten. Beide

Datentokens von *o1* und *o2* würden dabei nach *i3* gehen.

Diese Kombinationen von gleichzeitigen Datenflüssen, die auch für *CentralBufferNodes* und *ActivityParameterNodes* möglich sind, müssen ebenfalls berechnet werden. Das Makro **CALCULATEMULTIDATAFLOWS** (siehe Anhang A.2) reichert die Gesamtmenge der Flüsse um genau diese Kombinationen an.

Dabei wird auf eine Funktion *combinations*(*n*, *k*) zurückgegriffen, die eine Liste von Listen der möglichen Kombinationen für *k* aus *n* ohne Wiederholung zurückliefert. Wird die Funktion z. B. mit den Werten (4, 3) aufgerufen, liefert sie folgende Liste zurück: [[1,2,3], [1,2,4], [1,3,4], [2,3,4]]. Dieser Algorithmus wird in dieser Arbeit nicht weiter ausgeführt, da es sich dabei um einen Standardalgorithmus handelt.

Im Vorfeld wurden bereits die möglichen Datenquellen pro *ObjectNode* berechnet und in der Funktion *possibleDataSources* gespeichert. Diese Menge der Datenquellen pro Knoten wird in eine Liste umgewandelt.

Jede Nummer der Kombinationslisten zeigt nun auf den entsprechenden Eintrag in der Liste der möglichen Datenquellen. Die durch eine Kombinationsliste definierten Flüsse werden schließlich kombiniert und der Gesamtmenge hinzugefügt.

Für das Beispiel aus Abb. 3.8 würde der Fluss hinzugefügt werden, der sich ergibt, wenn Aktion B *und gleichzeitig* der *ForkNode* F2 an der rechten Kante jeweils ein Objekttoken anbieten und diese dann an den Knoten *i3* transferiert werden.

Natürlich ergeben sich auch für diese Flüsse wieder Kombinationen mit *ForkNodes*, so dass diese Berechnung vor der Kombination der *ForkNodes* durchgeführt wird.

Reduktion der Flüsse. Wie oben angekündigt, werden nach diesen Berechnungen der kombinierten Flüsse nicht notwendige Flüsse wieder entfernt. Dabei können zwei Gruppen unterschieden werden:

1. Flüsse, die *ForkNode*-Puffer befüllen, obwohl dies nicht nötig ist.
2. Flüsse, die keinen Effekt haben.

Flüsse der ersten Gruppe werden vom Makro **REMOVENOTNECESSARYFORKS** entfernt. *ForkNode*-Pufferung ist immer genau dann unnötig, wenn es sich bei den ausgehenden Kanten um einen Kontrollfluss handelt und dieser von keinem *JoinNode* oder impliziten Join gefolgt wird. Bei Objektflüssen kann es immer vorkommen, dass die nachfolgenden *InputPins* nicht genügend Platz für ankommende Objekttoken haben. Daher werden Objektflüsse nicht entfernt.

Flüsse ohne Effekt entstehen beispielsweise bei der Generierung der verschiedenen Kombinationen für *ForkNodes* mit *⟨Guards⟩* an ausgehenden Kanten (**CREATEFLOWCOMBINATIONSFORFORKNODE**). Die Abfragen, die notwendig wären, um diese Fälle zu vermeiden, sind komplizierter als diese berechnen zu lassen und im Nachhinein zu entfernen.

Die Funktion *removeFlowsWithNoEffect* liefert eine um diese Fälle bereinigte Menge an Flüssen zurück. Die Spezifikationen des Makros und der Funktion finden sich in Anhang A.2 und werden hier nicht erläutert.

CompFlows. Abschließend zur Flussberechnung wird zusammenfassend noch das Makro **COMPFLOWS** vorgestellt, welches den Ablauf der einzelnen Schritte festlegt. Nach der Erzeugung der für die Pufferung notwendigen zusätzlichen *ObjectNodes* und den vorbereitenden

Berechnungen der möglichen Datenquellen pro *ObjectNode*, werden zunächst die einfachen Flüsse erzeugt (**COMPSIMPLEFLOWS**).

Daran anschließend werden die Kombinationen der Datenflüsse erstellt, bevor auf allen Flüssen eine Fixpunktberechnung der *ForkNode*-Kombinationen erfolgt. Schließlich werden die nicht notwendigen *ForkNode*-Pufferungen und Flüsse ohne Effekt entfernt. Zur leichteren Handhabung in den nächsten Schritten werden abschließend die Datenflüsse reduziert bzw. transitive Abhängigkeiten aufgelöst.

```

COMPFLOWS : void →  $\mathcal{P}(\text{Flow})$ 
COMPFLOWS ≡
  local flows :  $\mathcal{P}(\text{Flow}) = \emptyset$ 
  CREATEFORKEDGEBUFFERS
  CALCULATEPOSSIBLEDATASOURCES
  seq
  flows := COMPSIMPLEFLOWS
  seq
  flows := CALCULATEMULTIDATAFLOWS(flows)
  seq
  flows := CALCULATEFORKSCOMBINEDFLOWS(flows)
  seq
  flows := REMOVENOTNECESSARYFORKS(flows)
  seq
  flows := removeFlowsWithNoEffect(flows)
  seq
  forall f with f ∈ flows
    REDUCEDATAFLOWS(f)
  seq
  result := flows

```

Aufbauend auf diesen Flüssen können dann im nächsten Schritt die Zustandsübergänge für das Model Checking erstellt werden.

3.2.5 Erzeugung der Zustandsübergänge

Zustandsübergänge in einem Aktivitätsdiagramm ergeben sich aus Modusänderungen der enthaltenen Aktionen. Wie in Abschnitt 3.2.3 beschrieben, können Aktionen entweder inaktiv oder aktiv sein oder Tokens an den ausgehenden Kanten anbieten.

Ein Zustandsübergang findet statt, wenn eine oder mehrere Aktionen ihren Modus ändern, da ein Systemzustand unter anderem die aktuellen Modi aller Aktionen beinhaltet (siehe Abschnitt 3.2.3). Dies kann auftreten, wenn z. B. eine *AcceptEventAction* ein Signal empfangen hat und daher beendet wird oder weil eine Aktion mit dem in ihr spezifizierten Verhalten fertig ist und daher Tokens für die nachfolgenden Aktionen anbietet.

Prinzipiell sind zwei wesentliche Modusübergangsarten zu unterscheiden:

- **running** → **offeringToken**

Eine Aktion wird beendet, wechselt in den Modus **offeringToken** und bietet somit ein Token für nachfolgende Knoten an. Je nachdem, welche Art von Aktion beendet wird, können noch weitere Tätigkeiten notwendig sein, wie beispielsweise Signale erzeugen

und in Signalpuffer schreiben oder einem Signalpuffer Signale entnehmen.

Im Rahmen des Model Checkings können Aktionen jederzeit spontan und weitgehend unabhängig von den restlichen Aktionen beendet werden. Nur für *AcceptEventActions* muss beachtet werden, dass das entsprechende Signal empfangen wurde.

- **offeringToken** → **running**

Bei dieser Art von Zustandsübergang wird auf die im letzten Schritt berechneten Flüsse zurückgegriffen. An Hand der Flüsse wird in Abhängigkeit der Modi anderer Knoten bestimmt, welche nachfolgenden Knoten in den Modus **running** versetzt werden können. Dieser Modusübergang beeinflusst also vor allem andere Knoten als den aktuell betrachteten. Der aktuelle Knoten wird meist wieder in den Modus **inactive** versetzt. Insbesondere finden bei diesem Übergang typischerweise keine weiteren Systemzustandsänderungen wie Signalerzeugung, Datenmanipulation oder ähnliches statt.

Entsprechend dieser Unterscheidung werden die Zustandsübergänge, die sich durch Tokenflüsse ergeben (**offeringToken** → **running**) und die Zustandsübergänge, die sich durch Ausführung einer Aktion ergeben (**running** → **offeringToken**), getrennt voneinander behandelt. Den Ablauf bei der Erzeugung der Zustandsübergänge verdeutlicht Abb. 3.15. Nach der Definition der für die Umsetzung notwendigen Variablen (Abschnitt 3.2.5.1), wird zunächst die Zustandsübergangserzeugung für Flüsse vorgestellt. Anschließend wird die Umsetzung der Übergänge innerhalb von Aktionen in Abschnitt 3.2.5.3 definiert.

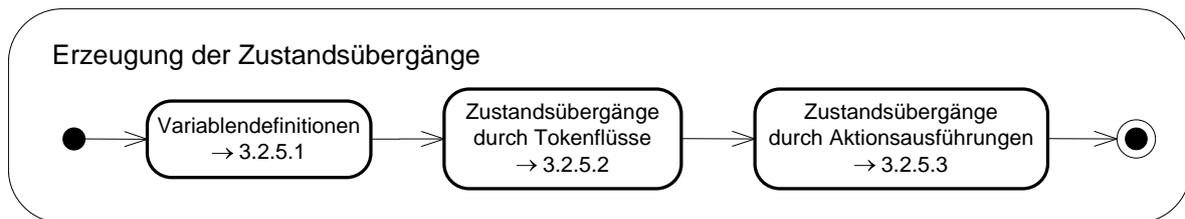


Abbildung 3.15: Einzelne Schritte der Zustandsübergangserzeugung.

Vor der formalen Definition des Algorithmus im übernächsten Abschnitt, wird im Folgenden die prinzipielle Vorgehensweise bei der Umsetzung der Flüsse in Zustandsübergänge beschrieben.

Am Ende des Abschnitts 3.2.4.1 wurde folgende kompakte Darstellung der Flüsse eingeführt:

$$\begin{aligned}
 & (\text{actionsOffering}, \text{forksOffering}, \text{objectNodesOffering}, \text{guards}) \rightarrow \\
 & \quad (\text{actionsStart}, \text{forksBuffering}, \text{objectNodesBuffering}, \\
 & \quad \text{regionsEntered}, \text{regionsInterrupted}, \text{dataFlows})
 \end{aligned}$$

Der Teil links des Pfeils beschreibt dabei die Flussvorbedingung und der Teil rechts davon den Flusseffekt. Die Flussdatenstruktur lässt sich somit auch folgendermaßen lesen: „Wenn die Flussvorbedingungen wahr sind, dann treten die im Flusseffekt zusammengefassten Änderungen ein.“

Diese Darstellung ist der Definition von *Guarded Commands* sehr nahe. Guarded Commands sind elementarer Bestandteil der 1975 von Dijkstra in [Dij75] eingeführten *Guarded*

Command Language zur Beschreibung der Semantik des Prädikatentransformers *wp* (weakest pre-condition)[Dij76]. Die Syntax der Guarded Commands ist folgendermaßen definiert:

$$\begin{aligned} \langle \text{guarded command} \rangle &::= \langle \text{guard} \rangle \rightarrow \langle \text{guarded list} \rangle \\ \langle \text{guard} \rangle &::= \langle \text{boolean expression} \rangle \\ \langle \text{guarded list} \rangle &::= \langle \text{statement} \rangle \{ ; \langle \text{statement} \rangle \} \end{aligned}$$

Guarded Commands können auch zur Beschreibung von Zustandsübergangssystemen eingesetzt werden. Dabei beschreibt der *guard* den aktuellen Zustand und die *guarded list* operationell den Folgezustand. Eine Flussvorbedingung (umgewandelt in eine logische Formel), die den aktuellen Zustand beschreibt, entspricht demnach einem *guard*. Der Flusseffekt (bzw. Zustandsübergang) kann als *guarded list* dargestellt werden. Tatsächlich sind Guarded Commands auch die Basis der Eingabesprachen von verschiedenen Model Checkern wie SPIN [Hol97], NuSMV2 [CCG⁺02], Mur φ [DDHY92] und PRISM [HKNP06]. Daher ist eine Umsetzung der Flüsse in diese Sprachen mit wenig Aufwand möglich.

Die prinzipielle Umwandlung der berechneten Flüsse in Guarded Commands zeigt folgende (vereinfachte) Formel:

$$\begin{aligned} \forall f \in \text{flows} : \\ \bigwedge_{a \in f.\text{actionsStart}} a = \text{offeringToken} \wedge \bigwedge_{fo \in f.\text{forksOffering}} fo = \text{offeringToken} \wedge \\ \bigwedge_{ono \in f.\text{objectNodesOffering}} ono = \text{offeringToken} \wedge \bigwedge_{g \in f.\text{guards}} g \longrightarrow \\ \forall r \in (f.\text{actionsStart} \setminus \text{nodesInterrupted}) \cup \text{nodesEntered} : r := \text{running}; \\ \forall i \in (f.\text{actionsOffering} \cup \text{nodesInterrupted}) : i := \text{inactive}; \\ \forall fb \in (f.\text{forksBuffering} \setminus \text{nodesInterrupted}) : fb := \text{offeringToken}; \\ \forall fi \in (f.\text{forksOffering} \cap \text{nodesEntered}) : fi := \text{inactive}; \\ \forall (n_0, n_1) \in f.\text{dataFlows} : n_1 := n_0 \oplus \text{first}(n_0); \text{remove}(n_0, 1) \end{aligned}$$

Für jeden Fluss wird für jedes Element der Vorbedingungsmengen eine Abfrage erstellt, die die Knoten auf angebotene Tokens überprüft. Diese Abfragen werden über eine Konjunktion miteinander und mit den *Guards* verknüpft. Diese entstandene logische Aussage bildet die Voraussetzung für das Schalten des Flusses.

Für die Effekte wird davon ausgegangen, dass sich in den Mengen „nodesInterrupted“ und „nodesEntered“ diejenigen Knoten befinden, deren umliegende Unterbrechungsbereiche bei diesem Fluss betreten bzw. abgebrochen werden. Damit ergeben sich die Mengen der Knoten, deren Modus auf *running*, *inactive* oder *offeringToken* gesetzt werden muss. Schließlich werden noch die in der Menge *dataFlows* beschriebenen Datenflüsse ausgeführt.

Die obige Formel stellt nur die prinzipielle Vorgehensweise dar. Die tatsächliche Umsetzung in ASM-Makros berücksichtigt weitere Details, die im Folgenden ausführlich beschrieben werden. In diesen Makros werden an verschiedenen Stellen ASM-Regeln aufgerufen, die in dieser Arbeit nur textuell spezifiziert sind, da sie die Übersetzung in die konkrete Eingabesprache eines Model Checkers erledigen und die Wahl des Werkzeugs flexibel gehalten werden soll.

In Abschnitt 4.1 werden die entsprechenden abstrakten Regeln näher erläutert und eine beispielhafte Umsetzung aufgezeigt. Diese hier ausgenutzte Möglichkeit der ASMs, auf beliebigen Ebenen zu abstrahieren, ist eine der Stärken und der Hauptgrund für einen flexiblen Einsatz des Formalismus.

3.2.5.1 Variablendefinitionen

Die Instanzen des Metamodells, auf die während der Berechnung der Flüsse zugegriffen wird, müssen für eine Realisierung des Zustandsübergangssystems in einer Eingabesprache eines Model Checkers in konkrete Variablen und Werte umgesetzt werden. Dieser Abschnitt beschreibt, welche Variablen für welche Art von Elementen erzeugt werden.

Die notwendigen Schritte sind im ASM-Makro `CREATEPREAMBLE` spezifiziert. Als Ergebnis liefert dieses Makro (wie alle anderen auch) eine Liste der Domäne `ModelCheckerInput` zurück. Diese Domäne beschreibt die Eingabesprache des eingesetzten Model Checkers, die von den erwähnten abstrakten Makros erzeugt wird. Eine Liste wird deswegen benutzt, da die Reihenfolge des erzeugten Model Checker Inputs natürlich wichtig ist.

An die ursprünglich leere Ergebnisliste r werden sukzessive Befehle, Ausdrücke oder ähnliches der Eingabesprache angehängt. Im ersten Schritt werden die eingeführten Modi definiert. Ob das Makro `DEFINEMODES` diese als eine Aufzählung oder Konstante umsetzt, bleibt dem Implementierer und den Möglichkeiten des verwendeten Model Checkers überlassen.

Anschließend werden für die zugreifbaren Attribute jeder Klasse in der Eingabesprache Variablen definiert. Auch hier ist die tatsächliche Umsetzung von den Möglichkeiten des Model Checkers abhängig.

```

CREATEPREAMBLE : void → ModelCheckerInput*
CREATEPREAMBLE ≡
  local r : ModelCheckerInput* = []
  r := r ⊕ DEFINEMODES
  seq
  foreach c with c ∈ Classifier
    r := r ⊔ DEFINECLASSATTRIBUTES(c)
  seq
  r := r ⊔ DEFINENODEVARIABLES
  seq
  r := r ⊔ DEFINESIGNALS
  seq
  result := r

```

Daran schließen sich die Definitionen der Variablen für die einzelnen Knoten und der Signale an. Beide Definition sind in den folgenden Makros verfeinert.

DefineNodeVariables. Für jede Aktion und für jeden *InitialNode* werden Variablen erzeugt, die den aktuellen Modus des Knotens repräsentieren. Mit der Definition der Variablen wird auch ihr Initialzustand festgelegt. Dieser ist im Allgemeinen `inactive`, muss aber für bestimmte Fälle auf einen anderen Modus gesetzt werden.

Dies betrifft einerseits *InitialNodes* von als Startaktivitäten gekennzeichnete Aktivitäten. Deren Anfangswert wird auf `offeringToken` gesetzt (Zeilen 7ff). Andererseits werden in den Startaktivitäten befindliche Aktionen ohne eingehende Kanten, die sich nicht in einem Unterbrechungsbereich befinden (Zeilen 12–13), gestartet. Ihr Modus wird daher auf `running` gesetzt.

Für *AcceptEventActions* wird jeweils eine zusätzliche (Puffer-)Variable definiert und auf `inactive` gesetzt. Diese Variable ist notwendig, da *AcceptEventActions* Signale empfangen kön-

nen (und damit Tokens anbieten) *ohne* dass sie beendet werden. In solchen Fällen bleibt die Aktionvariable im Modus `running` und die dazugehörige Puffervariable wird in den Modus `offeringToken` gesetzt.

Für alle Objektknoten werden Listen erzeugt, die die Objekttoken aufnehmen können. Dies schließt die *ObjectNodes*, die für die Fork-Pufferung erzeugt wurden, mit ein. Eine spezielle Variable, die anzeigt, in welchem Modus sich diese Knoten befinden, ist nicht nötig. Der Knoten kann nur zwei Modi einnehmen, da er nicht in den Modus `running` gelangen kann. Die zwei verbleibenden Modi (`inactive`, `offeringToken`) können durch eine Abfrage über den Inhalt der Liste abgeprüft werden. Hat die Liste eine Länge größer Null, wird ein Token angeboten, anderenfalls nicht.

```

1 DEFINENODEVARIABLES : void → ModelCheckerInput*
2 DEFINENODEVARIABLES ≡
3   local r : ModelCheckerInput* = []
4   local m : NodeMode = inactive
5   foreach n with n ∈ ActivityNode
6     if n ∈ InitialNode then
7       if tagValue(n.activity, StartAction, initialStart) = true then
8         r := r ⊕ DEFINENODEVARIABLE(n, offeringToken)
9       else
10        r := r ⊕ DEFINENODEVARIABLE(n, inactive)
11
12    if n ∈ Action then
13      if predecessors(n) = ∅ ∧ n.inInterruptibleRegion = ∅ ∧
14        tagValue(n.activity, StartAction, initialStart) = Yes then
15        m := running
16      else
17        m := inactive
18      seq
19        r := r ⊕ DEFINENODEVARIABLE(n, m)
20      seq
21        if n ∈ AcceptEventAction then
22          r := r ⊕ DEFINEAEABUFFERVARIABLE(n, inactive)
23
24    if n ∈ ObjectNode
25      r := r ⊕ DEFINELIST(n)
26  seq
27  result := r

```

DefineSignals. Das Makro `DEFINESIGNALS` definiert neben den verwendeten Signalen für jede Aktivität einen Signalpuffer, der die gesendeten und noch nicht verarbeiteten Signale aufnimmt. Wiederum hängt die tatsächliche Umsetzung von den Möglichkeiten des Model Checkers ab (siehe Abschnitt 4.1.2).

```

DEFINESIGNALS : void → ModelCheckerInput*
DEFINESIGNALS ≡

```

```

local  $r : ModelCheckerInput^* = []$ 
foreach  $s$  with  $s \in Signal$ 
   $r := r \oplus DEFINE\_SIGNAL(s)$ 
seq
foreach  $a$  with  $a \in Activity$ 
   $r := r \oplus DEFINE\_SIGNAL\_LIST(a)$ 
seq
result  $:= r$ 

```

3.2.5.2 Zustandsübergänge durch Tokenflüsse

Dieser Abschnitt beschreibt, wie aus den im letzten Schritt erfassten Flüssen Zustandsübergänge in einer Eingabesprache für einen Model Checker generiert werden können. Die prinzipielle Vorgehensweise wurde bereits im vorherigen Abschnitt skizziert, allerdings müssen einige semantische Feinheiten der Aktivitätsdiagramme und Implementierungsaspekte berücksichtigt werden.

Das umfangreichere Makro `CREATEFLOWTRANSITIONS` erfüllt diese Aufgabe und wird im Folgenden abschnittsweise erläutert. Es erhält als Eingabe die Menge der berechneten Flüsse und liefert eine Liste der Domäne `ModelCheckerInput` zurück. Die beiden Mengen `nodesInterrupted` und `nodesEntered` nehmen pro Fluss die im vorletzten Abschnitt bereits erwähnten Knoten, deren Umgebungen betreten oder abgebrochen werden, auf.

Das Flag `activityfinal` wird auf wahr gesetzt, sobald ein `ActivityFinalNode` im Flusseffekt enthalten ist. In Zeile 7 wird das (abstrakte) Makro `CREATESTEPHEADER` aufgerufen, welches den Code in der Model Checker Eingabesprache erzeugt, der notwendig ist, um die Definition eines neuen Zustandsübergangs einzuleiten. Alle weiteren Abschnitte werden für jeden Fluss der übergebenen Menge ausgeführt.

```

1 controlled  $dataSinkSum : ObjectNode \times Nat \rightarrow ModelCheckerInput$ 
2 CREATEFLOWTRANSITIONS  $: \mathcal{P}(Flow) \rightarrow ModelCheckerInput^*$ 
3 CREATEFLOWTRANSITIONS( $flows$ )  $\equiv$ 
4 local  $r : ModelCheckerInput^* = []$ 
5 local  $nodesInterrupted : \mathcal{P}(ActivityNode) = \emptyset$ 
6 local  $nodesEntered : \mathcal{P}(ActivityNode) = \emptyset$ 
7 local  $activityfinal : Boolean = false$ 
8  $r := r \oplus CREATESTEPHEADER$ 
9 foreach  $flow$  with  $flow \in flows$ 

```

Der nächste Abschnitt des Makros befasst sich mit der Erstellung der Vorbedingungen des aktuellen Flusses. Es wird davon ausgegangen, dass der Model Checker alle erzeugten Vorbedingungen in einer Konjunktion zusammenfasst. Für jede Aktion der Menge `actionsOffering` wird eine Abfrage, ob die entsprechende Variable den Wert `offeringToken` hat, erzeugt. Falls es sich um eine `AcceptEventAction` handelt, wird stattdessen eine Abfrage für die definierte Puffervariable erzeugt (vgl. Abschnitt 3.2.5.1).

Für die Elemente der Mengen `forksOffering` und `objectNodesOffering` werden Abfragen über die Mächtigkeit der Listen der entsprechenden Objektknoten erstellt. Im Fall der `ForkNodes` werden die Objektknoten über eine Indirektion mit Hilfe der Funktion `forkEdgeToObjectNode` ermittelt.

Der zweite Parameter des Makros `CREATELISTPRECONDITION` gibt an, ob auf eine leere (`true`) oder eine nicht-leere (`false`) Liste geprüft werden soll. In diesen Fällen müssen die Listen jeweils mindestens ein Element enthalten. Die im Fluss enthaltenen `⟨Guards⟩` werden auch in die Eingabesprache des Model Checkers übersetzt.

Daran schließt sich eine Abfrage an, ob die Zielobjektknoten überhaupt genügend Platz bieten, die fließenden Tokens aufzunehmen. Dabei werden einerseits die *ObjectNodes*, die sich in der Menge *objectNodesBuffering* befinden, überprüft. Andererseits werden die konkreten Datenflüsse aus der Menge *dataFlows* berücksichtigt.

Um herauszufinden, wieviel Platz tatsächlich gebraucht wird, werden zunächst in den Zeilen 28 bis 32 die Datensinken aufsummiert und dann in den Zeilen 34 bis 36 die entsprechenden Abfragen erzeugt. Schließlich werden die temporär erstellten Daten für den nächsten Fluss wieder zurückgesetzt.

```

10  foreach a with a ∈ flow.actionsOffering
11    if a ∈ AcceptEventAction then
12       $r := r \oplus \text{CREATEAEAMODEPRECONDITION}(a, \text{offeringToken})$ 
13    else
14       $r := r \oplus \text{CREATEMODEPRECONDITION}(a, \text{offeringToken})$ 
15  seq
16  foreach f with f ∈ flow.forksOffering
17     $r := r \oplus \text{CREATELISTPRECONDITION}(\text{forkEdgeToObjectNode}(f.First, f.Second), \text{false})$ 
18  seq
19  foreach o with o ∈ flow.objectNodesOffering
20     $r := r \oplus \text{CREATELISTPRECONDITION}(o, \text{false})$ 
21  seq
22  foreach g with g ∈ flow.guards
23     $r := r \oplus \text{CREATEGUARDPRECONDITION}(g)$ 
24  seq
25  foreach o with o ∈ flow.objectNodesBuffering
26     $r := r \oplus \text{CREATEBOUNDPRECONDITION}(o, 1)$ 
27  seq
28  forall n1, n2 with (n1, n2) ∈ flow.dataFlows
29    if dataSinkSum(n2) = undef then
30      dataSinkSum(n2) = 1
31    else
32      dataSinkSum(n2) = dataSinkSum(n2) + 1
33  seq
34  foreach o with o ∈ ObjectNode
35    if dataSinkSum(o) ≠ undef then
36       $r := r \oplus \text{CREATEBOUNDPRECONDITION}(o, \text{dataSinkSum}(o))$ 
37  forall o with o ∈ ObjectNode
38    dataSinkSum(o) = undef
39  seq

```

Damit ist die Umsetzung der Flussvorbedingung abgeschlossen. Bevor die Effekte übersetzt werden, müssen die von den betretenen und abgebrochenen *InterruptibleActivityRegions* betroffenen Knoten ermittelt werden. Die über eine Komprehension beschriebenen Mengen

werden anschließend unter Berücksichtigung der semantischen Variationspunkte angepasst.

```

40  nodesInterrupted := {n ∈ ActivityNode | ∃r ∈ n.inInterruptibleRegion :
41      r ∈ flow.regionsInterrupted}
42  nodesEntered := {n ∈ AcceptEventAction | predecessors(n) = ∅ ∧
43      immediateRegion(n) ∈ flow.regionsEntered}
44  seq

```

Die Menge der Knoten, die abgebrochen werden (*nodesInterrupted*), ist die Menge aller Knoten, die sich in einem der vom Fluss abgebrochenen Unterbrechungsbereiche befinden. Alle *AcceptEventActions*, die keine eingehenden Kanten haben und sich unmittelbar in einer betretenen Region befinden, müssen zusätzlich gestartet werden und bilden die Menge *nodesEntered*.

Alle abgebrochenen Regionen werden noch dahingehend überprüft, ob sich darin Aktionen befinden, die auf Grund des Flusses neu gestartet werden sollen. Falls durch das $\langle \text{Tag} \rangle$ *ignoreFlowIntoInterruptedRegion* für die betroffene Region festgelegt ist, dass eingehende Flüsse nicht ignoriert werden sollen, wird die Aktion aus der Menge der abgebrochenen Aktionen entfernt (Zeilen 44–47).

Für alle betretenen Regionen werden die darin liegenden Unterbrechungsbereiche überprüft, ob darin enthaltene *AcceptEventActions* bereits gestartet werden sollen, sobald die umliegenden Bereiche betreten werden. Dies wird über das $\langle \text{Tag} \rangle$ *regionActivationPolicy* festgelegt (*OnParentFlow*) (siehe Abschnitt 2.3.2).

Ist dies der Fall, werden die in den entsprechenden Regionen enthaltenen *AcceptEventActions* in die Menge *nodesEntered* aufgenommen. Da es sinnvoll ist (vgl. [Sar06a, Seite 25]), *AcceptEventActions* nicht mehr neu zu starten, wenn der umgebende Unterbrechungsbereich abgebrochen wurde, werden diese Knoten in Zeile 53 ausgeschlossen.

```

44  forall region with region ∈ flow.regionsInterrupted
45      if tagValue(region, IARHandling, ignoreFlowIntoInterruptedRegion) = false then
46          forall n with n ∈ flow.actionsStart ∧ region ∈ n.inInterruptibleRegion
47              remove n from nodesInterrupted
48  forall region with region ∈ flow.regionsEntered
49      forall c with c ∈ children(region)
50          if ∀r ∈ regionsFromTo(c, region) :
51              tagValue(r, IARHandling, regionActivationPolicy) = OnParentFlow then
52              foreach n with n ∈ AcceptEventAction ∧ c ∈ n.inInterruptibleRegion ∧
53                  n ∉ nodesInterrupted
54                  add n to nodesEntered
55  seq

```

Schließlich kann der Flusseffekt in die Eingabesprache des Model Checkers übersetzt werden. Dabei ist allerdings ein Sonderfall zu beachten: Befindet sich ein *ActivityFinalNode* in der Menge der zu startenden Knoten (ohne abgebrochene Knoten), werden alle weiteren Übergänge nicht mehr beachtet, sondern stattdessen die gesamte Aktivität beendet.

Dazu wird jeder Knoten der Aktivität auf *inactive* gesetzt. Die Überprüfung, ob für den entsprechenden Knoten tatsächlich eine Variable im Model Checker definiert wurde, erledigt ebenfalls das Makro **CREATEACTIONTRANSITION**, welches neben dem Knoten als zweiten Parameter den zu setzenden Modus übergeben bekommt.

Die zusätzlichen Puffervariablen der betroffenen *AcceptEventActions* werden ebenfalls zurückgesetzt und falls die Aktivität mit Hilfe einer *CallBehaviorAction* von einer anderen Aktivität aufgerufen wurde (Zeile 63), wird dorthin zurückgesprungen.

Die dafür notwendigen Schritte erledigt das Makro **CREATERETURN**, welches weiter unten beschrieben wird. Darin werden auch alle Datenpuffer der Aktivität geleert und Parameter zurückgeschrieben. Schließlich wird das Flag *activityfinal* gesetzt, um im weiteren Verlauf unnütze Ausführungen zu vermeiden.

```

56  if  $\exists a \in \textit{ActivityFinalNode} \mid a \in \text{flow.actionsStart} \setminus \textit{nodesInterrupted}$  then
57    foreach  $n$  with  $n \in \textit{ActivityNode} \wedge a.\textit{activity} = n.\textit{activity}$ 
58       $r := r \oplus \text{CREATEACTIONTRANSITION}(n, \textit{inactive})$ 
59    seq
60      if  $n \in \textit{AcceptEventAction}$  then
61         $r := r \oplus \text{CREATEAEABUFFERTRANSITION}(n, \textit{inactive})$ 
62      seq
63        if  $\textit{callingAction}(a.\textit{activity}) \neq \textit{undef}$  then
64           $r := r \uplus \text{CREATERETURN}(a.\textit{activity})$ 
65           $\textit{activityfinal} := \textit{true}$ 
66    else

```

Ab Zeile 67 wird der Fall behandelt, wenn sich *kein ActivityFinalNode* in der Menge der zu startenden Aktionen befindet.

Die Menge der abgebrochenen Aktionen wird von der Menge der zu startenden Aktionen entfernt und die Menge der zusätzlich zu startenden *AcceptEventActions* hinzugefügt (*nodesEntered*).

Knoten, die zwar abgebrochen, aber mit dem gleichen Fluss wieder gestartet werden, werden dabei ebenfalls entsprechend berücksichtigt. In Zeile 47 werden genau diese Knoten je nach Vorgabe des $\langle \textit{Tags} \rangle$ *ignoreFlowIntoInterruptedRegion* von der Menge *nodesInterrupted* entfernt. Daher werden sie in Zeile 67 *nicht* von der Menge der zu startenden Aktionen abgezogen.

Diese nun entsprechend angepasste Menge wird elementweise durchlaufen. Für die meisten Aktionen genügt es, ihren Modus auf *running* zu setzen (Zeile 90). Nur für *FlowFinalNodes* geschieht nichts und dadurch wird das Token einfach vernichtet. Für *CallBehaviorActions*, die eine andere Aktivität starten, ist eine Sonderbehandlung erforderlich (Zeilen 69–87).

Für *CallBehaviorActions* werden zunächst die eingehenden Datenflüsse auf die *ActivityParameterNodes* der zu startenden Aktivität gesetzt. Dabei wird auf ein Makro **PINTOPARAMETER** zurückgegriffen, welches aus der Arbeit von Sarstedt[Sar06a, Seite 53] übernommen wurde und die *Pins* auf die entsprechenden Parameter abbildet.

Die aufrufende Aktion in der Funktion *callingAction* wird quasi als Rücksprungadresse vermerkt (Zeile 76) (siehe auch **CREATERETURN**), da die Aktivität von der aktuellen Aktion aufgerufen wird (und es sich nicht um eine Startaktivität handelt).

Anschließend wird die neue Aktivität gestartet. Dies entspricht dem Anbieten von Tokens an allen in der Aktion vorhandenen *InitialNodes* und dem Starten aller Aktionen, die keine eingehenden Kanten haben und sich in keinem Unterbrechungsbereich befinden (Zeilen 77–82).

Schließlich muss die aufrufende Aktion entweder in den Modus *running* oder, falls es sich um einen asynchronen Aufruf handelt, in den Modus *offeringToken* versetzt werden, damit das

Token unmittelbar weiterfließen kann, ohne auf eine Beendigung der aufgerufenen Aktivität zu warten (Zeilen 84–87).

```

67   foreach a with a ∈ (flow.actionsStart \ nodesInterrupted) ∪ nodesEntered
68     if a ∈ CallBehaviorAction then
69       foreach p with p ∈ a.input
70         let
71           dataFlow = d : d ∈ flow.dataFlows ∧ d.Second = p
72           parameter = PINTOPARAMETER(p)
73         in
74           r := r ⊕ CREATEDATAFLOW(dataFlow.First, parameter)
75         seq
76         add a to callingAction(a.behavior)
77         foreach n with n ∈ ActivityNodes ∧ n.activity = a.behavior
78           if n ∈ InitialNode then
79             r := r ⊕ CREATEACTIONTRANSITION(n, offeringToken)
80           if n ∈ Action then
81             if predecessors(n) = ∅ ∧ n.inGroup = ∅ then
82               r := r ⊕ CREATEACTIONTRANSITION(n, running)
83             seq
84             if a.isSynchronous = false then
85               r := r ⊕ CREATEACTIONTRANSITION(a, offeringToken)
86             else
87               r := r ⊕ CREATEACTIONTRANSITION(a, running)
88             else
89               if a ∉ FlowFinalNode then
90                 r := r ⊕ CREATEACTIONTRANSITION(a, running)
91   seq

```

Die Abfrage in Zeile 92 stellt sicher, dass die folgenden Schritte nur durchgeführt werden, wenn nicht sowieso die komplette Aktivität beendet wird. Ist dies nicht der Fall, werden zunächst die Datenflüsse, die sich in der Menge *dataFlows* befinden, abgearbeitet.

Das nicht weiter spezifizierte Makro **C****R****E****A****T****E****D****A****T****A****F****L****O****W****W****I****T****H****O****U****T****R****E****M****O****V****E** fügt das erste Element des *ObjectNodes* d.*First* der Liste des *ObjectNodes* d.*Second* hinzu. Erst im nächsten Schritt werden die Daten aus dem Ursprungspuffer entfernt, da ein Objekt z. B. an *ForkNodes* auch an mehrere Ziele kopiert werden kann.

ObjectNodes der Mengen *forksBuffering* und *forksOffering*, die an einer Kontrollkante liegen, müssen extra behandelt werden, da die *ForkNode*-Puffer von Kontrollflüssen nicht in der Menge der Datenflüsse auftauchen. Dies erfolgt in den Zeilen 99 bis 105.

Fork-Puffern, die ein Token puffern sollen, wird ein Nulltoken hinzugefügt. Der Begriff „Nulltoken“ ist in der UML als ein Token definiert, welches keine Daten transportiert. Ist ein solches Nulltoken in der aktuellen Flussvorbedingung involviert, muss es mit Hilfe des Makros **C****R****E****A****T****E****D****A****T****A****R****E****M****O****V****E** aus dem Puffer gelöscht werden, um zu verhindern, dass es in einem späteren Schritt noch einmal berücksichtigt wird.

Ähnliches gilt für die Aktionen der Mengen *actionsOffering* und *nodesInterrupted*, deren Modus auf *inactive* gesetzt wird (vorausgesetzt, sie sind nicht in der Menge *nodesEntered* oder *actionsStart*). Wiederum muss beachtet werden, dass bei *AcceptEventActions* die ent-

sprechende Puffervariable berücksichtigt und Objektknoten eine leere Liste zugewiesen wird. Schließlich wird die Gesamtliste des erstellten ModelCheckerInputs zurückgeliefert.

```

92   if activityfinal = false then
93     foreach d with d ∈ flow.dataFlows
94       r := r ⊕ CREATEDATAFLOW(d.First, d.Second)
95     seq
96     foreach d with d ∈ flow.dataFlows
97       r := r ⊕ CREATEDATAREMOVE(d.First)
98     seq
99     foreach n, e with (n, e) ∈ flow.forksBuffering
100      if e ∈ ControlFlow ∧ n ∉ nodesInterrupted then
101        r := r ⊕ CREATENULLDATAADD(forkEdgeToObjectNodes(n, e))
102      seq
103      foreach n, e with (n, e) ∈ flow.forksOffering
104        if e ∈ ControlFlow then
105          r := r ⊕ CREATEDATAREMOVE(forkEdgeToObjectNodes(n, e))
106        seq
107        foreach n with n ∈ (flow.actionsOffering ∪ nodesInterrupted) \ nodesEntered ∧
108          n ∉ flow.actionsStart
109          if n ∈ Action then
110            r := r ⊕ CREATEACTIONTRANSITION(n, inactive)
111          seq
112          if n ∈ AcceptEventAction then
113            r := r ⊕ CREATEAEABUFFERTRANSITION(n, inactive)
114          if n ∈ ObjectNode then
115            r := r ⊕ CREATEEMPTYLIST(n)
116    seq
117    result := r

```

CreateReturn. Aktivitäten können über eine *CallBehaviorAction* synchron oder asynchron aufgerufen werden. Bei einem synchronen Aufruf verbleibt die aufrufende Aktion im Modus *running*, bis die aufgerufene Aktivität beendet ist.

Asynchron bedeutet, dass die aufrufende Aktivität weiterläuft, ohne auf das Ergebnis der aufgerufenen Aktivität zu warten. Die Ergebnisse der Subaktivität müssen dann über Signale oder andere Mechanismen zurückgesendet werden.

Das in Zeile 64 verwendete ASM-Makro **CREATERETURN** übernimmt die Abwicklung einer beendeten Aktivität inklusive der Rücksetzung aller Modi und Datenpuffer und kehrt gegebenenfalls (im synchronen Fall) zu den aufrufenden Aktionen zurück. Da, wie im Abschnitt 3.1 erläutert, für jede Aktivität nur eine Ausführung existiert (auch wenn eine Aktivität von verschiedenen Aktionen aus aufgerufen wird), werden bei deren Beendigung alle aufrufenden Aktionen beendet.

Dies gilt natürlich nur, wenn es sich um einen synchronen Aufruf handelt. Dann werden die von der Aktivität mit Daten versorgten *ActivityParameterNodes* auf die *OutputPins* der aufrufenden Aktionen abgebildet und die Aktionen in den Modus *offeringToken* versetzt.

Außerdem werden alle Puffer der in der Aktivität enthaltenen *ObjectNodes* geleert. Dies

muss nach der Übertragung der Parameter passieren, da anderenfalls die dort enthaltenen Daten verloren wären.

Im asynchronen Fall (wenn die *ActivityParameterNodes* noch keine Daten enthalten) muss das Makro **PARAMETERTOPIN** dafür sorgen, dass entsprechend der in dieser Arbeit verwendeten Semantik die *OutputPins* der aufrufenden Aktion mit `undef` gefüllt werden [Sar06a, Seite 52]. Bevor das Makro beendet wird, werden noch der Signalpuffer der Aktivität und die Menge der aufrufenden Aktionen zurückgesetzt.

```

CREATERETURN : Activity → ModelCheckerInput*
CREATERETURN(activity) ≡
  local r : ModelCheckerInput* = []
  foreach ca with ca ∈ callingAction(activity)
    if ca.isSynchronous = true then
      foreach p with p ∈ activity.ownedparameter
        let
          pin = PARAMETERTOPIN(p, ca)
        in
          r := r ⊕ CREATEDATAFLOWWITHOUTREMOVE(p, pin)
      seq
        r := r ⊕ CREATEACTIONTRANSITION(ca, offeringToken)
    seq
      foreach o with o ∈ ObjectNode ∧ o.activity = activity
        r := r ⊕ CREATEEMPTYLIST(o)
    seq
      r := r ⊕ CREATE SIGNALBUFFERREMOVEALL(activity)
    seq
      callingAction(activity) := undef
    seq
      result := r

```

3.2.5.3 Zustandsübergänge durch Aktionsausführungen

Neben den Zustandsübergängen, die auf Grund der Tokenflüsse in einem Aktivitätsdiagramm zustande kommen, wird der Systemzustand auch innerhalb von Aktionen verändert. Dabei gibt es erhebliche Unterschiede bezüglich der Aktionsart. Dieser Abschnitt beschreibt die ASM-Makros, die sich mit der Erstellung dieser „internen“ Zustandsübergänge befassen.

Das zentrale Makro bei dieser Aufgabe ist **CREATETRANSITIONSRUNNINGTOOFFERING**. Es erstellt für alle Aktionen die entsprechenden Zustandsübergänge. Dabei werden wieder abstrakte, hier nicht näher spezifizierte Makros zur Erstellung der eigentlichen Eingabesprache für den gewählten Model Checker verwendet. Außerdem sind einige komplexere Berechnungen in eigene Makros ausgelagert.

CallBehaviorActions. *CallBehaviorActions* wurden bereits im vorherigen Abschnitt im Rahmen der Zustandsübergänge aus Flüssen abgehandelt, so dass noch *CallOperationActions*, *SendSignalActions* und *AcceptEventActions* übrig bleiben.

CallOperationActions. Die Übersetzung von *CallOperationActions* in die Eingabesprache für den Model Checker ist relativ geradlinig. Dabei muss allerdings von der konkreten Umsetzung abstrahiert werden, da, wie am Ende des Abschnitts 3.1 dargestellt, die Möglichkeiten der Übersetzung stark von den Möglichkeiten des verwendeten Model Checkers abhängen.

Das prinzipielle Vorgehen sieht dabei folgendermaßen aus: Für jede *CallOperationAction* wird ein neuer Zustandsübergang mit dem Makro **CREATESTEPHEADER** eingeleitet. Als Voraussetzung genügt in diesem Fall, dass sich die Aktion im Modus *running* befindet und die *OutputPins* Platz für neue Daten bieten (Zeile 11 bzw. 13).

Das Makro **CREATECODE** übernimmt die eigentliche Übersetzung, nimmt die benötigten Parameter aus den *InputPin*-Puffern und schreibt gegebenenfalls auch die Ausgabedaten in die *OutputPins*. Sind diese Aufgaben erledigt, kann die Aktion ein Token anbieten (Zeile 17).

```

1 CREATETRANSITIONSRUNNINGTOOFFERING : void → ModelCheckerInput*
2 CREATETRANSITIONSRUNNINGTOOFFERING ≡
3   local r : ModelCheckerInput* = []
4   foreach coa with coa ∈ CallOperationAction
5     let
6       class = getContextClass(coa.activity)
7       code = locateCode(class, coa.operation)
8     in
9       r := r ⊕ CREATESTEPHEADER
10      seq
11      r := r ⊕ CREATEMODEPRECONDITION(coa, running)
12      seq
13      r := r ⊔ CREATEOUTPUTPRECONDITIONS(coa)
14      seq
15      r := r ⊔ CREATECODE(code)
16      seq
17      r := r ⊕ CREATEACTIONTRANSITION(coa, offeringToken)

```

SendSignalActions. Der nächste Teil des Makros befasst sich mit *SendSignalActions*. Bei der Behandlung dieser Aktionsart sind verschiedene Fälle zu unterscheiden:

1. Das Signal wird nicht gepuffert.
Wenn das Signal nicht gepuffert wird, müssen aktive *AcceptEventActions* das Signal sofort verarbeiten. Anderenfalls ist es verloren.
2. Das Signal wird gepuffert.
In diesem Fall lassen sich zwei weitere Fälle unterscheiden:
 - (a) Das Signal wird dem Puffer zusätzlich angefügt.
 - (b) Das Signal ersetzt ein eventuell bereits gepuffertes Signal.

In Zeile 19 findet die erste Fallunterscheidung in Abhängigkeit der Definition für den semantischen Variationspunkt *SignalBuffering* (siehe Abschnitt 2.3.2) statt. Wenn also entweder die Aktion selbst oder die umgebende Aktivität bestimmt, dass das Signal gepuffert werden soll, wird ein neuer Zustandsübergang mit Hilfe des Makros **CREATESTEPHEADER**

begonnen. Die Vorbedingung, dass die Aktion überhaupt läuft, wird hinzugefügt. Das Makro **CREATE SIGNAL DATA FROM** erstellt ein Datensignal aus den im *InputPin* der *SendSignalAction* übergebenen Daten. Daran schließt sich die zweite Fallunterscheidung an.

Es wird für jede Aktivität das entsprechende $\langle \text{Tag} \rangle$ überprüft, ob das neue Signal ein im Puffer vorhandenes Signal ersetzen soll oder nicht und entsprechende Eingaben für den Model Checker erzeugt. Zum Abschluss dieses Falls wird die Aktion in den Modus *offeringToken* versetzt.

Der Fall, dass das Signal nicht gepuffert wird, wird auf Grund der Komplexität von einem weiteren Makro **CREATE COMBINATIONS FOR HANDLING SIGNAL IMMEDIATELY** behandelt, welches eine Liste von Zustandsübergängen zurückliefert, die dem Gesamtergebnis hinzugefügt werden (Zeilen 37–41).

```

18  foreach ssa with ssa  $\in$  SendSignalAction
19    if tagValue(ssa, SignalBuffering, buffer) = Yes  $\vee$ 
20      (tagValue(ssa, SignalBuffering, buffer) = Unspecified  $\wedge$ 
21        tagValue(ssa.activity, SignalBuffering, buffer) = Yes) then
22       $r := r \oplus$  CREATESTEPHEADER
23      seq
24       $r := r \oplus$  CREATEMODEPRECONDITION(ssa, running)
25      seq
26       $r := r \oplus$  CREATE SIGNAL DATA FROM(ssa)
27      seq
28      foreach a with a  $\in$  Activity
29        if tagValue(ssa, SignalBuffering, replace) = No  $\wedge$ 
30          tagValue(a, SignalBuffering, replace) = No then
31           $r := r \oplus$  CREATE SIGNAL BUFFER CONCAT(a, ssa.signal)
32        else
33           $r := r \oplus$  CREATE SIGNAL BUFFER REPLACE(a, ssa.signal)
34        seq
35         $r := r \oplus$  CREATE ACTION TRANSITION(ssa, offeringToken)
36    else
37      let
38        rules = CREATE COMBINATIONS FOR HANDLING SIGNAL IMMEDIATELY(ssa)
39      in
40      foreach res with res  $\in$  rules
41       $r := r \uplus$  res

```

AcceptEventActions. Die Behandlung von *AcceptEventActions* ist komplexer als das der bisher behandelten Aktionen. *AcceptEventActions* müssen gleichzeitig auf ein im Puffer vorhandenes Signal reagieren. Anderenfalls würde immer nur genau eine *AcceptEventAction* ein Signal empfangen, was wiederum dem Sinn des Broadcasts (siehe Abschnitt 3.1) widersprechen würde.

Es müssen daher für alle möglichen Kombinationen der entsprechenden *AcceptEventActions* Zustandsübergänge erstellt werden, da nicht immer alle möglichen *AcceptEventActions* zu jedem Zeitpunkt aktiv sind. Diese Aufgabe ist auf das Makro **CREATE COMBINATIONS FOR HANDLING SIGNAL** ausgelagert, welches im Folgenden näher beschrieben wird.

```

42 foreach signal with signal  $\in$  Signal
43    $r := r \uplus$  CREATECOMBINATIONSFORHANDLINGSIGNAL(signal)
44 seq
45 result :=  $r$ 

```

Um alle Kombinationen der *AcceptEventActions* für ein Signal zu erhalten, wird auf das Makro **POWERSET** zurückgegriffen, welches die Potenzmenge über einer Menge bestimmt (Zeile 5). Als nächster Schritt wird für jedes Element der Potenzmenge ein Zustandsübergang initialisiert und die Vorbedingungen pro Aktion generiert (Zeilen 8ff). Die Vorbedingung, dass ein Signal im Puffer der entsprechenden Aktivität vorhanden ist, wird allerdings nur für interne Signale erzeugt (Zeile 13).

Als internes Signal werden in dieser Arbeit Signale definiert, die auch aus einem der bearbeiteten Aktivitätsdiagramme heraus erzeugt werden können, also eine *SendSignalAction* dafür existiert. Da über externe Signale keine Aussage getroffen werden kann, wird davon ausgegangen, dass diese zu jedem Zeitpunkt auftreten können und sich sozusagen „immer“ im Signalpuffer der Aktivität befinden. Daher wird keine entsprechende Vorbedingung für externe Signale generiert.

Als Effekt einer *AcceptEventAction* wird das im Signal transportierte Datum auf den *OutputPin* der Aktion übertragen und gegebenenfalls das Signal aus dem Puffer entfernt (Zeile 17 bzw. Zeile 22).

Eine wichtige Rolle spielt das Makro **CREATEMUSTSTAYACTIVECHECK**. Es überprüft, ob eine *AcceptEventAction* ohne eingehende Kanten nach dem Signalempfang aktiv bleibt. Dies ist genau dann der Fall, wenn die umgebende Region ebenfalls noch aktiv ist. Dementsprechend wird der Modus (und die dazugehörige Puffervariable) der *AcceptEventAction* gesetzt. Eine ausführlichere Erläuterung findet sich weiter unten.

```

1 CREATECOMBINATIONSFORHANDLINGSIGNAL : Signal  $\rightarrow$  ModelCheckerInput*
2 CREATECOMBINATIONSFORHANDLINGSIGNAL(signal)  $\equiv$ 
3 local  $r$  : ModelCheckerInput* = []
4 let
5   powerset = POWERSET({ $a \in$  AcceptEventAction |  $a.trigger.event.signal = signal$ })
6 in
7   foreach set with set  $\in$  powerset
8      $r := r \oplus$  CREATESTEPHEADER
9     seq
10    foreach aea with aea  $\in$  set
11       $r := r \oplus$  CREATEMODEPRECONDITION(aea, running)
12      seq
13      if internal(signal) then
14         $r := r \oplus$  CREATESIGNALBUFFERPRECONDITION(aea.activity, signal)
15      seq
16      foreach aea with aea  $\in$  set
17         $r := r \oplus$  CREATESIGNALDATATo(signal, aea)
18        seq
19         $r := r \uplus$  CREATEMUSTSTAYACTIVECHECK(aea)
20        seq
21        if internal(signal) then

```

```

22      $r := r \oplus \text{CREATE\_SIGNAL\_BUFFER\_REMOVE}(\text{aea.}activity, signal)$ 
23   seq
24   result :=  $r$ 

```

Der Aufbau des Makros `CREATE_COMBINATIONS_FOR_HANDLING_SIGNAL_IMMEDIATELY`, welches im Makro `CREATE_TRANSITIONS_RUNNING_TO_OFFERING` zur Behandlung von ungepufferten Signalen aufgerufen wurde, ist sehr ähnlich zum eben vorgestellten Makro zur Behandlung von `AcceptEventActions`.

Die wesentlichen Unterschiede finden sich in der Erstellung der Vorbedingung und des Datenflusses. Die Überprüfung, ob das Signal im Puffer vorhanden ist, entfällt, da das Signal nicht gepuffert, sondern direkt verarbeitet wird. Außerdem ist eine sofortige Verarbeitung nur für interne Signale möglich. Stattdessen muss sich die betrachtete `SendSignalAction` im Modus `running` befinden (Zeile 11).

Da in diesem Fall kein Signal erzeugt wird, werden die möglicherweise transportierten Daten jeweils direkt in den `OutputPin` der empfangenden `AcceptEventActions` geschrieben (Zeilen 16–19). Schließlich werden die Modi der `AcceptEventActions` in Abhängigkeit ihrer Umgebung gesetzt und ein Token an der ausgehenden Kante der `SendSignalAction` angeboten.

```

1  CREATE\_COMBINATIONS\_FOR\_HANDLING\_SIGNAL\_IMMEDIATELY :
2       $SendSignalAction \rightarrow ModelCheckerInput^*$ 
3  CREATE\_COMBINATIONS\_FOR\_HANDLING\_SIGNAL\_IMMEDIATELY( $ssa$ )  $\equiv$ 
4  local  $r : ModelCheckerInput^* = []$ 
5  let
6      powerset =  $\text{POWERSET}(\{a \in AcceptEventAction \mid a.trigger.event.signal = ssa.signal\})$ 
7  in
8      foreach set with set  $\in$  powerset
9           $r := r \oplus \text{CREATE\_STEP\_HEADER}$ 
10         seq
11          $r := r \oplus \text{CREATE\_MODE\_PRE\_CONDITION}(ssa, running)$ 
12         seq
13         foreach aea with aea  $\in$  set
14              $r := r \oplus \text{CREATE\_MODE\_PRE\_CONDITION}(aea, running)$ 
15         seq
16         if  $|ssa.input| > 0$  then
17             foreach aea with aea  $\in$  set
18                  $r := r \oplus \text{CREATE\_DATA\_FLOW\_WITHOUT\_REMOVE}$ 
19                     ( $first(asList(ssa.input)), first(asList(aea.output))$ )
20             seq
21              $r := r \oplus \text{CREATE\_DATA\_REMOVE}(first(asList(ssa.input)))$ 
22         seq
23         foreach aea with aea  $\in$  set
24              $r := r \uplus \text{CREATE\_MUST\_STAY\_ACTIVE\_CHECK}(aea)$ 
25         seq
26          $r := r \oplus \text{CREATE\_ACTION\_TRANSITION}(ssa, offeringToken)$ 
27     seq
28     result :=  $r$ 

```

Die beiden zuletzt vorgestellten Makros benutzen `CREATEMUSTSTAYACTIVECHECK`, um Eingaben für den Model Checker zu erzeugen, die überprüfen, ob die als Parameter übergebene *AcceptEventAction* beendet werden darf oder nicht. Der Modus der Variablen wird entsprechend gesetzt. Im Folgenden wird die Funktionsweise dieses Makros näher erläutert.

Die erste Abfrage in Zeile 4 entscheidet, ob es sich bei der betrachteten Aktion um eine Aktion ohne eingehende Kanten handelt. Ist dies *nicht* der Fall, wird der Modus einfach auf `offeringToken` gesetzt und das Makro ist beendet (Zeilen 24–26).

Anderenfalls wird unterschieden, ob sich die Aktion in einem Unterbrechungsbereich befindet oder nicht (Zeile 5). Falls nicht, läuft die Aktion per Definition der UML auf alle Fälle weiter und nur die entsprechende *AcceptEventAction*-Puffervariable bietet ein Token an (Zeilen 6–8). Befindet sie die *AcceptEventAction* jedoch in einem Unterbrechungsbereich, wird das Verfahren etwas komplizierter.

```

1 CREATEMUSTSTAYACTIVECHECK : AcceptEventAction → ModelCheckerInput*
2 CREATEMUSTSTAYACTIVECHECK(action) ≡
3   local r : ModelCheckerInput* = []
4   if action.incoming ∪ action.input = ∅ then
5     if action.inInterruptibleRegion = ∅ then
6       r := CREATEACTIONTRANSITION(action, running)
7       seq
8       r := r ⊕ CREATEAEABUFFERTRANSITION(action, offeringToken)
9     else
10      let
11        set = CREATEMUSTSTAYACTIVECHECKSET(action)
12      in
13        r := r ⊕ CREATEAEABUFFERTRANSITION(action, offeringToken)
14        seq
15        r := r ⊕ CREATEIFMODEFORNODES(set, inactive)
16        seq
17        r := r ⊕ CREATEACTIONTRANSITION(action, inactive)
18        seq
19        let
20          elsecode = CREATEACTIONTRANSITION(action, running)
21        in
22          r := r ⊕ CREATEELSE(elsecode)
23      else
24        r := CREATEACTIONTRANSITION(action, offeringToken)
25    seq
26  result := r

```

Zunächst werden im Makro `CREATEMUSTSTAYACTIVECHECKSET` die Aktionen bestimmt, die zur Entscheidung, ob eine *AcceptEventAction* aktiv bleibt oder nicht, hinzugenommen werden müssen. *AcceptEventActions* in *InterruptibleActivityRegions* bleiben nämlich nur so lange aktiv, wie der unmittelbar umgebende Bereich aktiv ist.

Ein Bereich ist genau dann inaktiv, wenn alle darin enthaltenen Aktionen inaktiv und alle darin enthaltenen *ObjectNode*-Puffer leer sind. Dieses Verhalten kann zudem über den semantischen Variationspunkt `regionReactivationPolicy` dahingehend beeinflusst werden, dass

eine *AcceptEventAction* auch aktiv bleibt, wenn nur eine ihrer umgebenden Regionen aktiv ist.

Dazu wird für alle Regionen, in denen sich der Signalempfänger befindet, geprüft, ob das $\langle \text{Tag} \rangle$ *regionReactivationPolicy* den Wert *OnParentActive* hat. Ist dies der Fall, werden alle Knoten, die sich im aktuell betrachteten Unterbrechungsbereich befinden und kein *ControlNode* (außer *InitialNode*) sind, der Ergebnismenge hinzugefügt.

Für diese Menge wird im ursprünglichen Makro **CREATEMUSTSTAYACTIVECHECK** eine Abfrage erzeugt (Zeile 15), die überprüft, ob alle Elemente *inactive* oder leer sind (je nachdem ob es ein *ObjectNode* oder eine *Action* ist). Ist dies der Fall, wird die *AcceptEventAction* ebenfalls auf *inactive* gesetzt und nur die Puffer-Variable auf *offeringToken*. Ist eines der Elemente der Menge aber nicht inaktiv, bleibt auch die *AcceptEventAction* im Modus *running* (Zeilen 19–22).

CREATEMUSTSTAYACTIVECHECKSET : $\mathcal{P}(\text{AcceptEventAction}) \rightarrow \mathcal{P}(\text{ActivityNode})$

CREATEMUSTSTAYACTIVECHECKSET(*action*) \equiv

local *res* : $\mathcal{P}(\text{ActivityNode}) = \emptyset$

let

ir = *immediateRegion*(*action*)

in

forall *r* **with** $r \in \text{action.inInterruptibleActivityRegion}$

if $\text{ir} = r \vee \forall r' \in \text{regionsFromTo}(\text{ir}, r)$:

tagValue(*r*, *IARHandling*, *regionReactivationPolicy*) = *OnParentActive* **then**

forall *n* **with** $n \in \text{Action} \cup \text{InitialNode} \cup \text{ObjectNode} \wedge n \in r.\text{node}$

add *n* **to** *res*

seq

result := *res*

Hiermit sind alle möglichen Zustandsübergänge innerhalb von Aktionen für den Model Checker aufbereitet. Für jede Aktivität wird noch ein Zustandsübergang generiert, der aktiviert wird, sobald eine Aktivität nicht mehr aktiv ist *ohne* dass ein *ActivityFinalNode* aktiviert wurde. Dieser Fall tritt z. B. auf, wenn alle Tokens in *FlowFinalNodes* vernichtet wurden oder wenn alle Aktionen ohne Nachfolger — wie in Beispiel aus Abb. 3.8 die Aktionen C, D, E und F — beendet sind.

Das entsprechende Makro lautet **CREATEFINISHCHECKS**, für dessen Spezifikation und Erläuterung auf Anhang A.3 verwiesen wird. Bei Sarstedt fehlt diese Abfrage mit der Folge, dass das System in einen Deadlock gelangen könnte, ohne dass dieser im Diagramm irgendwie ersichtlich wird.

Abschließend werden die vorgestellten ASM-Regeln im Makro **CREATETRANSITIONS** zusammengefasst und in eine Ausführungsreihenfolge gebracht, wobei diese (bis auf die Präambel) eigentlich unerheblich ist.

CREATETRANSITIONS : $\mathcal{P}(\text{Flow}) \rightarrow \text{ModelCheckerInput}^*$

CREATETRANSITIONS(*flows*) \equiv

local *r* : $\text{ModelCheckerInput}^* = []$

r := *r* \uplus **CREATEPREAMBLE**

seq

```

 $r := r \uplus \text{CREATETRANSITIONSRUNNINGTOOFFERING}$ 
seq
 $r := r \uplus \text{CREATEFINISHCHECKS}$ 
seq
 $r := r \uplus \text{CREATEFLOWTRANSITIONS}(flows)$ 
seq
result :=  $r$ 

```

3.3 Zur Korrektheit der Transformation

Nach der Beschreibung des formalisierten Transformationsalgorithmus widmet sich dieser Abschnitt der Überprüfung der Korrektheit der vorgestellten Transformation. Für einen sinnvollen Einsatz des Model Checkings ist es wichtig, dass dessen Ergebnisse dem Verhalten bei Ausführung eines Aktivitätsdiagramms entsprechen. Dies ist nur gewährleistet, wenn das ursprüngliche Aktivitätsdiagramm und das erstellte Zustandsübergangssystem semantisch äquivalent sind.

Anderenfalls könnte es passieren, dass das Model Checking ein Gegenbeispiel generiert, welches in der Ausführung nicht nachvollzogen werden kann. Schlimmer noch wäre der Fall, dass das Model Checking auf Grund der geänderten Semantik *keine* Verletzung der spezifizierten Eigenschaft findet, obwohl ein Fehlerfall bei der Ausführung auftreten kann.

Der Begriff „Verhalten der Aktivitätsdiagramme“ bezeichnet im Folgenden das Verhalten, welches durch die formale Semantik der Aktivitätsdiagramme wie sie von Sarstedt in [Sar06a] definiert wurde, bestimmt ist. Der Begriff „Verhalten des Zustandsübergangssystems“ bezieht sich auf das durch den in der vorliegenden Arbeit vorgestellten Algorithmus generierte Zustandsübergangssystem.

3.3.1 Deduktiver Nachweis

Das Verhalten des Zustandsübergangssystems muss dem Verhalten der Aktivitätsdiagramme entsprechen. Um dies zu zeigen, ist ein formaler, deduktiver Nachweis erforderlich, dass das Verhalten der Aktivitätsdiagramme die gleichen Zustände einnimmt wie das Zustandsübergangssystem. Interne Zwischenzustände dürfen auf Grund der unterschiedlichen Verarbeitung dabei nicht berücksichtigt werden.

Ein Hauptgrund für die Formalisierung des oben vorgestellten Algorithmus in ASMs ist, diesen Nachweis zu ermöglichen ohne zunächst zwischen verschiedenen Formalismen konvertieren zu müssen (und dadurch evtl. neue Fehler einzubringen).

Die Regeln sowohl bei Sarstedt als auch in der vorliegenden Arbeit sind so umfangreich und komplex, dass ein formaler Nachweis genügend Arbeit für eine weitere Arbeit bedeuten würde. Im Folgenden wird daher nur die prinzipielle Vorgehensweise für diesen Nachweis dargestellt. Ein rein manueller Beweis müsste außerdem mit Hilfe eines maschinellen Theorembeweislers überprüft werden. Ziel weiterer Arbeiten könnte sein, die Äquivalenz zumindest für Teile (z. B. nur des Kontrollflusses) nachzuweisen und dann sukzessive den kompletten Nachweis aufzubauen.

Der Korrektheitsnachweis müsste in folgenden Schritten durchgeführt werden:

1. Angleichung der Semantik von Sarstedt an die in der vorliegenden Arbeit vorgestellten Änderungen (*ForkNode*-Pufferung, semantische Variationspunkte).

2. Berücksichtigung der Einschränkungen aus Abschnitt 3.1.
3. Strukturelle Zerlegung zum Nachweis
 - (a) von äquivalentem Verhalten der einzelnen Aktionen.
 - (b) der Übereinstimmung der Flussberechnung.

Die einzelnen Schritte werden im Folgenden ausführlicher erläutert.

Angleichung. Zunächst müssen die in der vorliegenden Arbeit vorgenommenen Änderungen in die Semantik von Sarstedt [Sar06a] eingearbeitet werden. Konkret trifft dies vor allem auf das geänderte Verhalten von *ForkNodes* und die Anpassung des semantischen Variationspunkts von Signalen (siehe Abschnitt 2.3.2) zu.

Berücksichtigung der Einschränkungen. Sarstedt hatte als primäres Ziel die Ausführbarkeit der Aktivitätsdiagramme. Daher wurden in der Semantikbeschreibung einige Konzepte verwendet, die für die Transformation in Zustandsübergangssysteme nicht übernommen werden konnten (siehe Abschnitt 3.1). Das Hauptproblem liegt darin, dass Sarstedt während der Berechnung neue Objekte erzeugt, was, wie in Abschnitt 3.1 bereits erläutert, für das Model Checking nur schwer zu realisieren ist.

Um trotz dieser Unterschiede die semantische Äquivalenz der Beschreibungen zu zeigen, müssen die in der vorliegenden Arbeit vorgestellten Änderungen und Erweiterungen in die formale Semantik von Sarstedt eingearbeitet werden. Dies betrifft vor allem die Verteilung von Signalen, das Timerverhalten und die Beschränkung auf eine Ausführung pro Aktivität bzw. Aktion.

Strukturelle Zerlegung. Nach Erledigung der oben genannten Punkte, wird die gesamte Spezifikation in verschiedene, möglichst unabhängige Teilaspekte zerlegt. Die Einteilung erfolgt zunächst in das Verhalten der verschiedenen Aktionen einerseits und die Tokenflussberechnung andererseits. Für jeden dieser Bereiche muss nun wieder eine vollständige strukturelle Zerlegung angegeben werden.

Äquivalentes Verhalten der Aktionen. Das Verhalten der Aktionen kann pro Aktionsart (z. B. *CallBehaviorAction*, *SendSignalAction*, *CallOperationAction*, usw.) gegen das Verhalten des entsprechenden Zustandsübergangs verifiziert werden.

In der Arbeit von Sarstedt werden für die Ausführungen der Aktivitäten und der Aktionen sogenannte $\langle \text{Activity-} \rangle$ bzw. $\langle \text{ActionExecutions} \rangle$ erzeugt, die die eigentliche Verarbeitung ereignisgesteuert übernehmen. Jede $\langle \text{Execution} \rangle$ wird in einem neuen ASM-Agenten ausgeführt (siehe Abschnitt 2.2.3).

Dieses „verteilte“ Vorgehen wird bei Sarstedt im Makro `CREATEACTIONEXECUTION` (siehe [Sar06a, Seite 49]) mit verschiedenen Untermakros beschrieben. Das Verhalten der Aktionen im Zustandsübergangssystem ist weitgehend im Makro `CREATETRANSITIONSRUNNING-TOOFFERING` (siehe Seite 75) zusammengefasst.

Für einen Nachweis müssen also diese beiden Regeln dahingehend überprüft werden, ob sich das System nach Ausführung einer Aktion jeweils im gleichen Zustand befindet.

Äquivalenznachweis der Flussberechnung. Die Berechnung der Tokenangebote ist durch den operationellen Charakter der Beschreibung von Sarstedt für das Model Checking unbrauchbar. Bei Sarstedt werden, sobald ein neues Tokenangebot über ein internes Ereignis signalisiert wurde, durch das Makro `HANDLEOFFEREVENT` alle Tokenangebote über die Kanten propagiert. Dabei ist es unerheblich, ob das neue Tokenangebot mit den bisherigen, nicht verarbeiteten, überhaupt in Beziehung steht. Anschließend wird ein Tokenangebot an einer Aktion ausgewählt und die entsprechende Transition durchgeführt.

Diese Vorgehensweise führt nicht nur im Rahmen des Model Checkings zu Problemen. Sie bringt auch die von Sarstedt implementierte Interpretation von Aktivitätsdiagrammen schnell an ihre Grenzen. Aus diesem Grund werden in der vorliegenden Arbeit alle möglichen Flüsse vorab berechnet und als Zustandsübergänge beschrieben. Für jede mögliche Tokenangebotsmenge und für jede mögliche Selektion daraus wird ein Zustandsübergang erzeugt.

Durch diese unterschiedliche Vorgehensweise reicht es allerdings nicht aus, z. B. für jeden Kontrollknoten die Äquivalenz zu überprüfen. Stattdessen ist es notwendig, einen Fluss in seiner Gesamtheit mit allen beteiligten Aktionen, Objekt- und Kontrollknoten zu betrachten.

Dementsprechend ist eine strukturelle Zerlegung der Aktivitätsdiagramme erheblich komplexer. Jede mögliche Kombination von Kontrollknoten zwischen Aktionen muss für theoretisch unendlich viele beteiligte Aktionen und *ObjectNodes* untersucht werden. Abb. 3.16 zeigt einige der nur für Kontrollflüsse zu betrachtenden Kombinationen.

Für jeden dieser Fälle muss nun gezeigt werden, dass die Flussberechnung von Sarstedt mit der Flussberechnung der vorliegenden Arbeit übereinstimmt. Auch mögliche Wechselwirkungen mit Unterbrechungsbereichen und -kanten müssen dabei berücksichtigt werden.

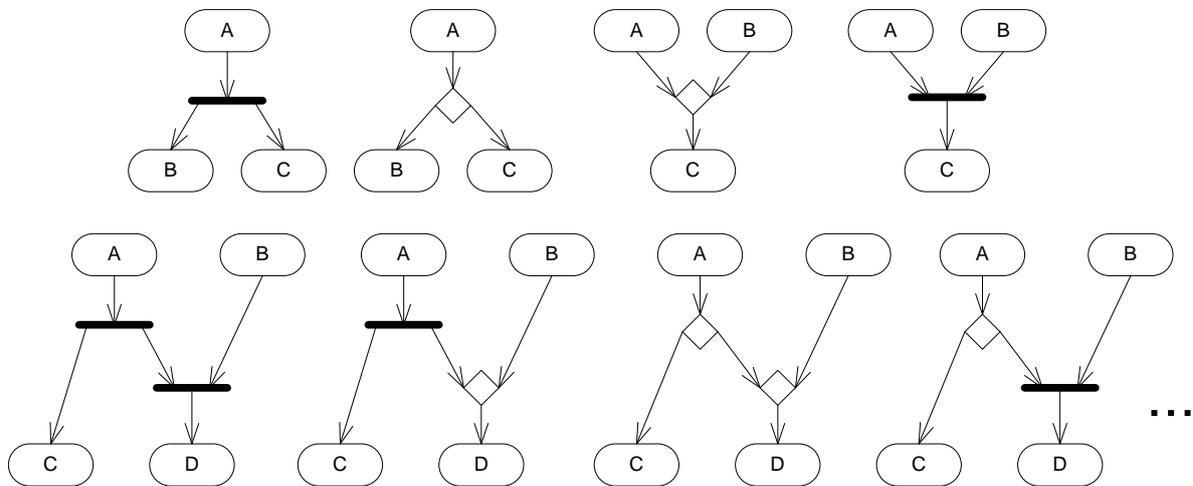


Abbildung 3.16: Einige Kombinationen von Kontrollknoten und Aktionen zur strukturellen Zerlegung von Aktivitätsdiagrammen.

Die Zerlegung muss vollständig sein, so dass alle Verknüpfungsmöglichkeiten beliebiger Aktionen damit konstruiert werden können. Dass dies tatsächlich der Fall ist, muss ebenfalls bewiesen werden.

Es zeigt sich also, dass der deduktive Nachweis der semantischen Äquivalenz zwischen der formalen Semantik und dem generierten Zustandsübergangssystem alles andere als trivial ist.

3.3.2 Konformitätstest

Wie im letzten Abschnitt beschrieben, ist der formale Nachweis der semantischen Äquivalenz der Aktivitätsdiagramme und des erstellten Zustandsübergangssystems nicht einfach. Eine andere, nicht formale Möglichkeit, diesem Ziel einen Schritt näher zu kommen, ist der sogenannte Konformitätstest.

Im Allgemeinen werden Konformitätstests dazu verwendet, die Übereinstimmung einer Implementierung mit einer (formalen) Spezifikation — zumindest für die getesteten Fälle — zu zeigen. Er wird typischerweise bei der Überprüfung von Protokollen eingesetzt, bei denen gezeigt werden soll, dass sich eine Implementierung gemäß dem Standard verhält.

Dieses Verfahren könnte auch zur Überprüfung der Korrektheit dieser Arbeit eingesetzt werden. Dazu müssten beliebige Beispiele von Aktivitätsdiagrammen in das korrespondierende Zustandsübergangssystem übersetzt und in beiden Spezifikationen ausgeführt werden. Dann kann für jeden Schritt überprüft werden, ob sich die Ausführungen im gleichen Zustand befinden oder nicht, wobei „interne“ Zustände, wie oben erwähnt, übersprungen werden müssen.

Durch diesen Konformitätstest lassen sich allerdings — wie bei Tests üblich — lediglich Fehler finden und nicht die Korrektheit oder die Vollständigkeit der Transformation nachweisen.

Kapitel 4

Prototypische Umsetzung mit Model Programs und weitere Aspekte

Nach der Erstellung eines abstrakten Zustandsübergangssystems zeigt Abschnitt 4.1 eine Möglichkeit, eine Eingabe für einen bestimmten Model Checker zu generieren. Da Model Checking häufig auch ein zumindest rudimentäres Modell der Umgebung erfordert, werden einige dieser Aspekte in Bezug auf Aktivitätsdiagramme in Abschnitt 4.2 vorgestellt. Schließlich werden im letzten Abschnitt dieses Kapitels noch einige mögliche Optimierungen bei der automatischen Verifikation von Aktivitätsdiagrammen auf verschiedenen Abstraktionsebenen aufgezeigt.

4.1 Prototypische Umsetzung mit Model Programs

Nach der formalen Spezifikation der Bestimmung von Zustandsübergängen aus Aktivitätsdiagrammen im vorherigen Kapitel, widmet sich dieser Abschnitt der Beschreibung, wie eine konkrete Umsetzung für einen Model Checker aussehen könnte. Dabei wird auch auf allgemeine Aspekte, die bei einer Realisierung berücksichtigt werden müssen, eingegangen.

Die dargestellten Erkenntnisse beruhen auf einer prototypischen Implementierung, die die meisten Aspekte des vorgestellten Algorithmus abdeckt. Sie ist in das am Institut für Programmiermethodik und Compilerbau der Universität Ulm entwickelte Werkzeug ActiveChartsIDE integriert. Aktivitäts- und Klassendiagramme werden von diesem Werkzeug aus Microsoft Visio eingelesen, auf syntaktische Fehler überprüft und dann zusammen mit ergänztem Code interpretiert.

Für eine ausführliche Beschreibung dieses verwendeten Werkzeugs sei auf [Sar06b] verwiesen. Die prototypische Implementierung nutzt die bestehende Möglichkeit, Aktivitätsdiagramme einzulesen. Anschließend berechnet sie die Flüsse und Zustandsübergänge wie in dieser Arbeit beschrieben und liefert eine Eingabedatei für den Model Checker zurück.

Der folgende Abschnitt führt kurz in den Formalismus der Model Programs ein und beschreibt die Vor- und Nachteile des dafür existierenden Model Checkers. Anschließend wird konkret auf die Umsetzung der Zustandsübergänge in Model Programs eingegangen und kurz alternative Lösungen diskutiert.

Model Programs wurden aus mehreren Gründen als beispielhafte Umsetzung ausgewählt. Der Formalismus entspricht im Wesentlichen den in Abschnitt 3.2.5 vorgestellten Guarded Commands. Dies vereinfacht die Übersetzung des Zustandsübergangssystems in Model Programs erheblich. Außerdem existieren für diesen Formalismus auch Werkzeuge die ähnlich dem expliziten Model Checking funktionieren und daher auch komplexere Objektflüsse verarbeitet werden können.

4.1.1 Model Programs

Der Formalismus der Model Programs wurde bei Microsoft Research zur formalen Spezifikation von Systemen und Protokollen entwickelt. Die Hauptanwendung liegt im Bereich des Model Based Testing, welches an Hand einer gegebenen formalen Spezifikation Testfälle generiert und eine Implementierung dagegen testet. Eine ausführliche Beschreibung für diese Anwendung findet sich in [JVCS08].

Model Programs bestehen aus getypten Variablen, einem Startzustand für jede Variable und einer Menge von Regeln, die „Actions“ genannt werden. Diese Variablen werden auch Zustandsvariablen genannt, da ihre Inhalte den Systemzustand beschreiben. Für eine leichtere Unterscheidung von Aktionen aus Aktivitätsdiagrammen, wird für Actions aus Model Programs konsequent nur die englische Bezeichnung verwendet.

Jede Action hat eine Signatur mit einem Bezeichner, optional Parametern und einem Rumpf. Der Rumpf besteht aus Vorbedingungen und verschiedenen Anweisungen, die — ähnlich zu Abstract State Machines — parallel ausgeführt werden. Abb. 4.1 zeigt ein einfaches Beispiel für ein Model Program.

In diesem Beispiel wird zunächst ein Aufzählungstyp `MyEnum` mit den Elementen `one` und `two` definiert. Als nächstes werden drei Systemvariablen mit ihrem jeweiligen Typ und Initialzustand definiert. `y` und `z` sind Integervariablen, `m` ist eine Menge des Aufzählungstyps `MyEnum`.

Die Action `Incr` hat zwei Parameter vom Typ `Integer` und `Set of MyEnum`, die Action `Swap` keine. Vorbedingungen von Actions werden mit dem Schlüsselwort `require` eingeleitet und werden mit einer Konjunktion verknüpft. Das Schlüsselwort `skip` steht für die leere Anweisung. Blöcke werden nur durch Einrückung kenntlich gemacht. Als Anweisungen stehen unter anderem Zuweisungen und Verzweigungen zur Verfügung. Allerdings wird keinerlei Schleifenkonstrukt oder Rekursion angeboten.

Für Model Programs existiert nur genau eine Hauptschleife, die nicht-deterministisch eine Action auswählt und — falls deren Vorbedingungen erfüllt sind — die darin enthaltenen Anweisungen ausführt. Parameter werden dabei entweder mit zufälligen Werten, mit Werten aus einem bestimmten Wertebereich (im Fall des Model Based Testing) oder für das Model Checking mit den Werten gefüllt, die die zu prüfende Eigenschaft verletzen (näheres dazu in Abschnitt 4.1.2).

In obigem Beispiel würde zunächst `Incr` ausgeführt werden, da `z` den Wert 5 hat und die Vorbedingung für `Swap` nicht erfüllt ist (`two ∉ m`). Je nach Belegung des Parameters `param2` wird `z` erhöht oder auch nicht. Auf alle Fälle wird das Element `two` der Menge `m` hinzugefügt. Angenommen, `z` wurde erhöht, dann kommt im nächsten Schritt die Action `Swap` sicher zur Ausführung, da nur deren Bedingung erfüllt ist. Auf Grund der parallelen Ausführung der Anweisungen (die konsistent sein müssen), werden darin die Werte von `z` und `y` vertauscht und `m` auf die leere Menge zurückgesetzt.

```

enum MyEnum
  one
  two

var m as Set of MyEnum = {one}
var y as Integer = 2
var z as Integer = 5

[Action]
Incr(param as Set of MyEnum)
  require z <= 5
  if one in param
    skip
  else
    z := z + 1
    m := m + {two}

[Action]
Swap()
  require z > 3
  require two in m
  z := y
  y := z
  m := {}

```

Abbildung 4.1: Beispiel für ein Model Program.

Wie oben bereits erwähnt, passen die Model Programs aus mehreren Gründen als Eingabesprache für das Model Checking von Aktivitätsdiagrammen:

- Der Aufbau der Actions mit Vorbedingungen und Anweisungen entspricht im Wesentlichen den in Abschnitt 3.2.5 vorgestellten Guarded Commands. Dementsprechend ist eine Übersetzung der abstrakten Zustandsübergänge in Model Programs relativ einfach anzugeben.
- Jeder Zustandsübergang ist mit den Vorbedingungen und den Effekten in einer Action zusammengefasst.
- Die Benennung der Actions kann bei der Rekonstruktion eines Ablaufs aus dem eventuell vom Model Checker zurückgelieferten Gegenbeispiel hilfreich sein.
- Der in [VBR08] vorgestellte Model Program Checker (MPC) unterstützt viele auch höherwertige Datentypen, die die Übersetzung vereinfachen.

Allerdings ist nur Bounded Model Checking möglich, da der MPC ein SMT-Model Checker ist (siehe Abschnitt 2.1.3.2). Außerdem unterstützt er in seiner bisherigen Fassung keine Temporallogik zur Eigenschaftsspezifikation.

Stattdessen müssen die zu prüfenden Eigenschaften als Invarianten formuliert werden. Damit sind nur Erreichbarkeits- und einige Sicherheitseigenschaften abprüfbar (siehe Abschnitt 2.1.2). Zur Verifikation von Lebendigkeits- und Fairnesseigenschaften muss auf andere

Werkzeuge zurückgegriffen werden. Die Invarianten werden als Action ohne Parameter und ohne Anweisungsblock definiert. Außerdem ist im MPC festgelegt, dass der Bezeichner der entsprechenden Action mit „Invariant“ beginnen muss.

Neben dem MPC existieren weitere Werkzeuge zur Verarbeitung von Model Programs (NModel [JVCS08], SpecExplorer [VCG⁺08]), die zwar keine Model Checker im klassischen Sinn sind, aber den Zustandsraum von Model Programs für einen begrenzten Wertebereich aufspannen können, so dass eine manuelle oder werkzeuggestützte Analyse möglich ist.

Für diesen Zweck empfiehlt sich eine andere Darstellung der Model Programs in Form von C#-Code, da darin auch komplexe Datentypen wie Klassen und Strukturen dargestellt werden können. Dies ist insbesondere bei der Betrachtung von Objektflüssen interessant.

Der nächste Abschnitt zeigt nun ausführlich, wie Model Programs zur Überprüfung von Aktivitätsdiagrammen erstellt werden. An verschiedenen Stellen wird auch kurz darauf eingegangen, was bei der Übertragung auf andere Eingabesprachen berücksichtigt werden müsste.

4.1.2 Zustandsübergänge in Model Programs

Die Umsetzung der Zustandsübergänge in Model Programs knüpft an die in Abschnitt 3.2.5 eingeführten abstrakten Makros an, die Ergebnisse der Domäne *ModelCheckerInput* zurückliefern. Im allgemeinen Fall kann diese Domäne auf Strings zurückgeführt werden, so dass als Eingabe für den Model Checker ein einfaches Textdokument erzeugt wird. Davon wird auch im Folgenden ausgegangen.

Für die einzelnen Makros werden in diesem Abschnitt mögliche Realisierungen in Model Programs diskutiert. Die Anforderungen an die einzelnen Makros werden textuell beschrieben. Diese Anforderungen können meist kurz gehalten werden, da die meisten Berechnungen bereits in den aufrufenden Makros (siehe Abschnitt 3.2.5) erledigt werden.

4.1.2.1 Definition der Zustandsvariablen

Verschiedene Modus- und Puffervariablen spiegeln den Systemzustand wider. Deren Umsetzung in die Eingabesprache des Model Checkers beeinflusst die in den weiteren Makros durchzuführenden Anweisungen zu deren Manipulation erheblich.

Daher werden im Folgenden gerade für die Definition der Variablen verschiedene Möglichkeiten diskutiert. Die gewählte Umsetzung hängt schließlich maßgeblich von den Möglichkeiten des gewählten Model Checkers bezüglich der Verarbeitung komplexer Datenstrukturen ab.

Tabelle 4.1 gibt einen Überblick über die Elemente, für die Variablen definiert werden müssen. In der zweiten Spalte wird angegeben, welchen Typ die jeweiligen Variablen haben können. Werden die in Klammern angegebenen Typen bei der Umsetzung verwendet, können nicht mehr alle Aspekte der Aktivitätsdiagramme dargestellt werden. Die jeweiligen Auswirkungen werden in der nachfolgenden ausführlichen Erläuterung genauer beschrieben.

Die letzte Spalte zählt die Elemente bzw. Makros auf, deren Umsetzung von der gewählten Variablendarstellung abhängt. Werden die verschiedenen Modi z. B. als Aufzählung realisiert, müssen auch die Variablen der Aktionen von diesem Typ sein. Dies wiederum hat Auswirkungen auf das Makro `CREATEACTIONTRANSITION`, welches Code für die Manipulation dieser Modi erzeugt.

Das Makro `CREATEPREAMBLE` (siehe Anhang A.3) ruft folgende Submakros auf, die die notwendigen Variablendefinitionen durchführen:

Tabelle 4.1: Mögliche Umsetzungen der Zustandsvariablen in Model Programs.

Element	mögliche Darstellungen	Auswirkungen auf
Modus	Aufzählung, Integer	Aktion, AEABufferVariable
Klassenattribute	(entsprechend des Typs und den Möglichkeiten des Model Checkers)	CREATECODE, CREATEGUARDPRECONDITION
Aktion, AEABufferVariable	Variable vom Typ „Modus“	CREATEMODEPRECONDITION, CREATEAEAMODEPRECONDITION, CREATEACTIONTRANSITION, CREATEAEABUFFERTRANSITION, CREATEIFMODEFORNODES
ObjectNode	Liste, Array + Zähler, (Menge, einfacher Typ)	CREATELISTPRECONDITION, CREATEOUTPUTPRECONDITION, CREATEBOUNDPRECONDITION, CREATEDATAFLOW, CREATEDA- TAFLOWWITHOUTREMOVE, CREATEDATAREMOVE, CREATEEMPTYLIST, CREATENULLDATAADD
Signal	Struktur, (ohne Daten auch: Integer, Aufzählung)	CREATE SIGNALDATAFROM, CREATE SIGNALDATA TO
Signalpuffer	Liste, (Multimenge, Menge, einfache Variable)	CREATE SIGNALBUFFERCONCAT, CREATE SIGNALBUFFERREPLACE, CREATE SIGNALBUFFERREMOVE, CREATE SIGNALBUFFERREMOVE- ALL

DEFINEMODES : *void* → *ModelCheckerInput*

DEFINEATTRIBUTE : *Attribute* → *ModelCheckerInput*

DEFINENODEVARIABLE : *Node* × *NodeMode* → *ModelCheckerInput*

DEFINEAEABUFFERVARIABLE : *Node* × *NodeMode* → *ModelCheckerInput*

DEFINELIST : *ObjectNode* → *ModelCheckerInput*

DEFINESIGNAL : *Signal* → *ModelCheckerInput*

DEFINESIGNALLIST : *Activity* → *ModelCheckerInput*

Diese Makros werden im Folgenden jeweils ausführlicher beschrieben und dabei auf die Auswirkungen der unterschiedlichen Variablendefinitionen eingegangen.

Zunächst werden über das Makro **DEFINEMODES** die möglichen Modi definiert. Für Model Programs empfiehlt sich eine Umsetzung als Aufzählung, allerdings könnten auch Konstanten feste Integerwerte zugewiesen werden (tatsächlich werden Aufzählungen intern so umgesetzt). Als Ergebnis des Makros wird für Model Programs also folgender Text zurückgeliefert:

```
enum Modes
  inactive
```

```
running
offeringToken
```

Das Makro `DEFINEATTRIBUTE` bekommt ein Attribut einer Klasse übergeben und erstellt daraus eine Variable mit einem eindeutigen Variablennamen und dem entsprechenden Typ.

Die meisten Model Checker unterstützen nur ganzzahlige Werte, Boolean und in beschränktem Umfang rationale Werte. Insbesondere Zeichenketten und zusammengesetzte Datenstrukturen werden nicht unterstützt. Daher ist die Verwendung von Klassenattributen (und auch sonstigen Datenflüssen) nur sehr eingeschränkt möglich. Die Unterstützung von Mengen und Multimengen des MPC hilft nur wenig, da die meisten Programmiersprachen wiederum diese Datentypen nicht nativ unterstützen.

Im nächsten Schritt wird für jede Aktion bzw. für jeden *InitialNode* der Aktivitätsdiagramme eine Variable definiert, die den Modus des Knotens widerspiegelt. Dem Makro `DEFINENODEVARIABLE` wird zusätzlich der Initialzustand der Variablen übergeben, so dass beispielsweise folgender Code erzeugt wird:

```
var AktionA as Modes = inactive
```

Die Variablenbezeichnung muss selbstverständlich eindeutig gewählt sein und sollte sich aus dem übergebenen Knoten ableiten lassen. Anderenfalls muss eine interne Tabelle geführt werden, die Elemente des Aktivitätsdiagramms auf Model Checker Variablen abbildet.

Wie auf Seite 67 erläutert, wird für *AcceptEventActions* eine weitere Variable zum Puffern des Modus der Aktion definiert. Obwohl das Ergebnis dem Makro `DEFINENODEVARIABLE` im Prinzip gleich ist, wurde dennoch ein eigenes Makro `DEFINEAEABUFFERVARIABLE` eingeführt, um eine Unterscheidung der Variablennamen zu ermöglichen. Dies gilt auch für die weiter unten beschriebenen Makros `CREATEAEAMODEPRECONDITION` und `CREATEAEABUFFERTRANSITION`.

Die Variablen für *ObjectNodes* (`DEFINELIST`) sollten mit einem Listentyp definiert sein, damit die eingehenden Tokens in einer Reihenfolge gehalten und nach und nach entnommen werden können. Die meisten Model Checker und auch der MPC unterstützen allerdings keine Listen. Zur Lösung des Problems gibt es zwei Möglichkeiten:

Entweder wird der Listentyp mit den vorhandenen Möglichkeiten nachgebildet. Dies ist im Fall von Model Programs beispielsweise mit Arrays und zusätzlichen Zählern für Anfangs- und Endposition möglich.

Eine zweite Möglichkeit wäre, auf die Unterstützung von Lower- und Upper-Bounds jenseits des Standardwerts eins (siehe Seite 62) zu verzichten und die Puffer durch eine einzige Variable abzubilden. Dazu müsste dann noch ein bestimmter Wert definiert werden, der anzeigt, dass der „Puffer“ leer ist.

Die Definition von Signalen, die über das Makro `DEFINESIGNAL` durchgeführt wird, kann im Falle von Signalen ohne Dateninhalt wieder als Aufzählung bzw. Konstante erfolgen.

Falls mit dem Signal Daten transportiert werden sollen, müssen zusätzliche Variablen definiert werden, die einerseits die Daten aufnehmen und andererseits dafür sorgen, dass die Signale unterschieden werden können. Wobei dies wiederum nur dann der Fall ist, wenn Datensignale gepuffert werden sollen. Falls Signale direkt verarbeitet werden, wird im Makro `CREATECOMBINATIONSFORHANDLINGSIGNALLMEDIATELY` der Datenfluss direkt von den *InputPins* der *SendSignalAction* auf die *OutputPins* der *AcceptEventActions* vollzogen.

Auch für die Umsetzung von Signalpuffern (`DEFINESIGNALLIST`) müssen auf Grund der eben erwähnten eingeschränkten Möglichkeiten von Model Checkern, insbesondere in Bezug auf Listen, Abwägungen getroffen werden.

Ist über den semantischen Variationspunkt **SignalBuffering** (siehe Abschnitt 2.3.2) für alle internen Signale festgelegt, dass im Puffer vorhandene Signale mit neu eintretenden ersetzt werden, lassen sich die Puffer pro Aktivität in Model Programs als Mengen darstellen.

Ist dies nicht der Fall, können die Signalpuffer in Model Programs als Multimengen (Bags) abgebildet werden. Allerdings ist die Verifikation von Multimengen im MPC nur vollständig für Integer Multimengen (siehe [VSB08]) und daher auch nur eingeschränkt einsetzbar.

Falls Multimengen vom gewählten Model Checker nicht unterstützt werden, könnten diese simuliert werden. Dazu wird pro Signal eine Variable eingesetzt, die jeweils die Anzahl der Vorkommen des Signals im Puffer angibt. Alle diese Variablen zusammen repräsentieren dann den Puffer. Allerdings lassen sich damit Signale, die Daten transportieren, wiederum nur schwer umsetzen.

4.1.2.2 Zustandsübergänge

Nach der Definition der Daten in der Präambel, werden an verschiedenen Stellen der Makros, die die Zustandsübergänge erzeugen (Abschnitt 3.2.5), immer wieder Makros aufgerufen, die einzelne Abfragen oder Anweisungen in der Eingabesprache für den Model Checker erzeugen. Diese lassen sich in folgende fünf Gruppen einordnen, die im Folgenden näher erläutert werden:

1. Vorbedingungen
2. Modusmanipulation
3. Datenflüsse
4. Signale
5. Vermischtes

Vorbedingungen

Für verschiedene Aspekte müssen Vorbedingungen für Zustandsübergänge generiert werden. Dies schließt neben den Abfragen, ob sich Aktionen in einem bestimmten Modus befinden, auch Abfragen über Puffergrößen, «Guards» und Signalpuffer mit ein. Diese Abfragen werden von folgenden Makros generiert:

CREATEMODEPRECONDITION : *ActivityNode* × *NodeMode* → *ModelCheckerInput*
CREATEAEAMODEPRECONDITION : *AcceptEventAction* × *NodeMode* → *ModelCheckerInput*
CREATELISTPRECONDITION : *ObjectNode* × *Boolean* → *ModelCheckerInput*
CREATEGUARDPRECONDITION : *GuardSpec* → *ModelCheckerInput*
CREATE SIGNALBUFFERPRECONDITION : *Activity* × *Signal* → *ModelCheckerInput*
CREATEOUTPUTPRECONDITION : *Action* → *ModelCheckerInput*
CREATEBOUNDPRECONDITION : *ObjectNode* × *Nat* → *ModelCheckerInput*

Die Makros **CREATEMODEPRECONDITION** und **CREATEAEAMODEPRECONDITION** erstellen für den übergebenen *ActivityNode* eine Vorbedingung, ob sich die entsprechende Variable im ebenfalls übergebenen Modus befindet. Für Model Programs sieht das Ergebnis beispielsweise folgendermaßen aus:

```
require AktionA = offeringToken
```

In Abhängigkeit von der Realisierung von Datenlisten an *ObjectNodes* (siehe oben), erstellt das Makro **CREATELISTPRECONDITION** eine Abfrage auf eine leere Liste oder auf eine bestimmte Variable oder eine Abfrage, die mehrere Variablen mit einschließt (falls eine Liste mit Arrays simuliert wird). Der zweite boolesche Parameter gibt an, ob eine Überprüfung auf eine leere (**true**) oder eine nicht-leere Liste (**false**) erstellt werden soll. Für eine Realisierung mit einfachen Variablen würde eine Abfrage auf einen leeren Puffer lauten:

```
require OutputPin_o1 = EmptyBufferConst
```

CREATEGUARDPRECONDITION übernimmt die Übersetzung der *«Guards»* in eine für den Model Checker verständliche Formel. Dabei müssen auch Klassenattributzugriffe auf die entsprechend definierten Variablen abgebildet werden. Wie die Generierung der Datenlistenabfragen hängen auch die Abfragen bezüglich der Signalpuffer von der gewählten Implementierung ab.

Wenn sich das übergebene Signal im Signalpuffer der angegebenen Aktivität befindet, sind die vom Makro **CREATE SIGNAL BUFFER PRECONDITION** erstellten Bedingungen wahr. **CREATE OUTPUT PRECONDITION** überprüft, ob in den *OutputPins* der übergebenen Aktion für Ergebnisse Platz ist.

Eine ähnliche Funktion nimmt das Makro **CREATE BOUND PRECONDITION** ein, welches Bedingungen generiert, die überprüfen, ob in den übergebenen *ObjectNodes* noch so viel Platz ist, wie im zweiten Parameter angegeben.

Modusmanipulation

Wenn alle Voraussetzungen für einen Zustandsübergang erfüllt sind, kann der eigentliche Zustandsübergang erfolgen. Dies erfordert unter anderem das Setzen des neuen Modus für betroffene Aktionen. Diese Aufgabe wird von folgenden Makros erfüllt:

CREATE ACTION TRANSITION : *ActivityNode* × *NodeMode* → *ModelCheckerInput*

CREATE AEABUFFERTRANSITION : *AcceptEventAction* × *NodeMode* → *ModelCheckerInput*

Diese beiden Makros erstellen jeweils für die entsprechenden Variablen eine Zuweisung des übergebenen Modus. Das Ergebnis in Model Programs sieht beispielsweise folgendermaßen aus:

```
AktionA := running
```

Datenflüsse

Neben der Modusanpassung beinhalten Zustandsänderungen auch die Durchführung der entsprechenden Datenflüsse. Allgemein werden Datenflüsse realisiert, indem Daten aus einem Puffer entfernt und an einen anderen angehängt werden. Diese Makros behandeln die verschiedenen Zuweisungen für Datenflüsse:

CREATE DATA FLOW : *ObjectNode* × *ObjectNode* → *ModelCheckerInput*

CREATE DATA FLOW WITHOUT REMOVE : *ObjectNode* × *ObjectNode* → *ModelCheckerInput*

CREATE DATA REMOVE : *ObjectNode* → *ModelCheckerInput*

CREATE EMPTY LIST : *ObjectNode* → *ModelCheckerInput*

CREATE NULL DATA ADD : *ObjectNode* → *ModelCheckerInput*

Das Makro `CREATEDATAFLOW` erstellt Eingabecode für den Model Checker, der genau das eben spezifizierte Verhalten realisiert: Es werden Daten aus dem Datenpuffer des einen *ObjectNodes* entfernt und an den Datenpuffer des zweiten übergebenen *ObjectNodes* angehängt. Falls ein Datum an mehrere Ziele kopiert werden soll, muss das Makro `CREATEDATAFLOWWITHOUTREMOVE` in Kombination mit `CREATEDATAREMOVE` verwendet werden.

Schließlich gibt es Fälle, in denen ein Puffer komplett geleert werden soll (`CREATEEMPTYLIST`) bzw. ein leeres Datentoken erstellt werden muss (`CREATENULLDATAADD`). Letzteres tritt z.B. an *ForkNodes* auf, wenn eine ausgehende Kante im Gegensatz zur eingehenden Kante eine reine Kontrollkante ist.

Signale

Auch für die Manipulation der Signalpuffer sind Makros definiert, um von der eigentlichen Umsetzung zu abstrahieren:

`CREATE SIGNAL BUFFER CONCAT` : $Activity \times Signal \rightarrow ModelCheckerInput$
`CREATE SIGNAL BUFFER REPLACE` : $Activity \times Signal \rightarrow ModelCheckerInput$
`CREATE SIGNAL BUFFER REMOVE` : $Activity \times Signal \rightarrow ModelCheckerInput$
`CREATE SIGNAL BUFFER REMOVE ALL` : $Activity \rightarrow ModelCheckerInput$
`CREATE SIGNAL DATA FROM` : $SendSignalAction \rightarrow ModelCheckerInput$
`CREATE SIGNAL DATA TO` : $Signal \times AcceptEventAction \rightarrow ModelCheckerInput$

Entweder wird ein Signal an einen Puffer angehängt oder es ersetzt ein bereits vorhandenes Signal (`CREATE SIGNAL BUFFER CONCAT`, `CREATE SIGNAL BUFFER REPLACE`). Wenn ein Signal von einer *AcceptEventAction* empfangen wurde, muss es aus dem Puffer wieder entfernt werden (`CREATE SIGNAL BUFFER REMOVE`). Der Puffer wird komplett geleert, wenn die Aktivität beendet wird (`CREATE SIGNAL BUFFER REMOVE ALL`).

Die beiden verbleibenden Makros stellen eigentlich Datenflüsse dar. Der damit erstellte Code muss sicherstellen, dass die Daten aus dem *InputPin* der angegebenen *SendSignalAction* in das Datensignal überführt werden (`CREATE SIGNAL DATA FROM`) oder vom Signal in den *OutputPin* der empfangenden *AcceptEventAction* geschrieben wird (`CREATE SIGNAL DATA TO`). Die konkrete Ausgestaltung all dieser Makros hängt von der für die Signalpuffer gewählten Realisierung ab und wird daher hier nicht näher ausgeführt.

Verschiedenes

Zum Abschluss werden noch einige spezielle Makros behandelt, die für verschiedene Aspekte zuständig sind.

`CREATE STEP HEADER` : $void \rightarrow ModelCheckerInput$
`CREATE CODE` : $Code \rightarrow ModelCheckerInput^*$
`CREATE IF MODE FOR NODES` : $\mathcal{P}(ActivityNode) \times NodeMode \rightarrow ModelCheckerInput^*$
`CREATE ELSE` : $ModelCheckerInput^* \rightarrow ModelCheckerInput$

`CREATE STEP HEADER` wird immer dann aufgerufen, wenn ein neuer Zustandsübergang begonnen wird. Die Realisierung in Model Programs entspricht der Definition einer Action:

```
[Action]
Action1()
```

Im Prinzip ist ein generischer Name für die Action ausreichend, da, falls der Model Checker ein Gegenbeispiel findet, für jeden Schritt der Inhalt aller Zustandsvariablen ausgelesen werden kann und damit die Reihenfolge der Modi der einzelnen Aktionen rekonstruiert werden kann. Für eine komfortable Auswertung kann es jedoch hilfreich sein, den diesem Zustandsübergang entsprechenden Schritt im Namen der Action zu codieren. Darüberhinaus können dem Makro auch entsprechende Parameter übergeben werden.

Das Makro `CREATECODE` übersetzt das in einer *CallOperationAction* spezifizierte Verhalten in eine für den Model Checker verarbeitbare Eingabe. Dabei muss natürlich wieder auf die Möglichkeiten des Model Checkers Rücksicht genommen werden. Auf alle Fälle übernimmt der damit erzeugte Code die Entnahme der Daten aus den *InputPins* und die Erzeugung der Daten für die *OutputPins*.

Falls an dieser Stelle beliebiger Code erlaubt ist, ist die Übersetzung in die Syntax und Semantik des verwendeten Model Checkers sehr aufwändig. Man könnte aber der Einfachheit halber fordern, dass nur entsprechend kompatibler Code benutzt werden darf. Dieser wird dann lediglich in die Eingabe des Model Checkers kopiert.

Die letzten beiden Makros werden gemeinsam verwendet. `CREATEIFMODEFORNODES` erstellt ein `if`-Konstrukt mit einer Konjunktion der Modusabfragen für alle übergebenen Knoten. Dabei muss dieses Makro selbständig unterscheiden, ob es sich um eine Aktion oder um einen *ObjectNode* handelt und entsprechenden Code erzeugen.

Dieses Makro wird eingesetzt, um im Makro `CREATEMUSTSTAYACTIVECHECK` die Abfrage zu generieren, ob eine *AcceptEventAction* im Modus `running` verbleiben soll oder nicht. Dazu passend wird das Makro `CREATEELSE` verwendet, welches die übergebene Model Checker Eingabe in einen `else`-Zweig setzt.

Mit diesen textuellen Anforderungen an die abstrakten Makros, ist der Übersetzungsprozess von Aktivitätsdiagrammen in Zustandsübergangsdigramme beendet. Abschließend sei vermerkt, dass sehr viel Komplexität in der Übersetzung durch die Berücksichtigung der Datenflüsse eingebracht wird (siehe dazu auch Abschnitt 4.3). Darüber hinaus können speziellere Aspekte wie lower- und upper-Bounds mit den meisten Model Checkern nicht überprüft werden und die Möglichkeiten der Datenmanipulation sind sehr beschränkt.

4.1.3 Fallbeispiele

Leider sind bisher keine größeren Systeme mit Aktivitätsdiagrammen modelliert worden. Dies liegt einerseits an der nach wie vor mangelhaften Werkzeugunterstützung für diesen Formalismus, andererseits an der fehlenden einheitlichen Semantik. Der Vorschlag aus [Sar06a] ist zwar die umfassendste Arbeit in diese Richtung, allerdings haben die Ergebnisse noch keinen Einzug in kommerzielle Werkzeuge gehalten.

Dementsprechend sind vorhandene Beispiele meist inkompatibel mit der in dieser Arbeit verwendeten Semantik bzw. basieren auf gar keiner. Dennoch werden im Folgenden zwei Fallbeispiele vorgestellt, die die Funktionsweise und den Nutzen der Verifikation darstellen.

Das erste Beispiel basiert auf einem im SysML-Tutorial [FMS08] veröffentlichten Beispiel und zeigt wie Fehler im Diagramm durch Model Checking gefunden werden können. Das zweite Beispiel beinhaltet einen Fehler der auch durch Testen nicht ohne weiteres gefunden werden würde. Model Checking gegen die gewünschte Sicherheitseigenschaft liefert aber ein Gegenbeispiel welches diese Eigenschaft verletzt.

4.1.3.1 SysML Beispiel

Dieses Fallbeispiel basiert auf einem Aktivitätsdiagramm, welches im Tutorial zur Beschreibung der SysML als Beispiel [FMS08, Seite 109] aufgeführt ist. Das Beispiel beschreibt in einer Mischung aus Workflow- und Systembeschreibung den Kontext und die Funktionsweise einer Alarmanlage. Abb. 4.2 zeigt das Diagramm. In der mittleren Spalte wird das Verhalten des „Extended Security Systems“ dargestellt, links das des Einbrechers und rechts der externen Dienste.

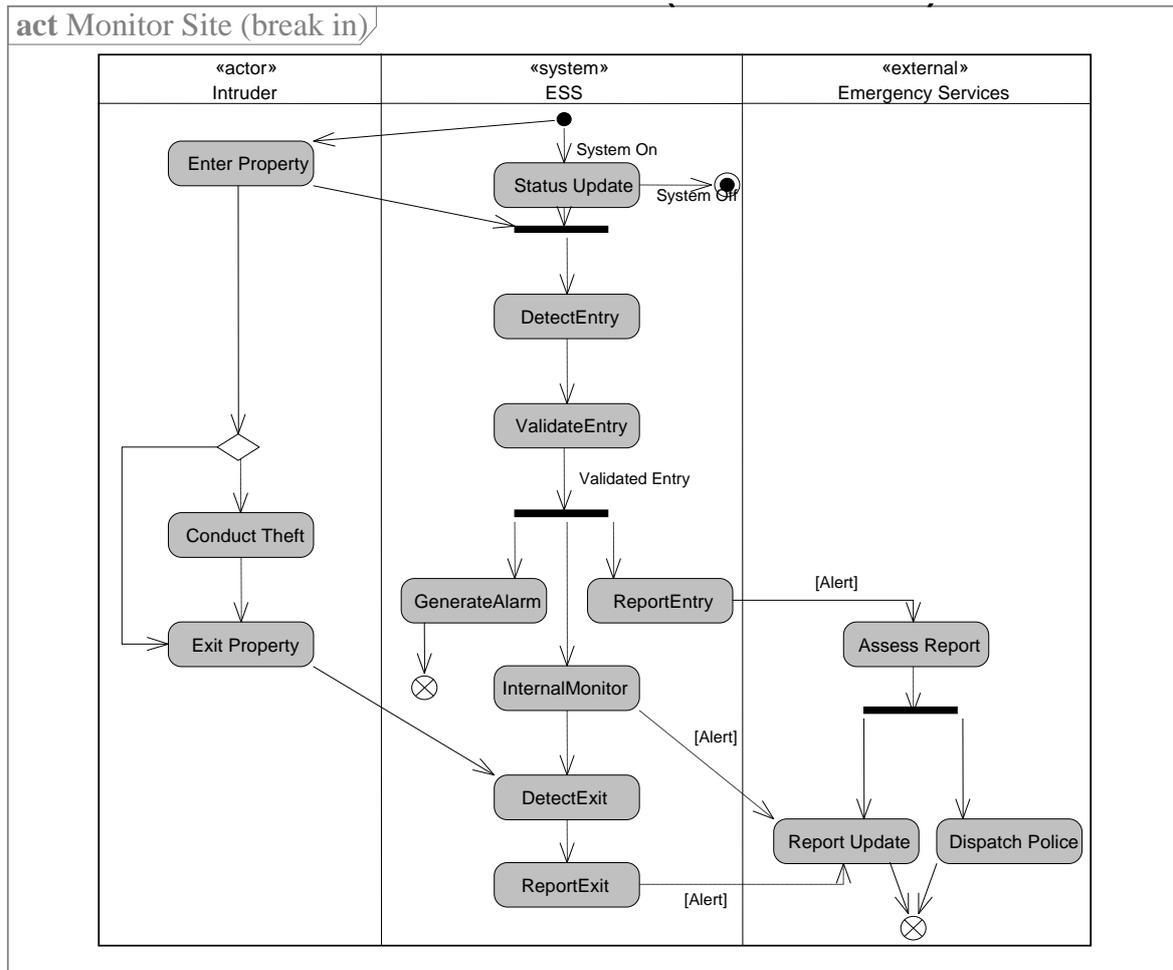


Abbildung 4.2: Aktivitätsdiagramm einer Alarmanlage (entnommen aus [FMS08]).

Nach dem Start wird — vorausgesetzt, dass System ist eingeschaltet — der aktuelle Status gesetzt. Sobald ein Einbrecher das Anwesen betritt (**Enter Property**), wird der Einbruch detektiert (**DetectEntry**). Daraufhin wird ein Alarm generiert (**GenerateAlarm**), die interne Überwachung informiert (**InternalMonitor**) und eine Benachrichtigung an die Notfalldienste erzeugt (**ReportEntry**).

Diese Benachrichtigung wird einerseits an die Polizei weitergeleitet (**Dispatch Police**) und andererseits ein interner Bericht aktualisiert (**ReportUpdate**). Sobald der Einbrecher das

Gebäude verlassen hat (Exit Property), wird auch dieses Ereignis registriert (DetectExit) und an den Notdienst übermittelt.

Verifikationsbeispiele. Eine zu überprüfende Eigenschaft wäre, ob bei einem Einbruch überhaupt die Polizei informiert wird. Da der in der Implementierung verwendete Model Program Checker nur Invarianten verarbeitet, wird diese Eigenschaft folgendermaßen formuliert: `require not(DispatchPolice = running)`. Es soll also keinen Zustand geben, indem sich die Aktion DispatchPolice im Modus running befindet.

Die in dieser Arbeit vorgestellte Flussberechnung bestimmt 37 kombinierte Flüsse und daher wird zusammen mit den Zustandsübergängen pro Aktion ein Zustandsübergangssystem mit 51 Zustandsübergängen generiert. Der Bounded Model Program Checker liefert in wenigen Sekunden ein Gegenbeispiel mit zwölf Schritten zurück, wobei bei der folgenden verkürzten Darstellung die Zwischenzustände nicht dargestellt werden.

Counterexample found for 12 steps:

STEP 1: InitialNode192EnterProperty_StatusUpdate()

```
ExitProperty1 = inactive
EnterProperty1 = running
StatusUpdate1 = running
ConductTheft1 = inactive
GenerateAlarm1 = inactive
ValidateEntry1 = inactive
DetectEntry1 = inactive
InitialNode191 = inactive
ForkNode33a1 = inactive
ForkNode33b1 = inactive
ForkNode33c1 = inactive
InternalMonitor1 = inactive
ReportEntry1 = inactive
AssessReport1 = inactive
ForkNode37a1 = inactive
ForkNode37b1 = inactive
DetectExit1 = inactive
ReportExit1 = inactive
ReportUpdate1 = inactive
DispatchPolice1 = inactive
EnterProperty__Forka1 = inactive
EnterProperty__Forkb1 = inactive
StatusUpdate__Forka1 = inactive
StatusUpdate__Forkb1 = inactive
InternalMonitor__Forka1 = inactive
InternalMonitor__Forkb1 = inactive
InitialNode19__Forka1 = inactive
InitialNode19__Forkb1 = inactive
SystemOff1 = False
SignalsBuffer1 = {}
```

```

STEP 2:  StatusUpdate_Running2Offering()
STEP 3:  EnterProperty_Running2Offering()
STEP 4:  EnterProperty_StatusUpdate2ConductTheft_DetectEntry()
STEP 5:  DetectEntry_Running2Offering()
STEP 6:  DetectEntry2ValidateEntry()
STEP 7:  ValidateEntry_Running2Offering()
STEP 8:  ValidateEntry2ReportEntry_GenerateAlarm_InternalMonitor()
STEP 9:  ReportEntry_Running2Offering()
STEP 10: ReportEntry2AssessReport()
STEP 11: AssessReport_Running2Offering()
STEP 12: AssessReport2DispatchPolice_ForkNode37b()

```

Wie man sieht, lässt sich das Gegenbeispiel einerseits auf Grund der Modi der Aktionen, andererseits durch die Benennung der Schritte auf das ursprüngliche Aktivitätsdiagramm abbilden. Im ersten Schritt werden die Aktionen `EnterProperty` und `StatusUpdate` auf `running` gesetzt.

Schritt 2 und 3 beenden diese Aktionen jeweils und daher starten in Schritt 4 die Aktionen `ConductTheft` und `DetectEntry`. Dies kann man nun entlang der Kette verfolgen, bis im letzten Schritt einerseits die Polizei benachrichtigt wird und andererseits ein Token an der linken ausgehenden Kante des *ForkNodes* gepuffert wird, da die Aktion `ReportUpdate` nicht starten kann.

Eine etwas interessantere Frage ist nun, ob auch die Aktion `Report Update` überhaupt starten kann (`require not(ReportUpdate = running)`). Ein Model Checker Durchlauf ergibt, dass auch für 30 Schritte kein Gegenbeispiel gefunden werden kann. Weitere Versuche mit unterschiedlichen Aktionen ergeben, dass schon die Aktion `Exit Property` nie gestartet werden kann.

Dies ist bei näherer Betrachtung nicht weiter verwunderlich, da die beiden eingehenden Kanten an dieser Aktion wie ein implizites `Join` wirken. Allerdings kann auf Grund des davor liegenden *DecisionNodes* nur eine Kante ein Token erhalten. Es ist also wahrscheinlich, dass an dieser Stelle ein *MergeNode* zwischen den Aktionen `Conduct Theft` und `Exit Property` vergessen wurde.

Wird dieser Fehler behoben, findet der Model Checker auch nach 18 Schritten einen Beispielablauf, bei dem die Aktion `Report Update` startet. Fraglich ist, ob die Aktion `Report Update` tatsächlich nur ausgeführt werden soll, wenn an allen drei Eingängen Tokens anliegen. Es ist sehr wahrscheinlich, dass auch hier eine *MergeNode* Semantik intendiert war.

Fazit. Die gefundenen Fehler sind zwar nur kleine Fehler, die eventuell auch durch andere Prüfverfahren gefunden werden können. Allerdings zeigt dieses Beispiel, wie das Model Checking von Aktivitätsdiagrammen prinzipiell funktioniert und dass auch mit der Überprüfung sehr einfacher Eigenschaften, Fehler gefunden werden können.

Interessanterweise wurden in fast allen untersuchten Beispieldiagrammen Fehler gefunden, die durch die Auswirkung impliziter *JoinNodes* verursacht werden. Obwohl in der UML-Spezifikation auf Seite 312 die Semantik von mehreren eingehenden Kanten erläutert wird, findet sich nur elf Seiten weiter in Abb. 12.36 ein diesbezüglich fehlerhaftes Beispiel.

Modell. Abb. 4.3 zeigt ein mögliches Modell der Steuerung für eine Zweihandpresse. Nach dem Start wird auf die beiden Tastendrücke gewartet. Sobald eine der beiden Tasten gedrückt wird, wird der Timer für eine Sekunde gestartet. Falls dieser Timer abläuft, wird der umliegende Unterbrechungsbereich verlassen und erneut auf die beiden Tastendrücke gewartet.

Wird der andere Knopf allerdings innerhalb einer Sekunde ebenfalls gedrückt, wird ebenfalls der Unterbrechungsbereich verlassen und damit der Timer abgebrochen. Durch das Betreten des nächsten Unterbrechungsbereichs werden (entsprechend des semantischen Variationspunkts) die beiden innerliegenden *AcceptEventActions* ebenfalls gestartet.

In der Aktion *MoveDownAbove* wird der Stempel nach unten bewegt bis er den Point-of-no-return erreicht hat. Anschließend wird dieses Signal versendet und von dem entsprechenden Signalempfänger registriert.

Durch das Verlassen des inneren Unterbrechungsbereichs wird auch die *AcceptEventAction AnyButtonReleased* abgebrochen, so dass fortan nicht mehr auf das Loslassen der Tasten reagiert wird. Sobald der Stempel ganz unten angekommen ist, fährt er wieder nach oben und eine neuer Durchlauf kann starten.

Wird während des Herunterfahrens eine Taste losgelassen, wird der gesamte untere Unterbrechungsbereich abgebrochen und der Stempel fährt sofort nach oben.

Verifikation. Die wichtigste Sicherheitseigenschaft dieser Steuerung lautet: „Es darf niemals sein, dass der Stempel nach unten fährt, obwohl nicht beide Tasten gedrückt sind.“

Auf Ebene des Aktivitätsdiagramms kann dies folgendermaßen formuliert werden: „*MoveDownAbove* darf nicht starten, wenn nicht beide Tasten gedrückt sind und *AnyButtonReleased* nicht aktiv ist.“ Der zweite Teil ist notwendig, denn falls *AnyButtonReleased* aktiv ist, dann wird auf das Lösen einer Taste entsprechend reagiert.

Die Frage ist nun, ob das oben vorgestellte Diagramm diese Eigenschaft erfüllt. Dies soll im Folgenden durch Verifikation herausgefunden werden.

Für die Spezifikation der zu prüfenden Eigenschaft ist es notwendig, den Status der einzelnen Tasten abfragen zu können. Die Umgebung muss in diesem Fall, wie im folgenden Abschnitt 4.2 beschrieben, teilweise modelliert werden, da z. B. das Loslassen einer Taste ein Ereignis der Umgebung ist.

Abb. 4.4 zeigt das angepasste Modell. Parallel zu dem bisherigen Modell werden die zusätzlichen externen Signale *LeftButtonReleased* und *RightButtonReleased* empfangen und eine Reihenfolge zwischen dem Drücken und dem Loslassen definiert.

Ein Ausschnitt des aus diesem Modell erzeugten Model Programs sieht folgendermaßen aus:

```
var InitialNode3 as Modes = offeringToken
var MoveUp as Modes = inactive
var MoveDownAbove as Modes = inactive
var Reached as Modes = inactive
var PointOfNoReturn as Modes = inactive
var AnyButtonReleased as Modes = inactive
var LeftButtonPressed as Modes = inactive
var RightButtonPressed as Modes = inactive
var _1s as Modes = inactive
var MoveDownBelow as Modes = inactive
var sendsignalPointOfNoReturn as Modes = inactive
```

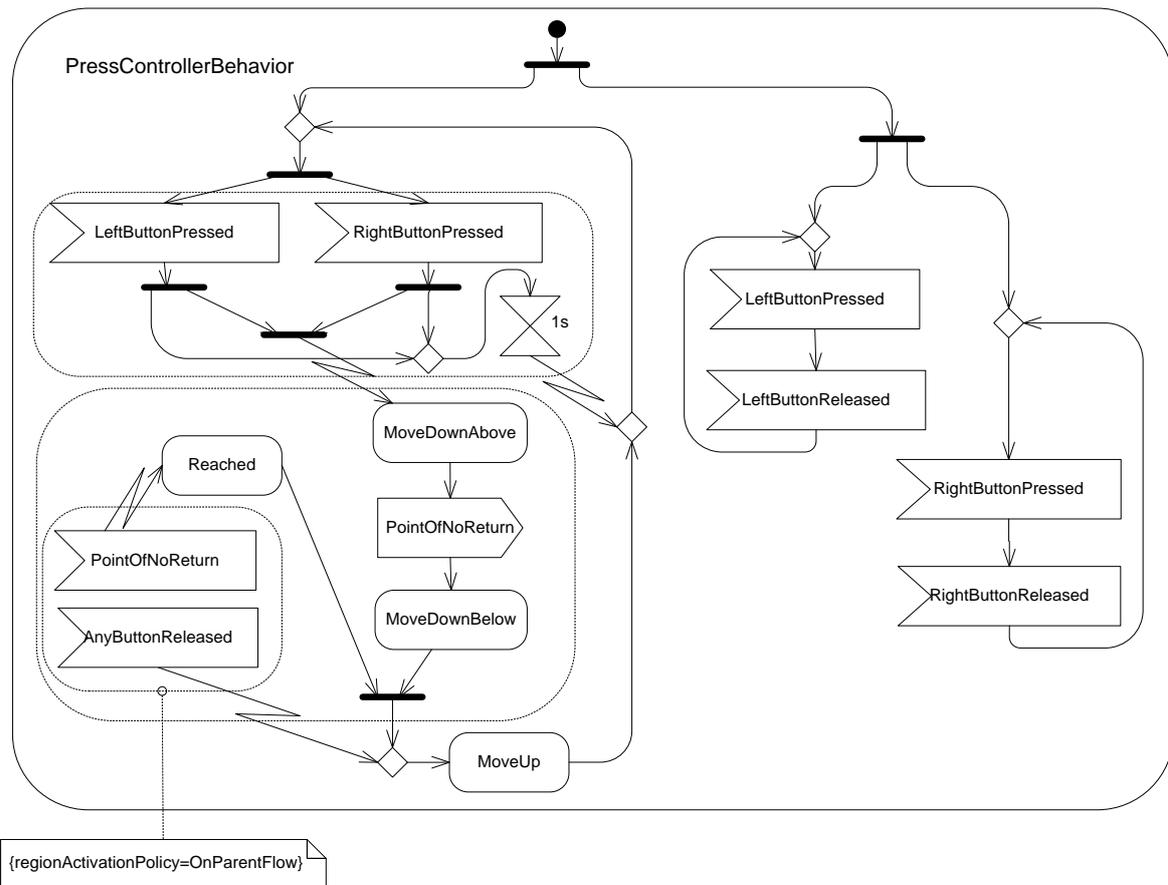


Abbildung 4.4: Aktivitätsdiagramm der Zweihandpressensteuerung.

[..]

enum Signals

```

    PointOfNoReturnSignal
    AnyButtonReleasedSignal
    LeftButtonPressedSignal
    RightButtonPressedSignal
    LeftButtonReleasedSignal
    RightButtonReleasedSignal

```

[..]

[Action]

```

PointOfNoReturn_Running2Offering(signal as Signals)
    require PointOfNoReturn = running
    require signal = PointOfNoReturnSignal
    require signal in SignalsQueue
    SignalsQueue := SignalsQueue - {PointOfNoReturnSignal}

```

```

    PointOfNoReturn := offeringToken

[.]

[Action]
Reached_MoveDownBelow2MoveUp()
    require Reached = offeringToken
    require MoveDownBelow = offeringToken

    Reached := inactive
    MoveDownBelow := inactive
    MoveUp := running

```

Die zu überprüfende Eigenschaft sieht als Invariante für den MPC folgendermaßen aus:

```

require not(
    (LeftButtonPressed=offeringToken and RightButtonPressed=offeringToken)
    and
    (LeftButtonReleased=offeringToken or RightButtonReleased=offeringToken)
    and AnyButtonReleased=inactive)

```

Der erste Teil beschreibt den Zustand vor dem Start von `MoveDownAbove`, der zweite Teil den unerwünschten Zustand, nämlich dass eine der beiden Tasten losgelassen wurde, obwohl `AnyButtonReleased` noch nicht aktiv ist.

Der MPC liefert in wenigen Sekunden ein Gegenbeispiel mit fünf Schritten:

```

Counterexample found for 5 steps:
STEP 1: InitialNode32RightButtonPressed_LeftButtonPressed_-
        LeftButtonPressedB_RightButtonPressedB()
STEP 2: LeftButtonPressed_LeftButtonPressedB_-
        Running2Offering(signal=LeftButtonPressedSignal)
STEP 3: LeftButtonPressedB2LeftButtonReleased()
STEP 4: LeftButtonReleased_-
        Running2Offering(signal=LeftButtonReleasedSignal)
STEP 5: RightButtonPressed_RightButtonPressedB_-
        Running2Offering(signal=RightButtonPressedSignal)

```

Der unerwünschte Zustand tritt also genau dann auf, wenn eine Taste gedrückt (Step 2) und vor dem zweiten Tastendruck (Step 5) innerhalb einer Sekunde wieder losgelassen (Step 3 und 4) wird.

Dieser Fehler tritt auf, da, während der erste Unterbrechungsbereich aktiv ist, noch nicht auf das Loslassen einer Taste reagiert wird. Die entsprechende *AcceptEventAction* (`AnyButtonReleased`) wird erst bei Betreten des zweiten Unterbrechungsbereichs gestartet.

Eine Erweiterung des zweiten Unterbrechungsbereichs, um den ersten mit einzuschließen, würde das Problem lösen, da dann von Anfang an `AnyButtonReleased` aktiv wäre und ein Loslassen eines Knopfes zu einem Neustart führen würde.

Fazit. An diesem Beispiel sind, neben dem Erkennen eines nicht trivialen Fehlers, mehrere Aspekte bemerkenswert:

- Obwohl die Aktionen vollkommen unspezifiziert sind, konnte der Kontrollfluss dieser Aktion bereits überprüft und dabei sogar Fehler gefunden werden. Diese Eigenschaft stärkt den Einsatz von Aktivitätsdiagrammen und deren Überprüfung in frühen Projektphasen.
- Das Beispiel zeigt, dass häufig Umgebungswissen in das Modell miteinbezogen werden muss. Eine Möglichkeit ist, das Aktivitätsdiagramm direkt um die entsprechenden Elemente zu erweitern.
- Trotz der Nichtbeachtung der Zeitangabe am Timer, kann das Modell verifiziert und ein Fehler entdeckt werden. Aspekte, die direkt von dieser Zeitangabe abhängen würden, könnten allerdings nicht überprüft werden.

4.2 Umgebungsmodellierung

Aktivitätsdiagramme werden als Teil der UML vor allem zur Spezifikation von Anforderungen in frühen Projektphasen eingesetzt. Der am Institut für Programmiermethodik und Compilerbau der Universität Ulm entwickelte Ansatz zur Integration von Programmcode in die Aktionen von Aktivitätsdiagrammen ermöglicht einen Einsatz auch in späteren Phasen der Softwareentwicklung (siehe [SRKS05]).

Bei diesem Ansatz wird der in den Diagrammen bereits spezifizierte Kontroll- bzw. Datenfluss interpretiert und an den entsprechenden Stellen der eingefügte Code aufgerufen. Dies vermeidet einerseits die Implementierung des Ablaufcodes und andererseits Probleme bei der Synchronisation der Code- und der Diagrammdarstellung.

Diese Möglichkeiten haben allerdings zur Folge, dass die ursprüngliche Trennung des Software Engineerings in Analyse, Entwurf und Implementierung vermischt wird. Dadurch kommen zwangsläufig Lösungsdetails in das ursprünglich abstrakte Modell. Zave und Jackson fordern in [ZJ97] allerdings, dass Anforderungen *nur* Informationen über die Umgebung beinhalten sollen und insbesondere keine Implementierungsdetails.

Sie begründen diese Forderung damit, dass es praktisch keinen Unterschied macht, ob eine Anforderung aus Sicht der „Maschine“ oder aus Sicht der Umgebung beschrieben wird. Die Sichtweise aus der Umgebung heraus bietet allerdings die Möglichkeit, mehr zu erfassen als dies für die eingeschränkte Sichtweise der Maschine möglich ist. Insbesondere Aktionen, die nur in der Umgebung stattfinden, aber indirekt Auswirkungen auf das System haben, werden im klassischen Requirements Engineering häufig nicht erfasst.

Diese Argumentation trifft auch für das Model Checking von Aktivitätsdiagrammen zu. Viele zu prüfende Eigenschaften können nicht nur mit den Elementen der Systembeschreibung überprüft werden, da sie von externen Eingaben abhängen bzw. das Verhalten des Systems typischerweise vom Verhalten der Umgebung abhängt.

Model Checking sollte bezüglich der Umgebung abgeschlossen sein [GMP02], da es automatisch ohne Interaktion abläuft. Um aussagekräftigere Ergebnisse zu erhalten, ist es aus diesem Grund notwendig, dem Model Checker Informationen über das Verhalten der Umgebung zur Verfügung zu stellen.

Die Schnittstelle zur Umgebung ist für Aktivitätsdiagramme klar definiert, da diese mit der Umgebung nur über Signale kommunizieren können. Das Verhalten der Umgebung kann

somit also durch die Spezifikation einer Reihenfolge und/oder Häufigkeit externer Ereignisse (siehe Abschnitt 3.2.5.3) näher beschrieben werden.

Daneben kann die Einführung weiterer Ereignisse nötig sein. Das System der Zweihandpresse, welches in Abschnitt 4.1.3.2 näher beschrieben ist, macht beispielsweise davon Gebrauch. Die Zweihandpresse reagiert in einem gewissen Zustand nur auf Tastendrucke. Zur Beschreibung der gewünschten Eigenschaft ist es aber notwendig, auch das Loslassen der Tasten zu registrieren. Aus diesem Grund werden zusätzliche Signale eingeführt und das Verhalten des externen Tasters in das Aktivitätsdiagramm mit aufgenommen.

Die Integration der Umgebungsspezifikation in das zu untersuchende Aktivitätsdiagramm birgt aber die Gefahr, das Verhalten der ursprünglichen Aktivität unabsichtlich zu verändern. Besser wäre es, das Verhalten der Umgebung in einer zusätzlichen Aktivität zu beschreiben und entsprechend zu kennzeichnen. Dies erfordert allerdings eine Erweiterung der Aktivitätsdiagramme.

Eine andere Möglichkeit wäre, die möglichen Ereignisabfolgen in Sequenzdiagrammen zu spezifizieren. Diese Diagrammart der UML bietet die Möglichkeit, komplexe Ablauffolgen des Nachrichtenaustauschs mit verschiedenen Kommunikationspartnern zu beschreiben (siehe [UML, RQZ07]). Im Folgenden wird kurz dargestellt, wie diese Diagrammart prinzipiell für diesen Zweck eingesetzt werden könnte.

Sequenzdiagramme bestehen aus nebeneinander platzierten Objekten, die Kommunikationspartner repräsentieren. Jeder Kommunikationspartner hat eine gestrichelt gezeichnete Lebenslinie, die einen zeitlichen Ablauf von oben nach unten vorgibt. Nachrichten zwischen Kommunikationspartnern werden über Pfeile zwischen den jeweiligen Lebenslinien dargestellt und mit den Signalnamen beschriftet. Dabei ist auch „Selbstkommunikation“, also das Senden einer Nachricht an sich selbst, möglich. Eine nicht ausgefüllte Pfeilspitze symbolisiert dabei einen asynchronen Nachrichtenaustausch. Dies wird der häufigste Fall von Kommunikation zwischen der Umgebung und dem System sein, da die Umgebung typischerweise nicht auf eine Reaktion des Systems wartet.

Außerdem können Teile der Nachrichten in sogenannten Fragmenten gekapselt werden. Diesen Fragmenten kann über definierte Schlüsselwörter eine Interpretation zugewiesen werden. Zur Beschreibung der möglichen Schlüsselwörter wird auf [RQZ07] verwiesen.

Abb. 4.5 zeigt ein Sequenzdiagramm, welches zur Bestimmung einer Umgebung für das Model Checking verwendet werden kann. Die darin enthaltenen Signale sind in dem Fallbeispiel „Zweihandpresse“ aus Abschnitt 4.1.3.2 definiert. Die jeweiligen Signalkomplexe `LeftButtonPressed` und `LeftButtonReleased` bzw. `RightButtonPressed` und `RightButtonReleased` beschreiben jeweils das Drücken und Loslassen des entsprechenden Knopfs.

Für jede Taste wird das Signal `AnyButtonReleased` gesendet. Da das Aktivitätsdiagramm nur die `Pressed`-Signale und das `AnyButtonReleased`-Signal empfängt, werden die `Released`-Signale wieder an die Umgebung geschickt. Damit werden diese für die Umgebung internen Signale in eine zeitliche Beziehung zu den restlichen Signalen gesetzt.

Die in diesem Beispiel verwendeten Fragmente geben an, dass die gesamte Kommunikation beliebig oft wiederholt werden kann (`loop`) und die rechte und die linke Taste unabhängig voneinander bedient werden können (`par`, wie parallel). Mit diesem Sequenzdiagramm werden nun einerseits die `Release`-Signale definiert und andererseits alle Signale in eine Ordnung gebracht.

Die Information, dass ein `Released`-Signal immer erst *nach* einem `Pressed`-Signal auftreten darf, ist eine wichtige Information für das Model Checking, welche nicht im Aktivitätsdiagramm enthalten ist. Würde diese Information fehlen, würde der Model Checker Fehlerfälle

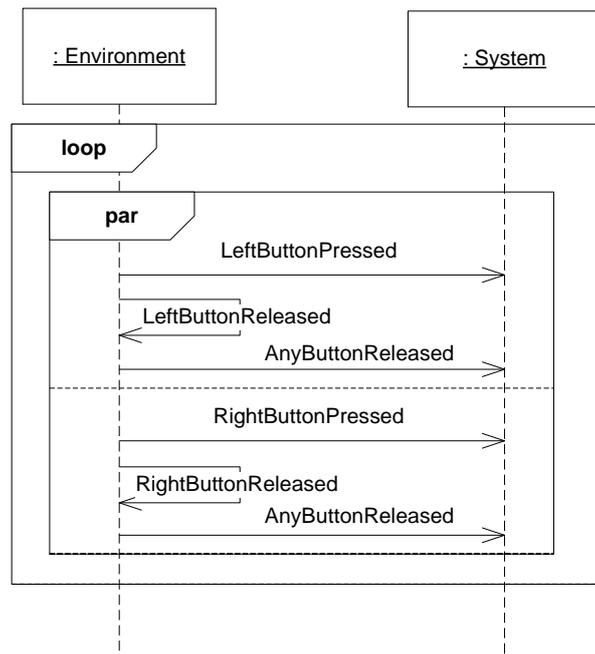


Abbildung 4.5: Sequenzdiagramm zur Umgebungsspezifikation.

zurückliefern, die auf Grund der gegebenen Signalkonstellation in der Realität nie auftreten können.

Damit diese Vorgaben während des Model Checkings tatsächlich berücksichtigt werden können, muss das Sequenzdiagramm eingelesen und entsprechend verarbeitet werden. Dabei muss eine Transformation des dargestellten Sachverhalts in Abhängigkeit der benutzten Fragmentarten (loop, par, etc.) z. B. in Zustandsübergänge des Model Checkers überführt werden, die sicherstellen, dass die spezifizierten Signale in der angegebenen Reihenfolge auftreten. Dies ist aber nicht Teil dieser Arbeit, sondern könnte in weiteren Arbeiten näher betrachtet werden.

4.3 Optimierungen

Wie in Abschnitt 2.1.3 bereits erwähnt, ist das Hauptproblem beim Model Checking der große Zustandsraum der meisten Systeme. Obwohl einige Techniken entwickelt wurden, den Zustandsraum zu verkleinern bzw. nicht komplett aufspannen zu müssen, ist es für effektives Model Checking enorm wichtig, das Modell schon im Vorfeld zu optimieren.

Zur Reduktion der Anzahl der Zustände kommen Abstraktionstechniken zum Einsatz. Eine häufig eingesetzte Methode untersucht, von welchen Zustandsvariablen die zu prüfende Spezifikation abhängt (*cone of influence reduction* [CGP99]). Alle anderen Variablen werden eliminiert, um ein kleineres Modell zu erhalten.

Eine weitere Methode ist die Datenabstraktion. Dabei werden die Wertebereiche der Variablen auf eine kleine Teilmenge (Äquivalenzklassen) reduziert, ohne die Aussagekraft zu verringern. Ein Beispiel wäre die Reduktion von Integervariablen auf die drei Wertebereiche „größer Null“, „kleiner Null“ und „gleich Null“. Weitere allgemeine Methoden zur Abstraktion finden sich in [CGP99].

Abstraktionstechniken werden typischerweise auf der obersten, abstraktesten Modellebene eingesetzt. Es kann zwischen einer automatisch und einer durch den Benutzer durchgeführten Abstraktion unterschieden werden. Durch tieferes Kontextwissen ist die Eingabe von Hinweisen durch den Benutzer häufig ein effektives Mittel, das Model Checking Problem erheblich einzuschränken.

Nach einer Betrachtung, welche Komponenten eines Aktivitätsdiagramms welchen Einfluss auf die Größe des Zustandsübergangssystems haben, werden für die Aktivitätsdiagramm- und die Übersetzungsebene verschiedene Optimierungen diskutiert.

4.3.1 Größenbetrachtungen

Um dem Problem des großen Zustandsraums adäquat begegnen zu können, muss zunächst abgeschätzt werden, wie groß ein Zustandsübergangssystem für ein Aktivitätsdiagramm wird. Auch wenn dieser Wert keine Aussage über die Größe des Zustandsraums machen kann, liefert die Betrachtung doch wertvolle Hinweise, wie sich ein Kontrollknoten im Aktivitätsdiagramm auf die Menge der Zustandsübergänge auswirkt.

Die Anzahl der Flüsse lässt einen Rückschluss auf die Anzahl der Zustandsübergänge zu, da pro Fluss ein Zustandsübergang erzeugt wird. Die Anzahl der einfachen Kontrollflüsse wird berechnet, indem zunächst folgende Werte bestimmt werden:

- AA = Anzahl der Ausgänge von Aktionen bzw. *InitialNodes*
- $FAK(n)$ = Anzahl der ausgehenden Kanten pro *ForkNode* n
- $DAK(n)$ = Anzahl der ausgehenden Kanten pro *DecisionNode* n
- $DAT(n)$ = Anzahl der an *DecisionNode* n maximal gleichzeitig anliegender Tokens. Berechnung analog zu **CALCULATEPOSSIBLEDATASOURCES** (siehe Abschnitt 3.2.4.2)
- $JEK(n)$ = Anzahl eingehender Kanten pro *JoinNode* n

Diese Werte bestimmen bereits komplette Flüsse, nicht nur einzelne Teile davon. Daher ergibt sich für die Anzahl der einfachen Kontrollflüsse SCF folgende Summe:

$$\begin{aligned}
 SCF = & AA + \\
 & \sum_{n \in ForkNode} FAK(n) + \\
 & \sum_{n \in DecisionNode} (DAK(n) - 1) \times DAT(n) - \\
 & \sum_{n \in JoinNode} JEK(n) - 1
 \end{aligned}$$

Zur Berechnung der Anzahl der einfachen Flüsse mit Objektflüssen, muss der Wert JEK dahingehend angepasst werden, dass eingehende Objektkanten nicht mitgerechnet werden. Außerdem müssen implizite Fork- und *JoinNodes* bei den jeweiligen Funktionen mit berücksichtigt werden.

Die Bestimmung der Anzahl der kombinierten Flüsse gestaltet sich ungleich schwieriger, da das Diagramm an sich keine Informationen über mögliche Kombinationen bietet. Es wäre daher eine Nachbildung der Kombinationen notwendig.

Durch die Verwendung von \langle Guards \rangle an ausgehenden Kanten von *ForkNodes*, wird über das Makro `CREATEFLOWCOMBINATIONSFORFORKNODE` für jede mögliche Kombination der negierten und nicht-negierten \langle Guards \rangle ein Fluss erstellt. Insbesondere während der Kombination von Flüssen erhält man dadurch eine potenzierte Menge von Flüssen. Abb. 4.6 zeigt ein Aktivitätsdiagramm mit \langle Guards \rangle an zwei der drei ausgehenden Kanten des *ForkNodes*.

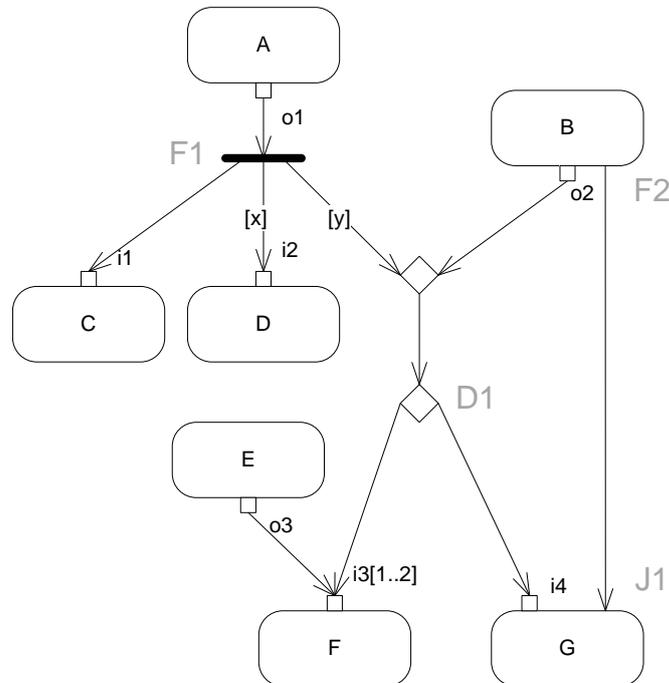


Abbildung 4.6: Aktivitätsdiagramm mit \langle Guards \rangle an ausgehenden Kanten des *ForkNodes*.

Nach obiger Formel erhält man ohne Berücksichtigung der \langle Guards \rangle neun einfache Flüsse ($AA = 3$, $FAK(F1) = 3$, $FAK(F2) = 2$, $DAK(D1) = 2$, $DAT(D1) = 2$, $JEK(J1) = 2$) und zwar folgende:

1. $A \longrightarrow \text{Puffer}(F1,1), \text{Puffer}(F1,2), \text{Puffer}(F1,3)$
2. $\text{Puffer}(F1,3) \longrightarrow F$
3. $\text{Puffer}(F1,3), \text{Puffer}(B,2) \longrightarrow G$
4. $\text{Puffer}(F1,1) \longrightarrow C$
5. $\text{Puffer}(F1,2) \longrightarrow D$
6. $\text{Puffer}(B,1), \text{Puffer}(B,2) \longrightarrow G$
7. $\text{Puffer}(B,1) \longrightarrow F$
8. $B \longrightarrow \text{Puffer}(B,1), \text{Puffer}(B,2)$
9. $E \longrightarrow F$

$\text{Puffer}(x,n)$ steht dabei für die Pufferung an der n -ten Kante des Knotens x . Für die Aktion B werden ebenfalls Puffer erzeugt, da es durch die beiden ausgehenden Kanten zu einem impliziten *ForkNode*-Verhalten kommt. Die Anzahl der kombinierten Flüsse beläuft sich in diesem Beispiel auf 54.

Auf Grund der Guardkombinationen am *ForkNode* F1 ergeben sich drei weitere Flüsse: $\neg x \wedge \neg y$, $\neg x \wedge y$, $x \wedge \neg y$. Der Fall $x \wedge y$ entspricht dem Fluss ohne $\langle \text{Guards} \rangle$. Die Kombination ergibt nun 78 Flüsse. Die zusätzliche Anzahl von 24 Flüssen ergibt sich aus der Tatsache, dass der rechte Teil (die Flüsse zwischen den Aktionen B, E, F und G), acht Flüsse beinhaltet. Dieser Wert multipliziert mit den drei zusätzlichen Flüssen auf Grund der $\langle \text{Guards} \rangle$ ergibt die Steigerung um 24.

Wie man sieht, wird die Anzahl der Flüsse insbesondere durch Datenflüsse schnell groß. Würde man das Beispiel aus Abb. 4.6 nur mit Kontroll- statt mit Datenflüssen betrachten, wären es elf einfache und 26 kombinierte Flüsse.

Eine weitere Quelle für viele Flüsse ist die Verwendung von upperBounds größer Eins. Das Makro `CALCULATEMULTIDATAFLOWS` fügt zu den einfachen Flüssen die Kombinationen der Flüsse hinzu, die sich durch die höhere Aufnahmekapazität des entsprechenden *InputPins* ergeben. In obigem Beispiel werden für den *InputPin* i3 drei weitere Flüsse erzeugt, die wiederum mit allen anderen kombiniert werden könnten.

Neben den Flüssen tragen auch die Aktionen selbst zur Anzahl der Zustandsübergänge bei. Für jede Aktion wird ein Zustandsübergang von `running` nach `offeringToken` erzeugt, so dass deren Summe einfach berechnet werden kann.

Nur im Fall der *AcceptEventActions*, muss, um den gleichzeitigen Empfang von Signalen simulieren zu können, für jede Kombination der empfangsbereiten und nicht-empfangsbereiten *AcceptEventActions* ein Zustandsübergang erzeugt werden (`CREATECOMBINATIONSFORHANDLING SIGNAL` und `CREATECOMBINATIONSFORHANDLING SIGNAL IMMEDIATELY`). Die Verwendung der Potenzmenge ergibt eine Potenzierung und somit enorme Steigerung der Zustandsübergänge in diesem Fall.

4.3.2 Optimierungen auf Ebene der Aktivitätsdiagramme

Dieser Abschnitt beschreibt einige Aspekte, wie eine Zustandsreduktion durch Veränderungen oder Ergänzungen im Aktivitätsdiagramm erreicht werden könnte. Die hier genannten Vorschläge wurden nicht formal ausgearbeitet oder anderweitig überprüft. Sie sollen nur Hinweise geben, in welche Richtung diesbezüglich weiter geforscht werden kann.

Verzicht auf Datenflüsse. Wie im letzten Abschnitt deutlich wurde, würde vor allem der Verzicht auf Datenflüsse die Anzahl der Zustandsübergänge erheblich reduzieren. Dies ist im Allgemeinen aber nicht praktikabel. Dennoch könnte in Anlehnung an die weiter oben vorgestellte *cone of influence reduction* zumindest auf die Datenflüsse verzichtet werden, die für die momentan zu prüfende Eigenschaft keine Rolle spielen.

Bei dieser Art der Reduktion ist allerdings auch zu beachten, dass sich die Semantik des Aktivitätsdiagramms zumindest für den relevanten Teil nicht verändert. Eine bloße Ersetzung der Objektflüsse durch Kontrollflüsse ist häufig nicht ausreichend.

Zum Beispiel bei der Umsetzung von mehreren eingehenden Kanten an einem *InputPin* in mehrere eingehende Kanten der Aktion, würde das Verhalten von einem impliziten *MergeNode* zu einen impliziten *JoinNode* verändern. Dementsprechend müsste in diesem Fall zusätzlich ein *MergeNode* eingefügt werden. Ähnliches gilt für andere Situationen.

Verzicht auf \langle Guards \rangle und lower- und upper-Werte. Der Verzicht auf \langle Guards \rangle und lower- oder upper-Werte größer als Eins würde ebenfalls eine Reduktion der Zustandsübergänge mit sich bringen. Diese Einschränkungen wären für viele Anwendungen vertretbar bzw. könnten teilweise durch andere Konstrukte simuliert werden. Lower-Werte größer Eins an *InputPins* bewirken ein implizites Join, da dadurch erzwungen wird, dass mehrere Flüsse gleichzeitig eintreffen müssen, damit die dazugehörige Aktion schalten kann.

Der Ersatz mit einem *JoinNode* ist hier allerdings unzureichend, da eine beliebige Untermenge der möglichen Kombinationen erlaubt ist. Falls beispielsweise drei Kanten in einem *InputPin* eintreffen und der *Pin* einen lower-Wert von zwei hat, dann können entsprechende Kombinationen von zwei aus drei Flüssen gebildet werden. Ein *JoinNode* würde immer fordern, dass an allen drei Kanten Tokens angeboten werden.

Unabhängige Signalempfänger. Weniger Eingriff in ein Aktivitätsdiagramm erfordert die Umsetzung unabhängiger Signalempfänger. In der Realität wird der Fall, dass zwei *AcceptEventActions* ein Signal absolut gleichzeitig empfangen, nie eintreten. Daher könnten Signalempfänger, die unabhängig voneinander sind, beispielsweise mit \langle Tags \rangle entsprechend gekennzeichnet werden. Dadurch würde sich die Potenzmenge der zu behandelnden *AcceptEventActions* verringern — mit der entsprechenden Auswirkung auf die Anzahl der Zustandsübergänge.

Es ist ebenfalls denkbar, die Unabhängigkeit von Signalempfängern zum Standard zu definieren und solche *AcceptEventActions* mit \langle Tags \rangle zu versehen, die gleichzeitig Signale empfangen können oder müssen.

Zusammenfassung von Aktionen. Aktionen in Aktivitätsdiagrammen sind „[...] *the fundamental unit of executable functionality*.“ [UML, Seite 236]. Durch diesen elementaren Anspruch werden komplexere Vorgänge meist durch eine Aneinanderreihung von mehreren Aktionen modelliert. Die einzelnen Aktionen stellen dann die einzelnen Schritte dar. Dies ist besonders bei der Anwendung von Aktivitätsdiagrammen als Beschreibung von Workflows der Fall. Abb. 3.4, welches den Ablauf des Model Checkings für Aktivitätsdiagramme zeigt, ist ein Beispiel dafür.

Dieser Umstand lässt sich auch zur Optimierung von Aktivitätsdiagrammen einsetzen. In Abb. 4.7 ist ein Ablauf für „Essen gehen in der Mensa“ dargestellt. Einige Aktionen sind linear ohne dazwischenliegende Kontrollknoten angeordnet.

Dieser Umstand lässt sich ausnutzen, falls die zu prüfende Spezifikation keine dieser Aktionen enthält. Dann können sequentiell auftretende Aktionen zu einer einzigen reduziert werden. Angenommen, im gegebenen Beispiel lautet die Spezifikation: Ist die Aktion „Essen“ erreichbar? Dann könnten die in verschiedenen Blautönen gruppierten Aktionen jeweils zu einer Aktion zusammengefasst werden, was den Zustandsraum bereits erheblich einschränken würde.

Darüber hinaus kann unter bestimmten Umständen ein Unterbrechungsbereich wie eine Blackbox betrachtet werden, in die einige Pfeile hinein- und einige Pfeile herausführen. Falls sich auch darin keine, für die Überprüfung der Spezifikation relevanten Aktionen befinden, kann der gesamte Unterbrechungsbereich zu einem einzigen *DecisionNode* reduziert werden. Abb. 4.8 zeigt das Ergebnis dieser Vereinfachung.

Auch wenn das Beispiel übertrieben reduziert wurde, veranschaulicht es das Potential dieser Vorgehensweise, da sich sequentielle Aktionen relativ häufig in Aktivitätsdiagrammen

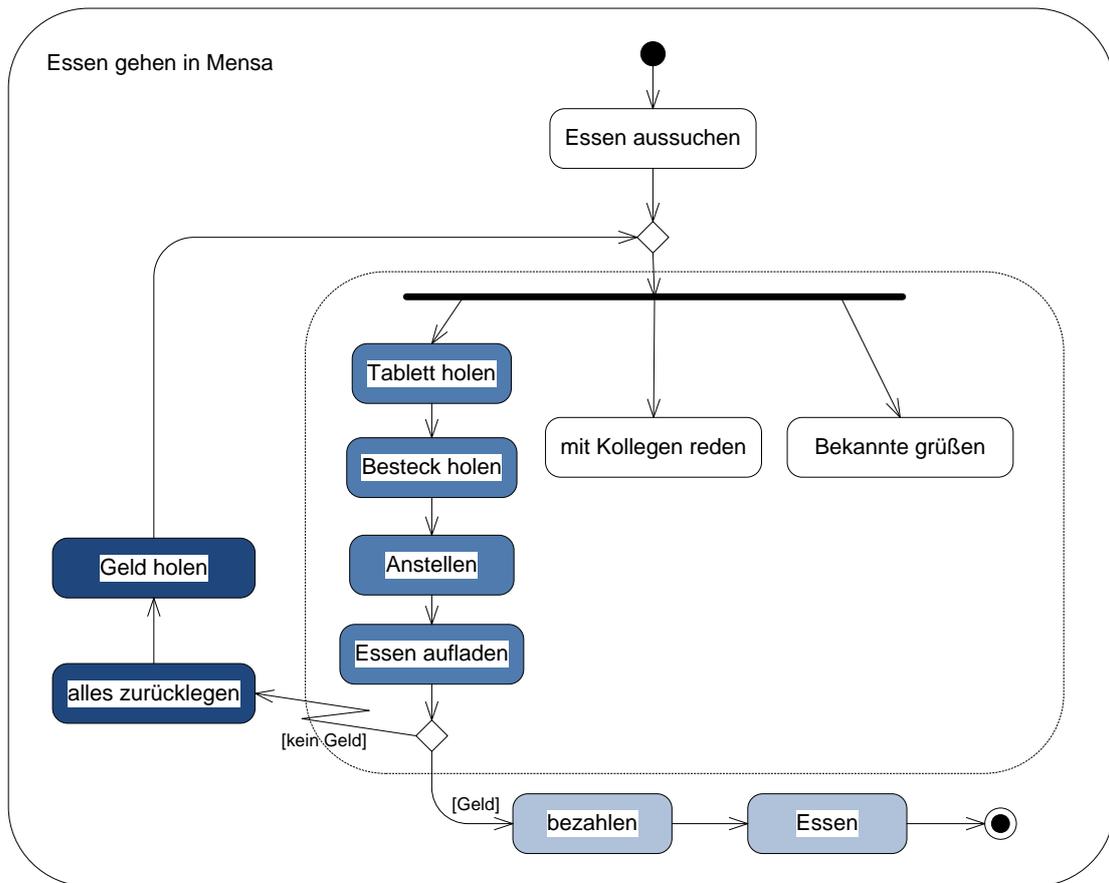


Abbildung 4.7: Beispiel zur Optimierung durch Zusammenfassen von Aktionen.

finden. Andererseits darf die Reduktion nur durchgeführt werden, wenn sicher ist, dass sie keine Auswirkungen auf das Ergebnis des Model Checkings hat. Insbesondere *SendSignal*- und *AcceptEventActions* können das Verhalten an anderer Stelle beeinflussen und dürfen somit meist nicht reduziert werden.

Noch einen Schritt weiter geht eine Methode, die für die Aktivitätsdiagramme einen bestimmten anfänglichen Ablauf vorgibt, so dass das Model Checking nur noch für einen Teilaspekt durchgeführt wird. Auch hierfür könnten die in Abschnitt 4.2 beschriebenen Sequenzdiagramme eingesetzt werden.

Über ein Sequenzdiagramm wird die Reihenfolge der auftretenden Signale spezifiziert und damit das System in einen bestimmten Zustand versetzt. Dieser Zustand dient als Ausgangszustand für das Model Checking. Somit ist der Zustandsraum auf den Teilbaum beschränkt, der vom vorgegebenen Pfad bestimmt wird.

4.3.3 Optimierungen auf Übersetzungsebene

In diesem Abschnitt werden Optimierungen beschrieben, die die Übersetzung von Aktivitätsdiagrammen in Zustandsübergangssysteme betreffen. Einige der im letzten Abschnitt beschriebenen Optimierungen könnten auch in diesem Abschnitt angesiedelt werden. Die Zu-

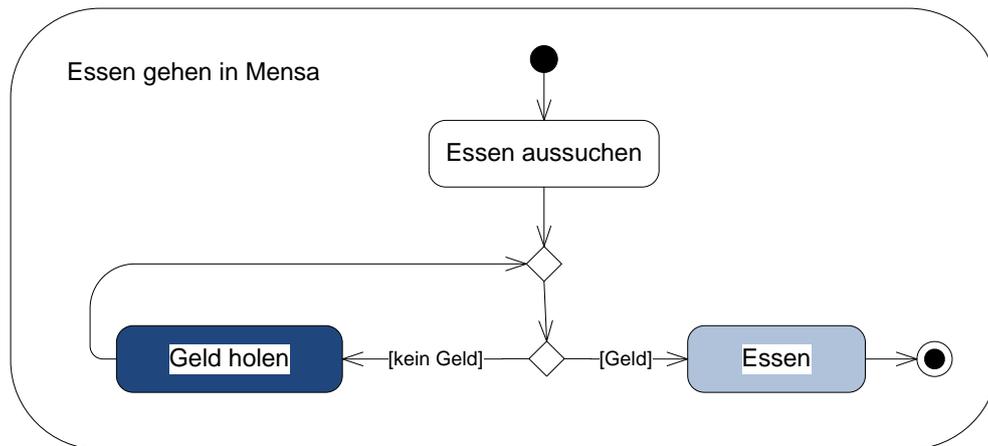


Abbildung 4.8: Ergebnis der Optimierung von Abb. 4.7.

sammenfassung von Aktionen beispielsweise kann mit gewissen Einschränkungen auch automatisch auf Übersetzungsebene erfolgen. Ein Schritt in diese Richtung ist der folgende Ansatz:

Elimination leerer Zustandsübergänge. Der Zustandsübergang vom Modus *running* in den Modus *offering* ist für *CallOperationActions*, die keinen Code enthalten, sehr einfach zu realisieren:

```

[Action]
CallOperationAction1_Running2Offering()
  require CallOperationAction1 = running
  CallOperationAction1 := offeringToken
  
```

Diese „leeren“ Zustandsübergänge sind einfach zu erkennen und zu eliminieren. Eine Aneinanderreihung mehrerer leerer Zustandsübergänge mit ebenso einfachen Flüssen dazwischen entspricht dem im letzten Abschnitt vorgestellten „Zusammenfassen von Aktionen“ und könnte daher auch auf Übersetzungsebene erledigt werden.

Elimination unerfüllbarer Flüsse. Ein weiteres Verfahren führt eine statische Analyse der Objektflüsse durch und entfernt Flüsse, die nicht erfüllbar sind. Solche Fälle treten bei *InputPins* mit einem *lower*-Wert größer Eins auf. Falls der *lower*-Wert beispielsweise den Wert drei hat und auf Grund der restlichen Konstellation im Aktivitätsdiagramm maximal nur zwei Objektflüsse gleichzeitig den *InputPin* bedienen können, kann die dazugehörige Aktion niemals starten.

Die Bedeutung von *lower*-Werten ist dabei nicht mit der *weight*-Angabe zu verwechseln, die die UML ebenfalls definiert. Während ein *lower*-Wert bedeutet, dass zu einem Zeitpunkt exakt so viele Objekte wie angegeben anliegen müssen, können diese bei einer *weight*-Angabe zeitlich kumuliert werden.

Der nicht erfüllbare Fluss kann aus der Menge der Flüsse herausgenommen werden. Gleiches gilt für die dazugehörige Aktion, deren Zustandsübergänge nicht erstellt werden müssen.

Sinnvollerweise sollten auch die Auswirkungen der Nichterreichbarkeit einer Aktion über das gesamte Diagramm überprüft werden, um noch mehr Zustandsübergänge zu eliminieren.

In diesem Fall sollte der Anwender allerdings mit einer Warnung informiert werden, da ein nicht erfüllbarer Fluss offensichtlich einen Fehler im Diagramm darstellt.

Moduscodierung. In Abschnitt 4.1.2 wurden die Modi einer Aktion als Aufzählungs- bzw. Integertypen definiert. Dadurch ergibt sich eine Menge an Zustandsvariablen, die jeweils verschiedene Werte einnehmen können.

Eine andere Möglichkeit wäre, drei Mengen für die drei Modi und die Aktionen als Elemente eines Aufzählungstyps zu definieren. Eine Aktion würde damit immer in genau einer der drei Mengen enthalten sein und Modusänderungen würden durch Entnahme eines Elements aus der einen Menge und Hinzufügen zur anderen Menge realisiert werden.

Diese Art der Codierung wurde prototypisch für den MPC implementiert. Dabei wurde allerdings festgestellt, dass der Model Checker zur Berechnung der Ergebnisse länger braucht. Dies liegt einerseits daran, dass er nicht auf die Berechnung von Mengen optimiert ist. Andererseits wird das Wissen, dass es sich um Mengen mit einer maximalen Größe und disjunkten Elementen handelt, nicht ausgenutzt.

Mit diesem Zusatzwissen könnte jede Aktion als genau ein Bit in einem Bitvektor fixer Größe dargestellt werden. Mengenoperationen könnten dann auf Bitoperationen abgebildet werden, wodurch sich einerseits Geschwindigkeitsvorteile ergeben, andererseits der Speicherplatz pro Zustand optimiert werden könnte.

Die zuletzt vorgestellte Anpassung der Modusdarstellung stellt spezielle Anforderungen an den verwendeten Model Checker. Der Model Checker sollte die für eine komfortable Umsetzung erforderlichen Mengen- und Listentypen unterstützen und deren Operationen optimieren.

Des Weiteren ist es denkbar, dass die spezielle Struktur des erstellten Zustandsübergangssystems ausgenutzt werden kann, um Model Checker auf die Verifikation von Aktivitätsdiagrammen zu optimieren. Darauf soll jedoch in dieser Arbeit nicht weiter eingegangen werden.

Kapitel 5

Diskussion und verwandte Arbeiten

Dieses Kapitel beschreibt in Abschnitt 5.1 zunächst verschiedene Ansätze zur automatischen Verifikation von Aktivitätsdiagrammen und begründet den in dieser Arbeit gewählten Weg. Abschnitt 5.2 diskutiert die vorgestellte Lösung und zeigt einige mögliche Erweiterungen zur Überwindung der in Abschnitt 3.1 beschriebenen Einschränkungen auf. Verwandte Arbeiten zur vorliegenden Arbeit werden in Abschnitt 5.3 vorgestellt und Unterschiede zu dieser Arbeit deutlich gemacht.

5.1 Alternative Vorgehensweisen

Ausgehend von der Problemstellung, Aktivitätsdiagramme mit einer vorliegenden ASM-Semantik zu verifizieren, sind folgende prinzipielle Lösungen denkbar, die im Folgenden diskutiert werden:

- Direktes Model Checking der Abstract State Machines der formalen Semantik
- Neuentwicklung eines Model Checkers speziell für Aktivitätsdiagramme
- Übersetzung in Zustandsübergangssysteme um vorhandene Werkzeuge zu nutzen

Direktes Model Checking von ASMs. Eine erste Idee ist, Aktivitätsdiagramme direkt über ihre formale Darstellung als Abstract State Machines aus der Arbeit von Sarstedt [Sar06a] zu verifizieren. Dazu ist ein Model Checker für Abstract State Machines notwendig.

Es gibt zwar einige Ansätze, Abstract State Machines zu überprüfen, allerdings reichen deren Möglichkeiten nicht aus. Je nach Ansatz treten verschiedene Probleme auf, warum die verfügbaren Werkzeuge nicht geeignet sind, die komplexen ASMs, wie sie in der formalen Semantik eingesetzt werden, zu verarbeiten.

Entweder werden nur einfache Konstrukte unterstützt oder der Zustandsraum wird durch explizites Model Checking sehr groß. Eine detailliertere Betrachtung erfolgt in Abschnitt 5.3.

Model Checker speziell für Aktivitätsdiagramme. Die Neuentwicklung eines Model Checkers speziell für Aktivitätsdiagramme ist eine weitere Alternative. Dieser Model Checker müsste für alle möglichen Abläufe eines Aktivitätsdiagramms die Ablaufpfade erstellen und den entstehenden Baum anschließend auf kritische Pfade untersuchen. Dies entspricht dem Vorgehen bei explizitem Model Checking (vgl. Abschnitt 2.1.3).

Aus dem bekannten Grund der Zustandsexplosion ist auch hier der Einsatz symbolischer Methoden sinnvoll, die für Aktivitätsdiagramme aber neu entwickelt bzw. angepasst werden müssen. Auch wenn der Aufwand hierfür vermutlich vertretbar wäre, würden sich dennoch einige Nachteile daraus ergeben, wie im übernächsten Absatz verdeutlicht wird.

Die Formeln, mit denen zu prüfende Eigenschaften spezifiziert werden, enthalten typischerweise wiederum Zustände (bzw. Modi) der einzelnen Aktionen. Ein Beispiel hierfür wäre: „Wann immer Aktion B gestartet wird (bzw. läuft), muss Aktion A beendet sein.“ Daher wäre es notwendig, entsprechende Systemzustände zu definieren. Um die Übergänge zwischen diesen Zuständen zu erhalten, könnten die Abläufe der Aktivitätsdiagramme simuliert werden, statt sie statisch zu berechnen.

Wie man sieht, würden also auch viele der für die Erstellung dieser Arbeit notwendigen Schritte bei der Neuentwicklung eines Model Checkers speziell für Aktivitätsdiagramme anfallen. Bei dieser Neuentwicklung müssten allerdings viele der in den letzten 15 Jahren entwickelten Optimierungsmethoden beim Model Checking nachimplementiert werden.

Darüber hinaus sind viele der in den erhältlichen Model Checkern verwendeten Optimierungen nicht explizit dokumentiert, sondern direkt im Code enthalten und speziell auf die konkrete Implementierung angepasst. Daher müssten viele Optimierungstechniken erst mit sehr großem Aufwand aus verfügbarem Code extrahiert werden.

Übersetzung in Zustandsübergangssystem. Der hier gewählte Ansatz, Aktivitätsdiagramme in Zustandsübergangssysteme zu transformieren, bietet den Vorteil, beliebige existierende Model Checker einzusetzen und somit auch von deren Weiterentwicklung ohne großen Aufwand zu profitieren.

Ein konkretes Beispiel für diesen Vorteil ist, dass wenige Wochen vor Fertigstellung dieser Arbeit eine neue Version des SMT-Solvers Z3 [dMB08], der als Basis für den Model Program Checker dient, veröffentlicht wurde. Diese neue Version unterstützt nun nativ Listen. Daher ist eine Anpassung des Model Program Checkers einfach möglich und somit sind viele der in Abschnitt 4.1.2 dargestellten Schwierigkeiten bei der Umsetzung gelöst.

5.2 Diskussion der gewählten Lösung und mögliche Erweiterungen

Die gewählte Lösung, Aktivitätsdiagramme in Zustandsübergangssysteme zu übersetzen und diese dann von beliebigen Model Checkern verifizieren zu lassen, bietet, wie im letzten Abschnitt beschrieben, einige Vorteile:

- Wiederverwendung der Optimierungen in den verschiedenen Model Checkern.
- Ausnutzung der Spezialisierung von Model Checkern je nach Art des Aktivitätsdiagramms (mit/ohne Objektflüsse, viele einfache Aktionen, wenige komplexe Aktionen, etc.)
- Einfache Nutzung weiterer Entwicklungen auf dem Gebiet des Model Checkings.

Allerdings müssen dafür auch ein paar Einschränkungen gemacht werden, die in Abschnitt 3.1 beschrieben sind. In diesem Abschnitt werden nun diese Einschränkungen und

ihre Auswirkungen diskutiert. Außerdem werden Möglichkeiten aufgezeigt, wie diese überwunden werden können und einige Erweiterungen dargestellt.

Wie in Abschnitt 3.1 dargestellt, führen einige der Einschränkungen zu einer Überabschätzung (engl. *over approximation*). Dadurch werden zwar zu viele, aber zumindest nicht zu wenige potentielle Fehler gefunden. Dennoch kann dies das Model Checking unbrauchbar machen, falls entweder zu viele oder sogar nur unrealistische Fehler gemeldet werden. In diesem Fall kann Counterexample-Guided Abstraction Refinement (CEGAR) [CGJ⁺03] eingesetzt werden.

Bei diesem Prinzip werden Gegenbeispiele, die sich durch das Model Checking auf Grund einer zu grob gewählten Abstraktion ergeben, automatisch überprüft. Bei Aktivitätsdiagrammen könnte versucht werden, den zurückgelieferten Ablauf durch Interpretation zu testen.

Wird dabei festgestellt, dass dieser Ablauf bei einer Interpretation nicht nachgestellt werden kann, muss das Modell oder die Spezifikation so verfeinert werden, dass dieser Fall bei einer weiteren Überprüfung ausgeschlossen wird. Wird dieses Verfahren iterativ durchlaufen, werden nach und nach die „unechten“ Fehler eliminiert und man erhält entweder „echte“ Fehler oder ein verifizierbares Modell.

AcceptEventActions mit Zeitsignalen

Die Einführung von *TimeEvents* als möglicher *Trigger* für *AcceptEventActions* in der UML erlaubt es komfortabel, zeitliche Aspekte in Aktivitätsdiagrammen zu modellieren. Diese verkürzt genannten Timer können eingesetzt werden, um, in Kombination mit Unterbrechungsbereichen, eine zeitliche obere Schranke für die Verarbeitung von mehreren Aktionen zu spezifizieren. Das System „Zweihandpresse“ aus Abschnitt 4.1.3.2 gibt dafür ein Beispiel. Außerdem können Aktionen um eine bestimmte Zeitspanne verzögert und damit z. B. periodische Signale modelliert werden.

Um Zeit in die Verifikation einbeziehen zu können, müssen Formalismen und dazugehörige Werkzeuge gewählt werden, die Zeitaspekte berücksichtigen. Ein verbreiteter Formalismus sind Timed Automata [HMU02]. Timed Automata sind endliche Zustandsautomaten, die mit einem Zeitmodell erweitert werden.

Dazu werden sogenannte „Uhrvariablen“ eingeführt, die synchron in bestimmten Zeitschritten erhöht werden. Die Transitionen und Zustände können mit Zeitbedingungen (Guards bzw. Invariants) versehen werden. Außerdem können an Transitionen die Zeitvariablen zurückgesetzt werden. Eine Transition findet dann statt, wenn neben den Ereignissen und Bedingungen auch die Zeitbedingungen erfüllt sind.

Das Werkzeug Uppaal [BDL04] verifiziert über einer Teilmenge von CTL erstellte Spezifikationen und liefert gegebenenfalls ein Gegenbeispiel zurück. Es lassen sich Erreichbarkeits-, Sicherheits- und Lebendigkeitseigenschaften prüfen. Ein weiteres Werkzeug zur Verifikation von Timed Automata, welches allerdings nicht mehr weitergepflegt wird, ist Kronos [Yov97].

Damit nun Aktivitätsdiagramme mit Timern verifiziert werden können, genügt es nicht, eine Übersetzung des Zustandsübergangssystems in Timed Automata zu definieren. Zusätzlich müssen alle Aktionen mit einer Abschätzung ihrer maximalen Zeitdauer versehen werden.

Dies könnte beispielsweise in an Aktionen angehängten Notizen erfolgen. Damit können dann Abschätzungen bezüglich der Zeitdauer von Aktionen getroffen und damit Auswirkungen von Timern berücksichtigt werden. Ebenso ist die Angabe einer minimalen Dauer denkbar, so dass genauere Aussagen getroffen werden können.

Abb. 5.1 zeigt ein Beispiel wie diese Annotation aussehen könnten. **dur** steht dabei für duration und gibt in einem Intervall die minimalen und maximalen Zeiteinheiten an, die zur

Ausführung dieser Aktion benötigt werden.

Ein Model Checker könnte verschiedene Folgen von Aktionen erzeugen, bei denen einmal der Timer nicht abläuft (da für die beiden Aktionen der Minimalwert der Ausführungszeit genommen wird), und einmal der Timer von 4s abläuft, bevor Schritt 2 beendet ist (da die maximalen Ausführungszeiten angenommen wurden).

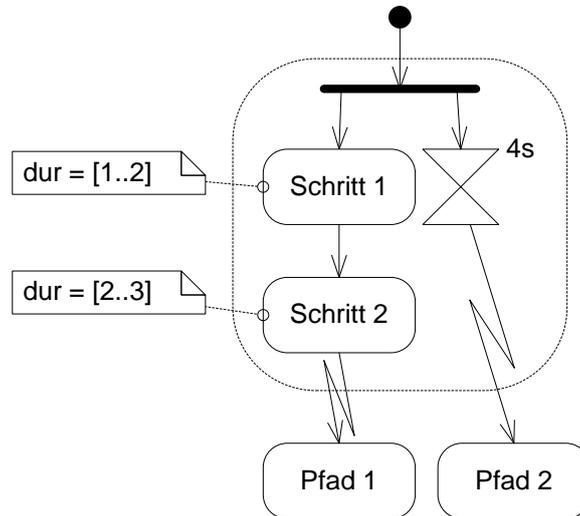


Abbildung 5.1: Beispiel für ein mit Ausführungszeiten versehenes Aktivitätsdiagramm.

Auf diese Weise würden nur noch alle realistischen Pfade zur Überprüfung der spezifizierten Eigenschaft durchlaufen werden. Fälle, die durch die Zeitangaben nicht erreicht werden können, würden unberücksichtigt bleiben. In obigem Beispiel könnte der Timer mit 6s annotiert sein und damit würde (unterschiedlich zur Verifikation ohne Timer) die Aktion *Pfad 2* nie betreten werden.

Ein Nachteil der Benutzung von Timed Automata und den entsprechenden Werkzeugen ist, dass die Möglichkeiten der Datenmanipulation auf Integer-Werte beschränkt sind. Es werden keine Listen, Mengen oder andere komplexere Datentypen und auch keine *if*-Konstrukte unterstützt.

Signale und Signalpfade

Wie bereits in Abschnitt 3.1 beschrieben, können die meisten Model Checker während der Modellüberprüfung keine neuen Instanzen von Klassen oder Ausführungen erzeugen. Daher wurde für diese Arbeit festgelegt, dass es jeweils nur eine Instanz gibt. Darüber hinaus werden alle Signale als *BroadcastSignal* an alle möglichen Empfänger gesendet.

Falls dieses Verhalten nicht gewünscht wird, würde dies den Übersetzungsprozess sogar vereinfachen. Statt jeweils einen Zustandsübergang für die Kombination aller möglichen Signalempfänger zu erstellen (siehe Makros `CREATECOMBINATIONSFORHANDLING SIGNAL` und `CREATECOMBINATIONSFORHANDLING SIGNAL IMMEDIATELY`), würde für jeden einzelnen Signalempfänger genau ein Zustandsübergang erstellt werden. Der Model Checker würde dann einen dieser Zustandsübergänge und damit den Signalempfänger zufällig auswählen.

Falls mehrere Instanzen berücksichtigt werden sollen, müssen diese *vor* der Modellüberprüfung erstellt werden. Dies ist beispielsweise durch eine Kopie von Aktivitätsdiagrammen und einer disjunkten Benennung der betroffenen Aktionen möglich. Da somit auch alle In-

stanzen vorher feststehen, könnten auch Signalpfade, die auf die unterschiedlichen Instanzen verweisen, teilweise berücksichtigt werden. Teilweise deshalb, da nur schwer z. B. dynamische Inhalte bei der Bestimmung der Zielaktivitäten berücksichtigt werden können.

Signalpfade, die z. B. ausdrücken, „alle Aktivitäten, deren Aktion A sich gerade im Modus *running* befindet, sollen das Signal erhalten“, sind nur schwer zu realisieren, da dann für alle möglichen Kombinationen wieder eigene Zustandsübergänge erstellt werden müssten.

Eine Möglichkeit, sich die manuelle Erstellung der Kopien und Umbenennungen der Aktionen zu sparen, ist, ein Objektdiagramm anzugeben. Ein Objektdiagramm [RQZ07] ist ähnlich einem Klassendiagramm aufgebaut, allerdings mit dem Unterschied, dass es statt der prinzipiell möglichen Instanzen und Relationen, die zu einem aktuellen Zeitpunkt tatsächlichen Instanzen und deren Beziehungen untereinander angibt.

Ein Objektdiagramm stellt somit eine mögliche konkrete Ausprägung eines dazugehörigen Klassendiagramms dar. Die darin enthaltenen Instanzen können dazu verwendet werden, die oben erwähnte Erzeugung der Kopien und Umbenennungen automatisch durchführen zu lassen. Eine Erweiterung des Werkzeugs *ActiveChartsIDE* [SGK⁺05] in diese Richtung ist sehr wahrscheinlich, da die Erstellung der Instanzen auch ein Problem beim Einsatz dieses Werkzeugs ist.

Eine weitere Möglichkeit, mit dynamisch kreierten Instanzen umzugehen, bietet das Modellierungsframework *IF-2.0* [BGM02]. Es basiert auf „dynamic extended timed automata“ die pro Komponente (process) definiert sind. Die Komponenten können während der Laufzeit instanziiert oder vernichtet werden und haben einen lokalen Speicher mit Variablen. Die Prozesse können untereinander Nachrichten austauschen.

In den Transitionen kann neben elementaren Aktionen wie Wertzuweisungen, Instanziierung und Senden von Signalen auch externer Code aufgerufen werden. Dies wird ermöglicht, indem Model Checking über explizite Exploration des Zustandsraums erfolgt. Von den damit verbundenen Nachteilen abgesehen, könnte das Framework für eine Lösung des Instanziierungsproblem interessant sein.

⟨Guards⟩ als Labels

In der vorgestellten Verifikationsmethode von Aktivitätsdiagrammen müssen ⟨Guards⟩ wahr sein, damit der entsprechende Fluss schalten kann. Gerade bei der Verwendung von Aktivitätsdiagrammen zur Darstellung von Workflows, werden die ⟨Guards⟩ typischerweise eher als Unterscheidung verschiedener Wege eingesetzt. Sie werden meist nicht als konkrete formale Daten-Bedingung genutzt.

Daher ist es für dieses Einsatzgebiet von Vorteil, wenn die Behandlung der ⟨Guards⟩ gesteuert werden kann. Dabei kann unterschieden werden, ob die ⟨Guards⟩ während des Model Checkings ausgewertet werden oder nur als Bezeichner der gewählten Kanten dienen. Im zweiten Fall sollten diese mit einem gefundenen Gegenbeispiel zurückgeliefert werden.

Modellierung der zu prüfenden Eigenschaft

Im vorgestellten Ansatz wird davon ausgegangen, dass die zu prüfende Eigenschaft in einer Eingabesprache des verwendeten Model Checkers formuliert wird. Dazu sind Kenntnisse über den Übersetzungsprozess vonnöten. Es ist beispielsweise notwendig, die verschiedenen Modi und die generierten Namen der Aktionen zu kennen. Für eine bessere Akzeptanz von formalen Methoden im Softwareentwicklungsprozess ist es allerdings erforderlich, von diesen Details weitgehend zu abstrahieren.

Es gibt bereits einige Ansätze, die sich damit beschäftigen, die zu prüfenden Eigenschaften in UML zu spezifizieren. In [KW06] und [GMP02] werden für diesen Zweck Sequenzdiagramme verwendet.

Dabei werden diese in einen „interaction automata“ überführt und als sogenannter „observer process“ synchron mit dem zu prüfenden System ausgeführt. Dies ist ähnlich zum Model Checking mit Büchi-Automaten (siehe Abschnitt 2.1.3). In [KW06] wird ausführlich beschrieben, wie verschiedene Fragmente der UML Sequenzdiagramme in Automaten umgesetzt werden.

Den Autoren von [OGO04] gehen die damit verbundenen Möglichkeiten der Eigenschaftsspezifikation nicht weit genug. Daher schlagen sie eine Variante von UML Zustandsautomaten als Observerprozess vor. Wesentlich mehr Aspekte der Temporallogik können über Live Sequence Charts abgedeckt werden, deren Einsatz zur Spezifikation von Eigenschaften für das Model Checking in [BDK⁺04] beschrieben wird. Eine Untersuchung, inwieweit diese Ansätze auch bei der Verifikation von Aktivitätsdiagrammen eingesetzt werden können, steht noch aus.

5.3 Verwandte Arbeiten

Dieser Abschnitt befasst sich mit der Beschreibung und dem Vergleich verschiedener Arbeiten, die sich auf dem Gebiet des Model Checkings von UML und insbesondere von Aktivitätsdiagrammen beschäftigen. Dabei können vier Bereiche unterschieden werden, die in den folgenden Abschnitten näher betrachtet werden:

1. Arbeiten zur Verifikation von ASMs
2. Verifikation von UML Modellen
3. Verifikation von Workflow Modellen
4. Verifikation von UML 2 Aktivitätsdiagrammen

5.3.1 Verifikation von ASMs

Wie in Abschnitt 5.1 erläutert, wäre die Verifikation von Aktivitätsdiagrammen auf Basis der formalen Semantik in ASMs möglich. Daher wird im Folgenden kurz auf die Möglichkeiten und Einschränkungen der für diesen Formalismus existierende Model Checking Ansätze eingegangen.

Die erste Arbeit, die sich mit der Verifikation von ASMs befasste, ist [Win97] bzw. in ausführlicherer Fassung [Win01]. In diesen Arbeiten werden ASMs über eine Zwischensprache ASM-IL, die ein flaches, einfaches Zustandsübergangssystem darstellt, in eine Eingabesprache für den Model Checker SMV übersetzt. Insofern ist dieser Ansatz mit dem in dieser Arbeit vorgestellten zu vergleichen.

Als ASM-Konstrukte werden nur die einfachen ASM-Konstrukte **skip**, **if-then**, **do-forall** und **choose** unterstützt. Die **forall**-Regel wird durch Entfalten (engl. *unfolding*) auf eine Menge von **if-then**-Regeln abgebildet. Multi-Agenten-ASMs werden ebenfalls durch Entfalten aller möglichen Ablaufkombinationen zwischen den beteiligten Agenten (die alle die gleichen ASM-Regeln verarbeiten müssen) verarbeitet.

Die **import**-Regel (entspricht der *new*-Funktion) wird nicht unterstützt, da sich das Verfahren auf endliche Domänen beschränkt. Trotz dieser Einschränkungen kommt die Verifikation bereits bei kleinen Modellen an ihre Grenzen. Die Beschränkung der Domänen wird zwar in einer Erweiterung behandelt, allerdings ist die Restriktion auf reine Basiskonstrukte ungeeignet, komplexere ASMs, wie sie im Rahmen der Semantikbeschreibung der Aktivitätsdiagramme auftreten, zu verifizieren.

Ein ähnlicher Ansatz wird in [GRR03] verfolgt. Eine ähnliche Menge von Konstrukten wird in die Spezifikationsprache Promela übersetzt, die die Eingabesprache des Model Checkers SPIN ist. Da Promela mehrere Prozesse unterstützt, werden Multi-Agenten in einzelne Prozesse übersetzt. Allerdings ist die Verifikation wiederum auf endliche Domänen und wenige Konstrukte begrenzt.

Die neuere Arbeit [TT07] nutzt Answer Set Programming (ASP) zur Verifikation von ASMs. ASP ist ein deklaratives logisches Programmierparadigma zur Lösung kombinatorischer Suchprobleme. Das Problem wird als logisches Programm repräsentiert und die Ergebnismenge (engl. *answer set*), die von einem entsprechenden Solver bestimmt wird, stellt die Lösung dar.

Wie bei den vorherigen Arbeiten, bleibt die unterstützte Menge der ASM-Konstrukte auf Basiskonstrukte beschränkt, allerdings liefert dieser Ansatz schneller Gegenbeispiele. Dafür kann jedoch nur Bounded Model Checking (siehe Abschnitt 2.1.3) durchgeführt werden.

Die Verwendung möglichst vieler ASM-Konstrukte war das Ziel der Arbeit von Kardos [Kar05]. Er verwendet eine auf ASMs aufbauende Sprache namens ASML [GRS05]. Für diese Sprache existiert ein Compiler für die Plattform .NET und der Zielsprache C#.

Die Arbeit instrumentiert den Compiler so, dass während des Laufs die einzelnen Zustandsübergänge registriert werden und so explizites Model Checking möglich ist. Das Verfahren wird auch „on-the-fly“ Model Checking genannt, da nicht wirklich der komplette Zustandsraum aufgespannt wird, sondern lediglich der Teil, der für die Evaluation des momentanen Pfads notwendig ist.

Das Ziel, die Sprache ASML möglichst umfassend zu unterstützen, wurde erreicht. Leider ist der Ansatz im Vergleich mit den vorherigen Arbeiten deutlich langsamer bzw. konnte für ein Beispiel keine Lösung bieten, da die Verifikation mit einem Speicherüberlauf abgebrochen wurde. Damit zeigt sich, dass explizites Model Checking ohne spezielle Optimierungen nur für sehr kleine Modelle praktikabel ist.

Zwei weitere Arbeiten wenden die bisher vorgestellten Ansätze auf das Werkzeug CoreASM [FGG07] an. Einerseits wird in [FGM08] eine Übersetzung nach Promela mit den bekannten Einschränkungen angegeben. Andererseits wird in [BKKS08] — ähnlich zur erwähnten Arbeit auf Basis von ASML — die Simulation der CoreASM-Maschine benutzt, um den Zustandsraum aufzuspannen. Beide Arbeiten kommen zu den gleichen Ergebnissen wie die vorherigen.

Wie man sieht, sind die grundlegenden Schwierigkeiten bei der Überprüfung von potentiell unendlichen Abstract State Machines noch nicht gelöst. Alle effizienteren Arbeiten beschränken sich auf eine kleine Teilmenge der zur Verfügung stehenden Konstrukte.

Werden diese Beschränkungen durch explizites Model Checking aufgelöst, scheitern die Ansätze an dem immensen Rechen- und Speicheraufwand. Daher wurde in der vorliegenden

Arbeit die Verifikation von Aktivitätsdiagrammen über ihre formale ASM-Semantik verworfen.

5.3.2 Verifikation von UML Modellen

Die Arbeiten, die sich der automatischen Überprüfung von UML-Notationen widmen, verwenden verschiedene Diagrammtypen, um diese gegeneinander oder gegen eine andere formale Spezifikation zu prüfen. Die meisten Arbeiten fokussieren dabei auf UML State Machines, die zum Model Checking in verschiedene Formalismen überführt werden. Einen Überblick und Vergleich über verschiedene Model Checking Ansätze in Bezug auf UML State Machines gibt [Alk07].

In [GMP02] werden Hinweise gegeben, wie UML State Machines gegen Sequenzdiagramme als Spezifikationen geprüft werden können, ohne technische Details anzugeben. Knapp et. al gehen in [KMR02] einen Schritt weiter. Sie geben für UML State Machines, die um Zeitaspekte erweitert wurden, eine Übersetzung nach Uppaal [BDL04] an. Zu prüfende Spezifikationen werden dabei als Sequenzdiagramme angegeben. Eine Erweiterung dieser Arbeit auf die UML 2 findet sich in [KW06].

Die Berücksichtigung von Klassen bzw. deren Instanzen zusammen mit State Machines wird in [OGO04] behandelt. Dabei werden die State Machines ebenfalls mit Zeitangaben erweitert. Außerdem werden sie als Observer-Automat zur Eigenschaftsspezifikation verwendet. Alle Modelle werden in communicating extended timed automata übersetzt und mit dem Framework IF-2.0[BGM02] verifiziert.

Ein interessanter Ansatz wird in [SCH02] verfolgt. Dort wird eine formale Semantik für Klassen-, Objekt- und Zustandsdiagramme unter Verwendung von ASMs angegeben. Statische Aspekte werden gegen das Meta-Modell und einige Bedingungen in Form von OCL geprüft. Für die Überprüfung des dynamischen Aspekts wird der Model-Checker SMV für die ASMs verwendet. Leider werden keine Details geliefert, wie die Übersetzung der ASMs in dessen Eingabesprache aussieht.

Die Verifikation von nebenläufigen UML Modellen ist der Inhalt von [GMB08]. Die statische Struktur wird in Klassendiagrammen spezifiziert, das interne Verhalten mit State Machines und die Kommunikation zwischen den Komponenten mit Kommunikationsdiagrammen. Kommunikationsdiagramme sind ein weiterer Diagrammtyp der UML, der, aufbauend auf einem Objektdiagramm, die Kommunikation zwischen den Objekten veranschaulicht. Diese Formalismen werden in das Werkzeug Maude übersetzt, welches auf Rewriting Logic aufbaut, um beliebige LTL-Eigenschaften zu verifizieren.

Auf eine Beschreibung von Arbeiten, die sich mit der Verifikation von Aktivitätsdiagrammen der UML 1.x befassen, wird verzichtet. Die Aktivitätsdiagramme der UML 1.x sind stark an Zustandsautomaten angelehnt und verfügen noch nicht über schwierigere Aspekte wie Datenflüsse, Unterbrechungsbereiche oder Signalverarbeitung. Insofern sind sie für einen Vergleich mit der vorliegenden Arbeit uninteressant. Die wenigen Arbeiten, die sich mit der Verifikation von UML 2 Aktivitätsdiagrammen befassen, werden in Abschnitt 5.3.4 vorgestellt.

5.3.3 Verifikation von Workflow Modellen

Verifikationsansätze von Workflow Modellen sind deswegen interessant zu betrachten, da Aktivitätsdiagramme auch zur Modellierung von Workflows eingesetzt werden können und daher viele Konstrukte aus dieser Domäne enthalten. Für viele verschiedene Modelle wurden Möglichkeiten der automatisierten Überprüfung meist auf Basis von Petri-Netzen veröffentlicht.

[Mor08] gibt einen Überblick über die verschiedenen, für die automatische Verifikation von Business Modellen eingesetzten Formalismen: Automaten, Petri-Netze und Prozessalgebren. Dabei werden jeweils auch Arbeiten vorgestellt, die den entsprechenden Formalismus zur Definition der Modellsemantik und zum Model Checking verwenden. Folgende zwei Notationsformen werden im Folgenden genauer betrachtet: Die Business Process Modeling Notation (BPMN)[Bus] und Yet Another Workflow Language (YAWL)[tHvdA05].

Die BPMN ist ein weiterer Standard der OMG, der eine gewisse Ähnlichkeit mit Aktivitätsdiagrammen aufweist. In [Whi04] schreibt der Autor sogar: „*Since the Activity Diagram and Business Process Diagram are very similar and are views for the same metamodel, it is possible that they will converge in the future.*“ Allerdings fehlt — wie für die UML auch — für die BPMN eine formale Semantik, welche für das Model Checking allerdings essentiell ist. Daher gibt es bisher auch praktisch keine Arbeiten zur automatisierten Modellüberprüfung der BPMN. Neuere Arbeiten (z. B. [BT08]) versuchen diese Lücke zu füllen und eine formale Semantik für die BPMN zu definieren.

YAWL wurde ab 2002 mit dem Ziel entwickelt, eine Sprache zu schaffen, die alle bis dahin definierten Workflow Patterns unterstützt. Daher enthält YAWL auch ein Cancellation-Konstrukt welches andere Aktivitäten oder sogar Regionen abbrechen kann. Es ist daher eng verwandt mit den Unterbrechungsbereichen von Aktivitätsdiagrammen. Auch wenn die Notation der YAWL an Petri-Netze angelehnt ist, wurde die Semantik von YAWL auf Zustandsmaschinen definiert. In [Wyn06] wurde eine Übersetzung von YAWL-Konstrukten nach Petri-Netze definiert. Dies erlaubt, bestimmte Eigenschaften eines Workflows automatisiert zu überprüfen.

In obigem Zitat aus [Whi04] wird behauptet, dass die BPMN den Aktivitätsdiagrammen ähnlich ist. Allerdings fehlen ihr einige wesentliche Konstrukte. So werden weder Datenflüsse noch Unterbrechungsbereiche oder Signale wie in Aktivitätsdiagrammen unterstützt. YAWL unterstützt zwar Unterbrechungsbereiche, allerdings fehlen die beiden anderen Aspekte ebenfalls. Darüber hinaus ist die Übersetzung von YAWL-Modellen nach Petri-Netzen noch aus anderen Gründen für die Verifikation von Aktivitätsdiagrammen uninteressant, wie der nächste Abschnitt aufzeigt.

5.3.4 Verifikation von UML 2 Aktivitätsdiagrammen

Für die automatische Verifikation von UML 2 Aktivitätsdiagrammen gibt es bislang nur sehr wenige Arbeiten. Die wichtigsten werden im Folgenden vorgestellt und mit der vorliegenden Arbeit verglichen.

Verifikation über Petri-Netze. Störrle und Hausmann versuchen in [SH05] eine Semantik für Aktivitätsdiagramme auf Basis von Petri-Netzen zu definieren. Nachdem Übersetzungen

von einfachen Konstrukten des Kontroll- und Datenflusses bereits erstellt wurden [Stö05], versuchen sie, die vorhandenen Arbeiten auf komplexere Konzepte auszuweiten.

Es werden Unterbrechungsbereiche und Streams untersucht. Dabei stoßen die Autoren auf einige Schwierigkeiten bei der Umsetzung. Die Verteilung des Systemzustands über alle Marken in einem Petri-Netz führt zu großen Problemen bei der Umsetzung von Unterbrechungsbereichen. Diese verändern den Zustand einer Aktivität an mehreren Stellen gleichzeitig. Die Behandlung von Streams gestaltet sich ähnlich schwierig, da die UML Spezifikation diesbezüglich viele Unklarheiten beinhaltet.

Ein weiterer in dieser Arbeit betrachteter Aspekt ist die „*traverse-to-completion*“-Regel (siehe Abschnitt 2.3.2). Das gewählte Übersetzungsschema, für jeden Knoten im Aktivitätsdiagramm eine Stelle im Petri-Netz zu erzeugen, widerspricht dieser Regel.

Für den Fluss von einer Aktion zur nächsten sind — je nach Menge der zu passierenden Kontrollknoten — mehrere Schritte im Petri-Netz auszuführen. Insbesondere können dabei Tokens in Kontrollknoten „steckenbleiben“, falls die nachfolgende Aktion nicht zum Schalten bereit ist. Genau dies versucht aber die „*traverse-to-completion*“-Regel zu verhindern. Eine Idee, wie dieses Problem gelöst werden könnte, wird folgendermaßen beschrieben:

„We could thus discard the simple mapping presented above and devise a more complicated compilation approach which evaluates the different possible flows through a control structure of an activity diagram and maps these to simple Petri-net transitions (synchronising all inputs and outputs).“[SH05]

Im Wesentlichen entspricht der vorgeschlagene Ansatz, vorab alle möglichen Flüsse zu berechnen und pro Fluss einen Zustandsübergang zu erzeugen, dem in der vorliegenden Arbeit gewählten Vorgehen. Dieser wird aber von Störrle und Hausmann nicht weiter verfolgt, da für diese Berechnung zunächst die Semantik der einzelnen Elemente definiert sein muss. Ihre Übersetzung in Petri-Netzes sollte aber gerade die Semantik von Aktivitätsdiagrammen definieren.

Um diesen Problemen auf andere Weise zu begegnen, diskutieren Störrle und Hausmann verschiedene Varianten von Petri-Netzen, die aber alle nicht zum Erfolg führen. Außerdem führt die Kombination der verschiedenen Arten von Petri-Netzen dazu, dass diese nicht mehr verifiziert werden können. Die Autoren schließen ihre Arbeit mit dem Kommentar: „[...] *the claimed alignment of activity diagrams to Petri-Nets remains rather superficial and does not yield any benefit.*“

Verifikation über Zustandsübergangssysteme. Eine weitere Arbeit, die sich mit der Verifikation von Aktivitätsdiagrammen auseinandersetzt, ist die Arbeit von Eshuis in [Esh06]. In dieser Arbeit werden zwei verschiedene Übersetzungen für Aktivitätsdiagramme in endliche Zustandsautomaten präsentiert. Diese Automaten werden dann in die Eingabesprache des Model Checkers NuSMV[CCG⁺02] überführt.

Die erste Übersetzung geht von Zustandsautomaten aus, deren Übergänge und Reaktionen keine Zeit benötigen. Die zweite geht von einer realistischeren Umgebung aus und benutzt Eingabeschlangen, um die Umgebung vom System zeitlich zu trennen. Während die erste Übersetzung schnell und effizient zu überprüfen ist, ist dies für die zweite nicht immer möglich.

Als einheitliche, normierte Basis für beide Übersetzungen dient ein sogenannter Activity Hypergraph, bei dem *Fork*- und *JoinNodes* in Hyperkanten umgewandelt werden. Schließlich wird gezeigt, dass beide Übersetzungen für eine große Klasse der Aktivitätsdiagramme die

gleichen Model Checking Ergebnisse liefern. Die beiden Übersetzungen werden benutzt, um Datenintegritätsbedingungen zwischen einer Menge von Klassendiagrammen, die die Datenmanipulationen spezifizieren, und einem Aktivitätsdiagramm zu prüfen.

Obwohl die UML 2 zum Zeitpunkt der Arbeit bereits veröffentlicht war (der Autor benutzt die Symbole der UML 2), lehnt sich die indirekt durch die Übersetzung gegebene Semantik an die der UML 1.5 an. Allerdings: „[...] *unlike the UML 1.5 semantics, our semantics does not reduce an activity diagram to a state chart.*“ [Esh06]. Auf die genauen Unterschiede wird nicht eingegangen.

Durch diese „vermischte“ Sichtweise der Aktivitätsdiagramme werden die interessanteren Aspekte der UML 2 wie Signalbehandlung und Unterbrechungsbereiche nicht berücksichtigt. Datenflüsse werden während der Transformation in Activity Hypergraphs zu Kontrollflüssen umgewandelt.

Verifikation über Dynamic Meta Modeling. In der Veröffentlichung [ESW07] von Engels et al. wird die Semantik von Aktivitätsdiagrammen mit Hilfe des Dynamic Meta Modeling (DMM) Ansatzes spezifiziert. DMM erweitert das Metamodell einer beliebigen Sprache um bestimmte Klassen zu einer „statischen Semantik“. Dieses erweiterte Modell kann dann mit Graphtransformationsregeln verändert werden („dynamische Semantik“).

Für eine konkrete Instanz des Metamodells wird die Semantik durch ein Zustandsübergangssystem angegeben. Dessen Zustände werden durch die jeweiligen Konstellationen der statischen Semantik gebildet. Die Übergänge werden durch die Graphtransformationsregeln beschrieben.

Das Werkzeug GROOVE erstellt aus dem gegebenen Graphtransformationssystem das dazugehörige Zustandsübergangssystem und bietet auch gleich die Möglichkeit, CTL-Eigenschaften zu verifizieren. Dieses Werkzeug wird in der Arbeit eingesetzt, um einige Workflow-Eigenschaften von Aktivitätsdiagrammen zu verifizieren.

DMM ermöglicht eine präzise Umsetzung der „*traverse-to-completion*“-Semantik. Allerdings beschränkt sich der Ansatz auf die grundlegenden Elemente der Aktivitätsdiagramme, so dass keine Unterbrechungsbereiche, keine Signale und keine Datenflüsse berücksichtigt werden. Wie schon in den anderen beiden Arbeiten, werden auch keine *CallBehaviorActions* behandelt.

Bei allen drei vorgestellten Ansätzen wird die Semantik der Aktivitätsdiagramme als das Ergebnis der jeweiligen Transformation definiert. Insofern ist für diese Ansätze ein Nachweis der Semantikerhaltung nicht nötig.

Tabelle 5.1 fasst die existierenden Arbeiten zusammen und vergleicht sie bezüglich der unterstützten Konstrukte und Model Checker mit der vorliegenden Arbeit.

Tabelle 5.1: Gegenüberstellung der Arbeiten zur Verifikation von Aktivitätsdiagrammen.

Arbeit	Formalismus	eingesetzter Model Checker	unterstützte Konstrukte					
			Kontrollflüsse	Objektflüsse	traverse-to-completion	Signalverarbeitung	Unterbrechungsbereiche	CallBehaviorActions
Störrle, Hausmann	Petri-Netze	—	×	×			×	
Eshuis	Zustandsübergangssystem	NuSMV	×	×	×			
Engels et. al	Dynamic Meta Modeling	GROOVE	×		×			
vorliegende Arbeit	Zustandsübergangssystem	flexibel	×	×	×	×	×	×

Kapitel 6

Fazit und Ausblick

Ziel dieser Arbeit ist die automatisierte Verifikation (Model Checking) von Aktivitätsdiagrammen. Dies wurde durch eine Übersetzung der Aktivitätsdiagramme in endliche Zustandsübergangssysteme und deren Umsetzung in eine Eingabesprache für einen Model Checker erreicht.

6.1 Beiträge

Automatisierte Verifikation von Aktivitätsdiagrammen. Die automatisierte Verifikation von Aktivitätsdiagrammen ist eine wichtige Voraussetzung für einen breiteren Einsatz dieser neuen graphischen Notation. Dieses wird in dieser Arbeit durch die Definition einer Übersetzung der Aktivitätsdiagramme in endliche Zustandsübergangssysteme erreicht. Schwerpunkt des Algorithmus ist dabei die Berechnung der möglichen Tokenflüsse innerhalb einer Aktivität.

Die Transformation basiert auf einer mit Hilfe von Abstract State Machines spezifizierten formalen Semantik [Sar06a], die die wesentlichen Aspekte der Aktivitätsdiagramme abdeckt. Bei der Übersetzung werden weitergehende Aspekte der UML 2, die in der Semantik definiert sind, berücksichtigt.

Die Unterstützung für Datenflüsse machen Aktivitätsdiagramme für die Modellierung von Informationssystemen interessant. Unterbrechungsbereiche und die Signalverarbeitung sind einerseits hilfreich bei der Modellierung eingebetteter Systeme, andererseits finden diese Aspekte auch bei der Beschreibung von Workflows Einsatz.

Durch die Hierarchisierung über *CallBehaviorActions* können auch komplexere Systeme modelliert werden. Durch die Unterstützung dieser „schwierigeren“ Aspekte der UML 2 geht die hier vorgestellte Arbeit weit über die in bisherigen Arbeiten abgedeckten Aspekte hinaus.

Durch die definierte Schnittstelle zur Eingabesprache des zu benutzenden Model Checkers, ist es einfach möglich, verschiedene Model Checker einzusetzen. Der Ansatz profitiert damit auch unmittelbar von den Möglichkeiten verschiedener Model Checker und von weiteren Entwicklungen auf diesem Gebiet.

Formalisierung der Transformation in ASMs. Der Transformationsalgorithmus ist — wie die zugrunde liegende Semantik — mit Hilfe der Abstract State Machines formal definiert. Die an Pseudo-Code angelehnte Notation bietet den Vorteil, leicht verständlich zu sein ohne auf eine formale semantische Basis zu verzichten. Darüber hinaus erleichtert die formale Definition den Nachweis, dass die Transformation semantikerhaltend ist.

Berücksichtigung von semantischen Variationspunkten. In der verwendeten Semantik wurden Variationspunkte definiert, da die UML auf Grund der natürlichsprachlichen Definition an verschiedenen Stellen mehrdeutig und unpräzise ist. Diese ermöglichen es dem Benutzer, die Semantik und damit das Verhalten einiger Aktionen und Aktivitäten den jeweiligen Bedürfnissen anzupassen.

Einige dieser semantischen Variationspunkte werden auch bei der Übersetzung und der Verifikation berücksichtigt. Dadurch muss sich der Modellierer bei der Verwendung von Aktivitätsdiagrammen diesbezüglich nicht einschränken.

Prototypische Implementierung. Die vorgestellte formale Transformation ist prototypisch in ein Werkzeug zur modellgetriebenen Softwareentwicklung integriert. Das Werkzeug ActiveChartsIDE wurde am Institut für Programmiermethodik und Compilerbau der Universität Ulm entwickelt und ermöglicht die Ausführung von Aktivitätsdiagrammen.

Die Implementierung zeigt einerseits die Umsetzbarkeit des gewählten Ansatzes und andererseits die Möglichkeiten, die sich durch die Verifikation der Aktivitätsdiagramme ergeben.

Grundlage für weitere Forschungsarbeiten. Die in dieser Arbeit gewonnenen Erkenntnisse gerade bezüglich der Flussberechnung können als Grundlage für weitere Forschungsarbeiten dienen. Die Ausführungsgeschwindigkeit von Aktivitätsdiagrammen, welche in dem Werkzeug ActiveChartsIDE interpretiert werden, kann durch die Vorausberechnung der möglichen Flüsse optimiert werden. Für eine noch schnellere Ausführung bildet die Flussberechnung schließlich die Basis für eine vollständige Codegenerierung aus Aktivitätsdiagrammen.

Während der intensiven Auseinandersetzung mit der formalen Semantik von Sarstedt wurden einige fehlende und fehlerhafte Punkte darin entdeckt und Lösungen dafür vorgeschlagen.

6.2 Ausblick

Verschiedene Erweiterungen und optimierende Anpassungen werden bereits in den Abschnitten 4.3 und 5.2 erwähnt. Die Betrachtungen bezüglich der Umgebungsmodellierung in Abschnitt 4.2 können ausgearbeitet und verfeinert werden. Auch eine Erweiterung der Unterstützung für weitergehende Konstrukte der Aktivitätsdiagramme ist denkbar. Beispiel hierfür wäre die Definition einer Semantik für Streams und deren Berücksichtigung während der Verifikation. Dieser Aspekt ist besonders interessant, da Streams in Aktivitätsdiagrammen der SysML eine wichtige Rolle spielen.

Die — aus Sicht des Autors — wichtigsten nächsten Schritte sind allerdings:

- Anbindung verschiedener Model Checker
Die Integration verschiedener Model Checker würde weitere Hinweise liefern, welches Werkzeug für welche Fragestellung am Besten geeignet ist. Außerdem könnten somit größere Modelle verifiziert werden.
- Formaler Beweis der Semantikerhaltung
Um mit dem hier vorgestellten Ansatz sicherheitskritische Systeme zu modellieren und zu verifizieren, ist der formale Nachweis der Semantikerhaltung durch die Transformation in ein Zustandsübergangssystem verglichen zur Ausgangssemantik unabdingbar.

- Ausführliche Fallbeispiele

Auf Grund der nach wie vor mangelhaften Werkzeugunterstützung für Aktivitätsdiagramme sind derzeit praktisch keinerlei größere Systeme verfügbar. Allerdings kann erst durch die Realisierung größerer Fallstudien der tatsächliche Nutzen der automatisierten Verifikation von Aktivitätsdiagrammen ausgelotet werden.

Anhang A

ASM-Regeln

Dieser Anhang enthält eine vollständige Auflistung der in dieser Arbeit verwendeten ASM-Regeln. Regeln, die in Kapitel 3.2 nicht ausführlich erklärt sind, werden kurz erläutert, ansonsten wird auf die entsprechenden Stellen in der Arbeit verwiesen.

A.1 Domänendefinitionen

ASM-Domänen beschreiben Mengen, die gleichartige Daten beinhalten. Dieser Abschnitt definiert alle in dieser Arbeit verwendeten Domänen. Zum einen sind dies Domänen die sich aus der Formalisierung des UML Metamodells ergeben. Zum anderen Domänen, die für die Berechnung der Zustandsübergänge notwendig sind. Funktionen, die Eigenschaften von Elementen der Domäne oder Beziehungen zwischen den Domänen abbilden, werden ebenfalls in diesem Abschnitt aufgelistet.

A.1.1 Grundlegende Datentypen

Die Definition einiger grundlegender Datentypen.

static domain <i>Nat</i>	<i>Natürliche Zahlen</i>
static domain <i>Boolean</i>	<i>boolesche Werte</i>

A.1.2 UML 2 Metamodell

Das UML 2 Metamodell für den betrachteten Ausschnitt der Aktivitätsdiagramme ist in Abb. A.1 dargestellt. Daran schließen sich die Definitionen der einzelnen Domänen, wie in Abschnitt 3.2.2 beschrieben, an.

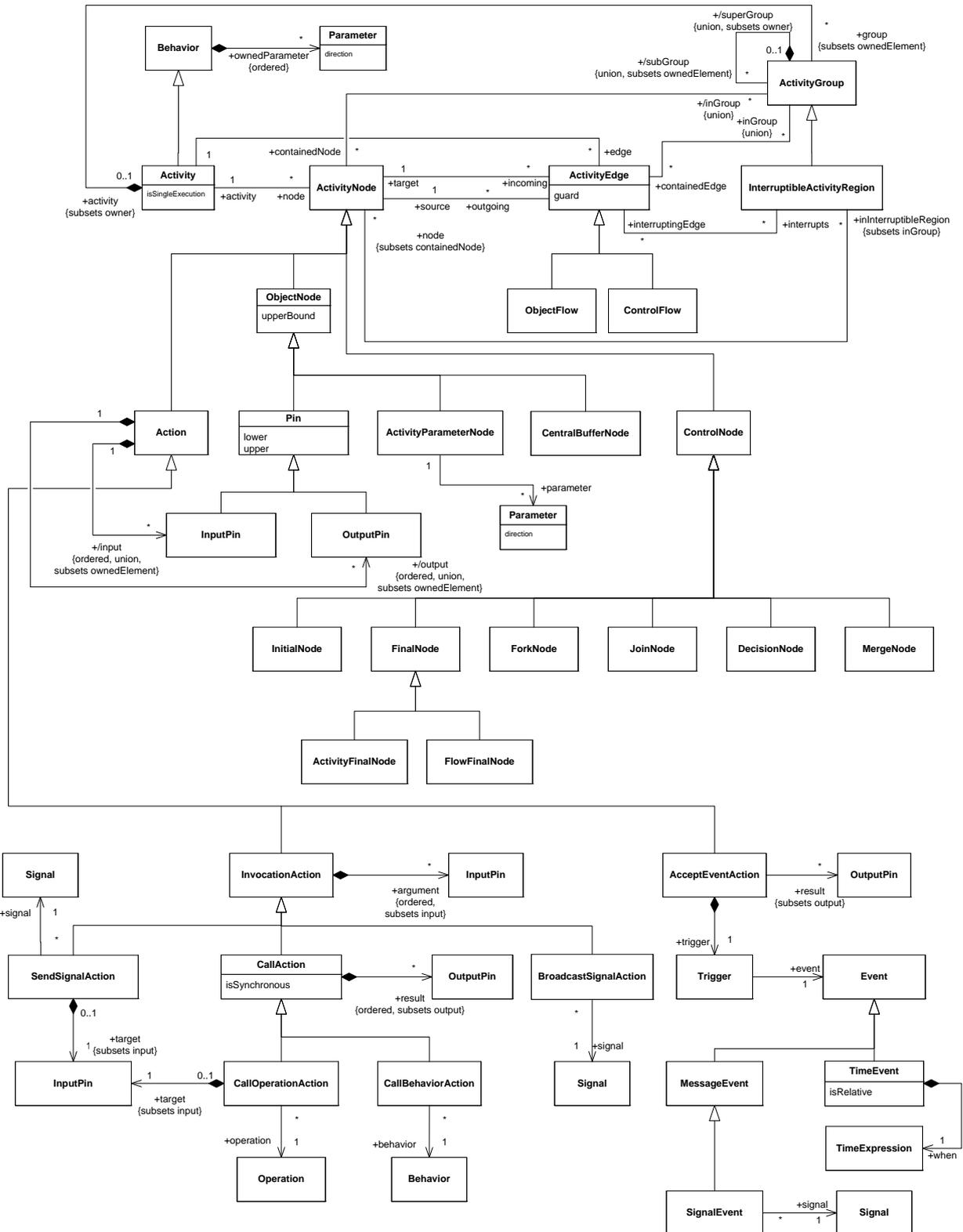


Abbildung A.1: In dieser Arbeit betrachteter Ausschnitt aus dem Metamodell der UML 2.

static domain *Behavior*
static *ownedParameter*: *Behavior* $\rightarrow \mathcal{P}(\text{Parameter})$

static domain *Activity* \subseteq *Behavior*
static *isSingleExecution*: *Activity* \rightarrow *Boolean*
static *node*: *Activity* $\rightarrow \mathcal{P}(\text{ActivityNode})$
static *edge*: *Activity* $\rightarrow \mathcal{P}(\text{ActivityEdge})$
static *group*: *Activity* $\rightarrow \mathcal{P}(\text{ActivityGroup})$

static domain *ActivityNode*
static *incoming, outgoing*: *ActivityNode* $\rightarrow \mathcal{P}(\text{ActivityEdge})$
static *inGroup*: *ActivityNode* $\rightarrow \mathcal{P}(\text{ActivityGroup})$
static *activity*: *ActivityNode* \rightarrow *Activity*
static *inInterruptibleRegion*: *ActivityNode* $\rightarrow \mathcal{P}(\text{InterruptibleActivityRegion})$

static domain *ActivityEdge*
static *guard*: *ActivityEdge* \rightarrow *GuardSpecification*
static *source, target*: *ActivityEdge* \rightarrow *ActivityNode*
static *inGroup*: *ActivityEdge* \rightarrow *ActivityGroup*
static *interrupts*: *ActivityEdge* $\rightarrow \mathcal{P}(\text{InterruptibleActivityRegion})$
static domain *ControlFlow* \subseteq *ActivityEdge*
static domain *ObjectFlow* \subseteq *ActivityEdge*

static domain *Action* \subseteq *ActivityNode*
static *input*: *Action* \rightarrow *InputPin*^{*}
static *output*: *Action* \rightarrow *OutputPin*^{*}

static domain *InvocationAction* \subseteq *Action*
static *argument*: *InvocationAction* \rightarrow *InputPin*^{*}
static domain *CallAction* \subseteq *InvocationAction*
static *isSynchronous*: *CallAction* \rightarrow *Boolean*
static *result*: *CallAction* \rightarrow *OutputPin*^{*}

static domain *CallBehaviorAction* \subseteq *CallAction*
static *behavior*: *CallBehaviorAction* \rightarrow *Behavior*

static domain *CallOperationAction* \subseteq *CallAction*
static *operation*: *CallOperationAction* \rightarrow *Operation*
static *target*: *CallOperationAction* \rightarrow *InputPin*
static domain *Operation*

static domain *Signal*
static domain *Trigger*

static *event*: *Trigger* \rightarrow *Event*
static domain *Event*
static domain *MessageEvent* \subseteq *Event*
static domain *SignalEvent* \subseteq *MessageEvent*
static *signal*: *SignalEvent* \rightarrow *Signal*

static domain *SendSignalAction* \subseteq *InvocationAction*
static *signal*: *SendSignalAction* \rightarrow *Signal*
static *target*: *SendSignalAction* \rightarrow *InputPin*
static domain *BroadcastSignalAction* \subseteq *InvocationAction*
static *signal*: *BroadcastSignalAction* \rightarrow *Signal*

static domain *AcceptEventAction* \subseteq *Action*
static *trigger*: *AcceptEventAction* \rightarrow $\mathcal{P}(\textit{Trigger})$
static *result*: *AcceptEventAction* \rightarrow *OutputPin**

static domain *ControlNode* \subseteq *ActivityNode*
static domain *DecisionNode* \subseteq *ControlNode*
static domain *MergeNode* \subseteq *ControlNode*
static domain *ForkNode* \subseteq *ControlNode*
static domain *InitialNode* \subseteq *ControlNode*
static domain *FinalNode* \subseteq *ControlNode*
static domain *ActivityFinalNode* \subseteq *FinalNode*
static domain *FlowFinalNode* \subseteq *FinalNode*

static domain *ObjectNode* \subseteq *ActivityNode*
static *upperBound*: *ObjectNode* \rightarrow *Nat*
static domain *CentralBufferNode* \subseteq *ObjectNode*
static domain *ActivityParameterNode* \subseteq *ObjectNode*
static *parameter*: *ActivityParameterNode* \rightarrow *Parameter*
static domain *ParameterDirectionKind* =_{def} {in, inout, out, return}
static domain *Parameter*
static *direction* : *Parameter* \rightarrow *ParameterDirectionKind*

static domain *Pin* \subseteq *ObjectNode*
static *lower*: *Pin* \rightarrow *Nat*
static *upper*: *Pin* \rightarrow *Nat*
static domain *InputPin* \subseteq *Pin*
static domain *OutputPin* \subseteq *Pin*

static domain *ActivityGroup*
static *superGroup*: *ActivityGroup* \rightarrow *ActivityGroup*
static *subGroup*: *ActivityGroup* \rightarrow $\mathcal{P}(\textit{ActivityGroup})$
static *containedNode*: *ActivityGroup* \rightarrow $\mathcal{P}(\textit{ActivityNode})$

static *containedEdge*: *ActivityGroup* $\rightarrow \mathcal{P}(\text{ActivityEdge})$

static *activity*: *ActivityGroup* $\rightarrow \text{Activity}$

static domain *InterruptibleActivityRegion* $\subseteq \text{ActivityGroup}$

static *node*: *InterruptibleActivityRegion* $\rightarrow \mathcal{P}(\text{ActivityNode})$

static *interruptingEdge*: *InterruptibleActivityRegion* $\rightarrow \mathcal{P}(\text{ActivityEdge})$

A.1.3 Datenstrukturen zur Transformation

Die folgenden Datenstrukturen sind nicht aus dem Metamodell generiert. Sie werden für den Übersetzungsalgorithmus benötigt.

Die folgenden Domänen bilden Klassen, ihre Attribute und ausführbaren Code ab. Die dazugehörigen Funktionen erlauben es, die Kontextklasse einer Aktivität zu finden und auf deren Attribute bzw. Methoden zuzugreifen.

static domain *Classifier*

static domain *Attribute*

static domain *Code*

static *getContextClass*: *Activity* $\rightarrow \text{Classifier}$

static *attributes* : *Classifier* $\rightarrow \mathcal{P}(\text{Attribute})$

static *locateCode*: *Classifier* $\times \text{Operation}$ $\rightarrow \text{Code}$

Zur Abstraktion von einem konkreten Model Checker werden die folgenden Domänen definiert. Sie beschreiben abstrakt eine Guardspezifikation, die Eingabesprache für den Model Checker und die möglichen Modi der Aktionen. Die Menge der möglichen Modi wird ebenfalls festgelegt.

static domain *GuardSpecification*

static domain *ModelCheckerInput*

static domain *NodeMode*

initially *NodeMode* = {inactive, running, offeringToken}

Im folgenden Abschnitt werden die semantischen Variationspunkte und eine Funktion für ihren Zugriff definiert.

static domain *Element* $=_{def} \text{Behavior} \cup \text{ActivityGroup} \cup \text{ActivityEdge} \cup \text{ActivityNode}$

static domain *ValueSpecification*

static domain *Stereotype*

initially *Stereotype* = {StartAction, SignalBuffering, IARHandling}

static domain *Tag*

initially *Tag* = {buffer, replace, ignoreFlowIntoInterruptedRegion, regionActivationPolicy, regionReactivationPolicy, initialStart}

static *tagValue*: *Element* $\times \text{Stereotype} \times \text{Tag}$ $\rightarrow \text{ValueSpecification}$

Die Datenstruktur *Flow* nimmt die berechneten möglichen Flüsse auf. Eine genaue Beschreibung findet sich in Abschnitt 3.2.4.1.

controlled domain *Flow* $=_{def} \{$

```

actionsOffering:  $\mathcal{P}(\text{Action})$ ;
forksOffering:  $\mathcal{P}(\text{ActivityNode} \times \text{ActivityEdge})$ ;
objectNodesOffering:  $\mathcal{P}(\text{ObjectNode})$ ;
guards:  $\mathcal{P}(\text{GuardSpecification})$ ;

actionsStart:  $\mathcal{P}(\text{Action})$ ;
forksBuffering:  $\mathcal{P}(\text{ActivityNode} \times \text{ActivityEdge})$ ;
objectNodesBuffering:  $\mathcal{P}(\text{ObjectNode})$ ;

regionsEntered:  $\mathcal{P}(\text{InterruptibleActivityRegion})$ ;
regionsInterrupted:  $\mathcal{P}(\text{InterruptibleActivityRegion})$ ;

dataFlows:  $\mathcal{P}(\text{ActivityNode} \times \text{ActivityNode})$ 
}

```

A.2 Makros zur Flussberechnung

Auf eine ausführliche Erläuterung der folgenden Makros wird hier verzichtet, da diese ausführlich in Abschnitt 3.2.4 beschrieben werden.

Das Makro **COMPFLAWS** wird am Ende des Abschnitts 3.2.4.4 näher erläutert.

```

COMPFLAWS : void  $\rightarrow \mathcal{P}(\text{Flow})$ 
COMPFLAWS  $\equiv$ 
  local flows :  $\mathcal{P}(\text{Flow}) = \emptyset$ 
  CREATEFORKEDGEBUFFERS
  CALCULATEPOSSIBLEDATASOURCES
  seq
  flows := COMPSIMPLEFLAWS
  seq
  flows := CALCULATEMULTIDATAFLAWS(flows)
  seq
  flows := CALCULATEFORKSCOMBINEDFLAWS(flows)
  seq
  flows := REMOVENOTNECESSARYFORKS(flows)
  seq
  flows := removeFlowsWithNoEffect(flows)
  seq
  forall f with f  $\in$  flows
    REDUCEDATAFLAWS(f)
  seq
  result := flows

```

Die folgenden drei Makros werden ausführlich in Abschnitt 3.2.4.2 dargestellt.

```

controlled forkEdgeToObjectNode : ActivityNode  $\times$  ActivityEdge  $\rightarrow$  ObjectNode
CREATEFORKEDGEBUFFERS  $\equiv$ 
  forall n with n  $\in$  ForkNodes
    forall e with e  $\in$  n.outgoing

```

```

let
  o = new(ObjectNode)
  o.inInterruptibleRegion = n.inInterruptibleRegion
in
  forkEdgeToObjectNode(n, e) := o
forall n with n ∈ Action
if multipleOutFlows(n) ∧ |n.outgoing| = 1 then
  let
    o = new(ObjectNode)
    o.inInterruptibleRegion = n.inInterruptibleRegion
  in
    forkEdgeToObjectNode(n, outgoing(n, 1)) := o

```

controlled *possibleDataSources* : *ActivityNode* → $\mathcal{P}(\textit{ObjectNode})$

CALCULATEPOSSIBLEDATASOURCES ≡

```

forall p with p ∈ InputPin ∪ CentralBufferNode ∪ ActivityParameterNode
  forall e with e ∈ p.incoming
  let
    r = CALCULATEPOSSIBLEDATASOURCESREC(e)
  in
    possibleDataSources(p) := possibleDataSources(p) ∪ r

```

CALCULATEPOSSIBLEDATASOURCESREC : *ActivityEdge* → $\mathcal{P}(\textit{ObjectNode})$

CALCULATEPOSSIBLEDATASOURCESREC(*edge*) ≡

```

local r :  $\mathcal{P}(\textit{ObjectNode})$ 
let
  node = edge.source
in
  if node ∈ OutputPin ∪ CentralBufferNode ∪ ActivityParameterNode then
    add node to r
  if node ∈ ForkNode then
    let
      res = forkEdgeToObjectNode(node, edge)
    in
      add res to r
  if node ∈ MergeNode ∪ JoinNode ∪ DecisionNode then
    forall e with e ∈ node.incoming
    let
      res = CALCULATEPOSSIBLEDATASOURCESREC(e)
    in
      add res to r
seq
result := r

```

Die Makros ab hier sind in Abschnitt 3.2.4.3 beschrieben.

```

COMP SIMPLE FLOWS : void →  $\mathcal{P}(\text{Flow})$ 
COMP SIMPLE FLOWS ≡
  local r :  $\mathcal{P}(\text{Flow}) := \emptyset$ 
  forall a with a in Activity
    forall n with n in a.node
      if n ∈ InitialNode then
        let
          simpleFlows = COMP SIMPLE FLOW FOR INITIAL NODE(n)
        in
          add simpleFlows to r
      if n ∈ Action then
        let
          simpleFlows = COMP SIMPLE FLOW FOR ACTION(n)
        in
          add simpleFlows to r
      if n ∈ CentralBufferNode ∪ ActivityParameterNode then
        let
          simpleFlows = COMP SIMPLE FLOW FOR OBJECT NODE(n)
        in
          add simpleFlows to r
  seq
  result := r

```

```

COMP SIMPLE FLOW FOR INITIAL NODE : InitialNode →  $\mathcal{P}(\text{Flow})$ 
COMP SIMPLE FLOW FOR INITIAL NODE(node) ≡
  let
    flow = new(Flow)
    edge = outgoing(node, 1)
  in
    add node to flow.actionsOffering
  seq
  result := FLOW DOWN(edge, flow)

```

```

COMP SIMPLE FLOW FOR OBJECT NODE : ObjectNode →  $\mathcal{P}(\text{Flow})$ 
COMP SIMPLE FLOW FOR OBJECT NODE(node) ≡
  local r :  $\mathcal{P}(\text{Flow}) := \emptyset$ 
  forall e with e ∈ node.outgoing
    let
      flow = new(Flow)
    in
      add node to flow.objectNodesOffering
      add (node, undef) to flow.dataFlows
  seq
  let

```

```

    s = FLOWDOWN(e, flow)
  in
    add s to r
seq
result := r

```

COMPSIMPLEFLOWFORACTION : *Action* → $\mathcal{P}(\textit{Flow})$

COMPSIMPLEFLOWFORACTION(*node* ≡

if *multipleOutFlows*(*node*) then

result ← COMPSIMPLEFLOWFORACTIONWITHMULTIPLEFLOWS

else

if *node.outgoing* ≠ ∅ then

let

flow = *new*(*Flow*)

edge = *outgoing*(*node*,1)

in

add *node* to flow.*actionsOffering*

seq

result := FLOWDOWN(edge, flow)

if |*node.output*| = 1 then

local *r* : $\mathcal{P}(\textit{Flow})$:= ∅

let

o = *output*(*node*, 1)

in

forall e with e ∈ o.*outgoing*

let

flow = *new*(*Flow*)

in

add *node* to flow.*actionsOffering*

add (o, undef) to flow.*dataFlows*

seq

let

s = FLOWDOWN(e, flow)

in

add s to *r*

seq

result := *r*

COMPSIMPLEFLOWFORACTIONWITHMULTIPLEFLOWS : *Action* → $\mathcal{P}(\textit{Flow})$

COMPSIMPLEFLOWFORACTIONWITHMULTIPLEFLOWS(*node*) ≡

local *res* : $\mathcal{P}(\textit{Flow})$:= ∅

let

bufferFlow = *new*(*Flow*)

in

add *node* to bufferFlow.*actionsOffering*

if *node.outgoing* ≠ ∅ then

```

    add (node, outgoing(node,1)) to bufferFlow.forksBuffering
  forall o with o in node.output
    forall e with e in o.outgoing
      add (o, e) to bufferFlow.forksBuffering
  seq
  add bufferFlow to res
  if node.outgoing ≠ ∅ then
    let
      flow = new(Flow)
    in
      add (node, outgoing(node,1)) to flow.forksBuffering
      seq
      let
        r = FLOWDOWN(outgoing(node,1), flow)
      in
        add r to res
  forall o with o in node.output
    local r : P(Flow) := ∅
    forall e with e in o.outgoing
      let
        flow = new(Flow)
      in
        add (o, e) to flow.forksOffering
        add (o, undef) to flow.dataFlows
        seq
        let
          s = FLOWDOWN(e, flow)
        in
          add s to r
    seq
  add r to res
  seq
  result := res

```

FLOWDOWN : $ActivityEdge \times Flow \rightarrow \mathcal{P}(Flow)$

FLOWDOWN(edge, flow) \equiv

local $r : \mathcal{P}(Flow) := \emptyset$

let

node : $ActivityNode = edge.target$

diffRegions : $\mathcal{P}(InterruptibleActivityRegion) =$

$edge.target.inInterruptibleRegion \setminus edge.source.inInterruptibleRegion$

in

add edge.guard to flow.guards

forall region with region $\in edge.interrupts$

add region to flow.regionsInterrupted

add diffRegions to flow.regionsEntered

```

if node  $\in$  FinalNode then
  add node to flow.actionsStart
  add flow to r
if node  $\in$  Action then
  r := ACTIONFLOWDOWN(node, flow)
if node  $\in$  ObjectNode then
  r := OBJECTNODEFLOWDOWN(node, flow)
if node  $\in$  MergeNode then
  r := FLOWDOWN(outgoing(node,1), flow)
if node  $\in$  DecisionNode then
  r := DECISIONFLOWDOWN(node, flow)
if node  $\in$  ForkNode then
  r := FORKFLOWDOWN(node, flow)
if node  $\in$  JoinNode then
  r := JOINFLOWDOWN(node, edge, flow)
seq
result := r

```

ACTIONFLOWDOWN : *ActivityNode* \times *Flow* \rightarrow $\mathcal{P}(\textit{Flow})$

ACTIONFLOWDOWN(node, flow) \equiv

```

local r :  $\mathcal{P}(\textit{Flow})$  :=  $\emptyset$ 
if node.input =  $\emptyset$  then
  add node to flow.actionsStart
  seq
  add flow to r
else
  let
    newFlow = CLONEFLOW(flow)
  in
    add node to newFlow.actionsStart
    seq
    r := {newFlow}
  foreach p with p  $\in$  node.input
    foreach e with e  $\in$  p.incoming
      let
        newFlow = CLONEFLOW(flow)
      in
        if newFlow.dataFlows =  $\emptyset$  then
          add (undef, p) to newFlow.dataFlows
        else
          first(asList(newFlow.dataFlows)).Second := p
        seq
        let
          upResult = FLOWUP(e, newFlow)
        in
          r := ELEMENTWISEUNION(r, upResult)

```

```

seq
result := r

```

OBJECTNODEFLOWDOWN : $ActivityNode \times Flow \rightarrow \mathcal{P}(Flow)$

OBJECTNODEFLOWDOWN(*node*, *flow*) \equiv

```

local r :  $\mathcal{P}(Flow)$  :=  $\emptyset$ 
if node  $\in Pin$  then
  let
    owner = node.owner
  in
  if multipleInFlows(owner) then
    let
      newFlow = CLONEFLOW(flow)
    in
    add owner to newFlow.actionsStart
    first(asList(newFlow.dataFlows)).Second := node
    seq
    r := {newFlow}
  seq
  foreach p with p  $\in$  owner.input
  if p  $\neq$  node then
    foreach e with e  $\in$  p.incoming
    let
      newFlow = CLONEFLOW(flow)
    in
    first(asList(newFlow.dataFlows)).Second := p
    seq
    let
      upResult = FLOWUP(e, newFlow)
    in
    r := ELEMENTWISEUNION(r, upResult)
  if owner.incoming  $\neq$   $\emptyset$  then
    let
      newFlow = CLONEFLOW(flow)
    in
    first(asList(newFlow.dataFlows)).Second := p
    seq
    let
      upResult = FLOWUP(incoming(owner, 1), newFlow)
    in
    r := ELEMENTWISEUNION(r, upResult)
  else
    add owner to flow.actionsStart
    first(asList(flow.dataFlows)).Second := node
    seq
    add flow to r

```

```

else
  add node to flow.objectNodesBuffering
  first(asList(flow.dataFlows)).Second := node
  seq
  add flow to r
seq
result := r

```

DECISIONFLOWDOWN : *ActivityNode* × *Flow* → $\mathcal{P}(\textit{Flow})$

DECISIONFLOWDOWN(*node*, *flow*) ≡

```

local r :  $\mathcal{P}(\textit{Flow})$  :=  $\emptyset$ 
local newFlow : Flow :=  $\emptyset$ 
foreach e with e ∈ node.outgoing
  if |flow.dataFlows| > 1 then
    foreach n1, n2 with (n1, n2) ∈ flow.dataFlows
      newFlow := CLONEFLOW(flow)
      seq
      newFlow.dataFlows := {(n1, n2)}
      seq
      let
        decisionResult :  $\mathcal{P}(\textit{Flow})$  = FLOWDOWN(e, newFlow)
      in
        r := r ∪ decisionResult
    else
      newFlow := CLONEFLOW(flow)
      seq
      let
        decisionResult :  $\mathcal{P}(\textit{Flow})$  = FLOWDOWN(e, newFlow)
      in
        r := r ∪ decisionResult
seq
result := r

```

FORKFLOWDOWN : *ActivityNode* × *Flow* → $\mathcal{P}(\textit{Flow})$

FORKFLOWDOWN(*node*, *flow*) ≡

```

local r :  $\mathcal{P}(\textit{Flow})$  :=  $\emptyset$ 
let
  bufferFlows = CREATEFLOWCOMBINATIONSFORFORKNODE(node, flow, 1)
in
  add bufferFlows to r
foreach e with e ∈ node.outgoing
  let
    newFlow : Flow = new(Flow)
    onode : ObjectNode = forkEdgeToObjectNodes(node, e)
  in
    add (node, e) to newFlow.forksOffering

```

```

if  $e \in \text{ObjectFlow}$  then
  add (onode, undef) to newFlow.dataFlows
  seq
  let
     $s = \text{FLOWDOWN}(e, \text{newFlow})$ 
  in
    add  $s$  to  $r$ 
seq
result :=  $r$ 

```

$\text{JOINFLOWDOWN} : \text{ActivityNode} \times \text{ActivityEdge} \times \text{Flow} \rightarrow \mathcal{P}(\text{Flow})$

```

JOINFLOWDOWN(node, edge, flow)  $\equiv$ 
  local  $r : \mathcal{P}(\text{Flow}) := \emptyset$ 
  local  $\text{upResults} : \mathcal{P}(\text{Flow}) := \emptyset$ 
  foreach  $e$  with  $e \in \text{node.incoming}$ 
    if  $e \neq \text{edge}$ 
      let
        newFlow = CLONEFLOW(flow)
        joinResult = FLOWUP(e, newFlow)
      in
         $\text{upResults} := \text{ELEMENTWISEUNION}(\text{upResults}, \text{joinResult})$ 
  seq
  foreach downFlow with downFlow  $\in \text{CLONEFLOWS}(\text{upResults})$ 
    let
      joinResult = FLOWDOWN(outgoing(node,1), downFlow)
    in
      add joinResult to  $r$ 
  seq
  result :=  $r$ 

```

$\text{FLOWUP} : \text{ActivityEdge} \times \text{Flow} \rightarrow \mathcal{P}(\text{Flow})$

```

FLOWUP(edge, flow)  $\equiv$ 
  local  $r : \mathcal{P}(\text{Flow}) := \emptyset$ 
  let
    node :  $\text{ActivityNode} = \text{edge.source}$ 
    diffRegions :  $\mathcal{P}(\text{InterruptibleActivityRegion}) =$ 
       $\text{edge.target.inInterruptibleRegion} \setminus \text{edge.source.inInterruptibleRegion}$ 
  in
    add edge.guard to flow.guards
    forall region with region  $\in \text{edge.interrupts}$ 
      add region to flow.regionsInterrupted
    add diffRegions to flow.regionsEntered

    if node  $\in \text{InitialNode}$  then
      add node to flow.actionsOffering
      add flow to  $r$ 

```

```

if node  $\in$  Action then
  if multipleOutFlows(node) then
    add (node, edge) to flow.forksOffering
  else
    add node to flow.actionsOffering
  add flow to r

if node  $\in$  CentralBufferNode  $\cup$  ActivityParameterNode then
  add node to flow.objectNodesOffering
  first(asList(flow.dataFlows)).First := node
  add flow to r

if node  $\in$  OutputPin then
  r := OUTPUTPINFLOWUP(node, edge, flow)

if node  $\in$  DecisionNode then
  r := FLOWUP(incoming(node, 1), flow)

if node  $\in$  MergeNode then
  r := MERGEFLOWUP(node, flow)

if node  $\in$  ForkNode then
  r := FORKFLOWUP(node, edge, flow)

if node  $\in$  JoinNode then
  r := JOINFLOWUP(node, flow)
seq
result := r

```

Das folgende Makro wird hier kurz beschrieben, da es im Text nicht ausführlich erläutert wird. Ein *OutputPin* bei der Verfolgung eines Flusses entgegen seiner Richtung stellt ein Ende der Rekursion dar. Wenn noch keine Datensinken im Fluss enthalten sind, wird ein neuer Datenfluss mit dem *OutputPin* als Datenquelle erzeugt. Anderenfalls wird dieser *Pin* dem existierenden Datenfluss hinzugefügt.

Wenn der *OutputPin* Teil eines impliziten Forks ist, wird der Knoten der Menge *forksOffering* hinzugefügt, anderenfalls der Menge *actionsOffering*.

```

OUTPUTPINFLOWUP : ActivityNode  $\times$  ActivityEdge  $\times$  Flow  $\rightarrow$   $\mathcal{P}(\textit{Flow})$ 
OUTPUTPINFLOWUP(node, edge, flow)  $\equiv$ 
  local r :  $\mathcal{P}(\textit{Flow})$  :=  $\emptyset$ 
  let
    owner = node.owner
  in
    if flow.dataFlows =  $\emptyset$  then
      add (undef, node) to flow.dataFlows
    else

```

```

    first(asList(flow.dataFlows)).First := node
  if multipleOutFlows(owner) then
    add (node, edge) to flow.forksOffering
  else
    add node to flow.actionsOffering
  result := r

```

Das Makro **MERGEFLOWUP** erzeugt für jede eingehende Kante einen neuen Fluss und ruft das Makro **FLOWUP** rekursiv auf.

```

MERGEFLOWUP : ActivityNode × Flow → P(Flow)
MERGEFLOWUP(node, flow) ≡
  local r : P(Flow) := ∅
  foreach e with e ∈ node.incoming
    let
      newFlow = CLONEFLOW(flow)
      upResult = FLOWUP(e, newFlow)
    in
      r := r ∪ upResult
  seq
  result := r

```

```

FORKFLOWUP : ActivityNode × ActivityEdge × Flow → P(Flow)
FORKFLOWUP(node, edge, flow) ≡
  local r : P(Flow) := ∅
  if edge ∈ ObjectFlow then
    if flow.dataFlows = ∅ then
      add (forkEdgeToObjectNode(node, edge), undef) to flow.dataFlows
    else
      first(asList(flow.dataFlows)).First := forkEdgeToObjectNode(node, edge)
  add (node, edge) to flow.forksOffering
  result := r

```

```

JOINFLOWUP : ActivityNode × Flow → P(Flow)
JOINFLOWUP(node, flow) ≡
  local r : P(Flow) := ∅
  foreach e with e ∈ node.incoming
    let
      newFlow = CLONEFLOW(flow)
      upResult = FLOWUP(e, newFlow)
    in
      r := ELEMENTWISEUNION(r, upResult)
  seq
  result := r

```

Die folgenden Makros sind für die Berechnung der kombinierten Flüsse zuständig. Sie werden

(bis auf einige Ausnahmen) in Abschnitt 3.2.4.4 beschrieben.

Das Makro `CALCULATEMULTIDATAFLOWS` wird im Text nicht ausführlich beschrieben. Daher erfolgt hier eine kurze Erläuterung.

Die externe Funktion `combinations` liefert alle Kombinationen n-elementiger Mengen aus einer Gesamtmenge mit k Elementen ohne Reihenfolge und ohne Zurücklegen als Liste von Listen zurück. n ist dabei der erste Parameter, k der zweite. Für `combinations(3, 4)` wird also `[[1,2,3], [1,2,4], [1,3,4], [2,3,4]]` erzeugt.

Diese Funktion wird im Makro dazu benutzt, alle möglichen Kombinationen von eingehenden Datenflüssen für einen `InputPin` mit einem upper-Wert größer Eins zu bestimmen. Dazu werden für alle Knoten, die von mehr als einer Datenquelle gespeist werden und größere upper-Werte besitzen, alle Kombinationen der Länge 2 bis zum upper-Wert erzeugt und für diese über eine Abbildung auf die Liste der möglichen Datenquellen neue Flüsse erzeugt.

```

monitored combinations :  $Nat \times Nat \rightarrow (Nat^*)^*$ 
CALCULATEMULTIDATAFLOWS :  $\mathcal{P}(Flow) \rightarrow \mathcal{P}(Flow)$ 
CALCULATEMULTIDATAFLOWS(flows)  $\equiv$ 
  local r :  $\mathcal{P}(Flow) = \emptyset$ 
  local max :  $Nat$ 
  local k :  $Nat$ 
  forall n with |possibleDataSources(n)| > 1  $\wedge$ 
    ((n  $\in$  CentralBuffer  $\cup$  ActivityParameterNode  $\wedge$  n.upperBound  $>$  1)  $\vee$ 
     (n  $\in$  Pin  $\wedge$  n.upper  $>$  1))
  let
    oFlows = possibleDataSources(n)
  in
    if n  $\in$  Pin then
      max := min(n.upper, |oFlows|)
    else
      max := min(n.upperBound, |oFlows|)
      k := 2
    seq
      iterate
        if k  $>$  max then
          skip
        else
          k := k + 1
        let
          combinations :  $(Nat^*)^* = \text{combinations}(|oFlows|, k)$ 
          newFlow = new(Flow)
          inputList :  $ObjectNode^* = \text{asList}(\text{possibleDataSources}(n))$ 
        in
          foreach c with c  $\in$  combinations
            foreach pos with pos  $\in$  c
              let
                foundFlows = GETFLOWSWITHOBJECTFLOWSOURCE
                  (flows, elementAt(inputList, pos), n)
              in

```

```

    foreach f with f ∈ foundFlows
        newFlow := COMBINEFLOWS(newFlow, f)
    seq
    add newFlow to r
seq
result := r

```

Dieses Makro liefert aus der Menge der übergebenen Flüsse diejenigen, die einen Datenfluss von dem ersten übergebenen *ObjectNode* zum zweiten haben. Dazu werden einfach die entsprechenden Mengen aller Flüsse durchsucht und die erzeugte Menge zurückgeliefert.

GETFLOWSWITHOBJECTFLOWSOURCE : $\mathcal{P}(\text{Flow}) \times \text{ObjectNode} \times \text{ObjectNode} \rightarrow \mathcal{P}(\text{Flow})$

GETFLOWSWITHOBJECTFLOWSOURCE(*flows*, *offerKey*, *bufferKey*) \equiv

```

local r :  $\mathcal{P}(\text{Flow}) = \emptyset$ 
forall f with f ∈ flows
    if offerKey ∈ f.objectNodesOffering ∧
        bufferKey ∈ f.objectNodesBuffering then
        add f to r
    forall n1, n2 with (n1, n2) ∈ f.dataFlows
        if n1 = offerKey ∧ n2 = bufferKey then
            add f to r
seq
result := r

```

CALCULATEFORKSCOMBINEDFLOWS : $\mathcal{P}(\text{Flow}) \rightarrow \mathcal{P}(\text{Flow})$

CALCULATEFORKSCOMBINEDFLOWS(*flows*) \equiv

```

local forkCombinations :  $\mathcal{P}(\text{Flow}) := \text{CLONEFLOWS}(\text{flows})$ 
iterate
    forkCombinations := forkCombinations ∪ MIXFORKS(forkCombinations, flows)
seq
result := forkCombinations ∪ flows

```

MIXFORKS : $\mathcal{P}(\text{Flow}) \times \mathcal{P}(\text{Flow}) \rightarrow \mathcal{P}(\text{Flow})$

MIXFORKS(*flows*, *poolFlows*) \equiv

```

local r :  $\mathcal{P}(\text{Flow}) = \emptyset$ 
forall f with f ∈ flows
    forall n, e with (n, e) ∈ f.forksOffering
        forall pool with pool ∈ poolFlows
            forall n2, e2 with (n2, e2) ∈ pool.forksBuffering
                if n = n2 ∧ e = e2 then
                    let
                        newFlow = COMBINEFLOWS(f, pool)
                    in
                        if CHECKDATAFLOWS(newFlow) then
                            REDUCEELEMENTS(newFlow)
                seq

```

```

        add newFlow to r
    forall n, e with (n, e) ∈ f.forksBuffering
        forall pool with pool ∈ poolFlows
            forall n2, e2 with (n2, e2) ∈ pool.forksOffering
                if n = n2 ∧ e = e2 then
                    let
                        newFlow = COMBINEFLOWS(f, pool)
                    in
                        if CHECKDATAFLOWS(newFlow) then
                            REDUCEELEMENTS(newFlow)
                        seq
                            add newFlow to r
    seq
    result := r

```

Fork-Pufferung ist nur notwendig, wenn es sich entweder um einen Objektfluss handelt oder ein Kontrollfluss von einem *JoinNode* gefolgt wird. Daher eliminiert folgendes Makro alle Flüsse, die nicht dieser Eigenschaft entsprechen.

```

REMOVE NOT NECESSARY FORKS :  $\mathcal{P}(\text{Flow}) \rightarrow \mathcal{P}(\text{Flow})$ 
REMOVE NOT NECESSARY FORKS(flows) ≡
    local r :  $\mathcal{P}(\text{Flow}) = \text{CLONEFLOWS}(\text{flows})$ 
    forall f with f ∈ flows
        forall n, e with (n, e) ∈ f.forksBuffering
            if n ∈ ForkNode ∧ e ∈ ControlFlow ∧ ¬ISFOLLOWEDBYJOIN(e) then
                remove f from r
        forall n, e with (n, e) ∈ f.forksOffering
            if n ∈ ForkNode ∧ e ∈ ControlFlow ∧ ¬ISFOLLOWEDBYJOIN(e) then
                remove f from r
    seq
    result := r

```

Flüsse, die nichts bewirken (deren Flusseffekt also leer ist), werden durch folgende Funktion entfernt.

```

derived removeFlowsWithNoEffect :  $\mathcal{P}(\text{Flow}) \rightarrow \mathcal{P}(\text{Flow})$ 
removeFlowsWithNoEffect(flows) =def
    {f ∈ flows | f.actionsStart ≠ ∅ ∨ f.forksBuffering ≠ ∅ ∨ f.objectNodesBuffering ≠ ∅}

```

A.3 Makros zur Zustandsübergangsbestimmung

Die Formalisierung der Zustandsübergangsbestimmung findet sich zusammengefasst in diesem Abschnitt. Eine Beschreibung erfolgt in Abschnitt 3.2.5.

```

CREATE TRANSITIONS :  $\mathcal{P}(\text{Flow}) \rightarrow \text{ModelCheckerInput}^*$ 
CREATE TRANSITIONS(flows) ≡
    local r : ModelCheckerInput* = []

```

```

r := r  $\uplus$  CREATEPREAMBLE
seq
r := r  $\uplus$  CREATETRANSITIONSRUNNINGTOOFFERING
seq
r := r  $\uplus$  CREATEFINISHCHECKS
seq
r := r  $\uplus$  CREATEFLOWTRANSITIONS(flows)
seq
result := r

```

CREATEPREAMBLE : *void* \rightarrow *ModelCheckerInput**

```

CREATEPREAMBLE  $\equiv$ 
  local r : ModelCheckerInput* = []
  r := r  $\oplus$  DEFINEMODES
  seq
  foreach c with c  $\in$  Classifier
    r := r  $\uplus$  DEFINECLASSATTRIBUTES(c)
  seq
  r := r  $\uplus$  DEFINENODEVARIABLES
  seq
  r := r  $\uplus$  DEFINESIGNALS
  seq
  result := r

```

DEFINECLASSATTRIBUTES : *Classifier* \rightarrow *ModelCheckerInput**

```

DEFINECLASSATTRIBUTES(c)  $\equiv$ 
  local r : ModelCheckerInput* = []
  foreach f with f  $\in$  attributes(c)
    r := r  $\oplus$  DEFINEATTRIBUTE(f)
  seq
  result := r

```

DEFINENODEVARIABLES : *void* \rightarrow *ModelCheckerInput**

```

DEFINENODEVARIABLES  $\equiv$ 
  local r : ModelCheckerInput* = []
  local m : NodeMode = inactive
  foreach n with n  $\in$  ActivityNode
    if n  $\in$  InitialNode then
      if tagValue(n.activity, StartAction, initalStart) = true then
        r := r  $\oplus$  DEFINENODEVARIABLE(n, offeringToken)
      else
        r := r  $\oplus$  DEFINENODEVARIABLE(n, inactive)

    if n  $\in$  Action then
      if predecessors(n) =  $\emptyset$   $\wedge$  n.inInterruptibleRegion =  $\emptyset$   $\wedge$ 

```

```

    tagValue(n.activity, StartAction, initialStart) = Yes then
    m := running
else
    m := inactive
seq
r := r  $\oplus$  DEFINE NODE VARIABLE(n, m)
seq
if n  $\in$  AcceptEventAction then
    r := r  $\oplus$  DEFINE AE A BUFFER VARIABLE(n, inactive)

if n  $\in$  ObjectNode
    r := r  $\oplus$  DEFINE LIST(n)
seq
result := r

```

DEFINESIGNALS : void \rightarrow ModelCheckerInput*

DEFINESIGNALS \equiv

```

local r : ModelCheckerInput* = []
foreach s with s  $\in$  Signal
    r := r  $\oplus$  DEFINE SIGNAL(s)
seq
foreach a with a  $\in$  Activity
    r := r  $\oplus$  DEFINE SIGNAL LIST(a)
seq
result := r

```

Zur Beschreibung der folgenden zwei Makros siehe Abschnitt 3.2.5.2.

CREATEFLOWTRANSITIONS : $\mathcal{P}(\text{Flow}) \rightarrow \text{ModelCheckerInput}^*$

CREATEFLOWTRANSITIONS(flows) \equiv

```

local r : ModelCheckerInput* = []
local nodesInterrupted :  $\mathcal{P}(\text{ActivityNode}) = \emptyset$ 
local nodesEntered :  $\mathcal{P}(\text{ActivityNode}) = \emptyset$ 
local activityfinal : Boolean = false
r := r  $\oplus$  CREATE STEP HEADER
foreach flow with flow  $\in$  flows
    foreach a with a  $\in$  flow.actionsOffering
        if a  $\in$  AcceptEventAction then
            r := r  $\oplus$  CREATE AE A MODE PRECONDITION(a, offeringToken)
        else
            r := r  $\oplus$  CREATE MODE PRECONDITION(a, offeringToken)
    seq
    foreach f with f  $\in$  flow.forksOffering
        r := r  $\oplus$  CREATE LIST PRECONDITION(forkEdgeToObjectNode(f.First, f.Second), false)
    seq
    foreach o with o  $\in$  flow.objectNodesOffering
        r := r  $\oplus$  CREATE LIST PRECONDITION(o, false)

```

```

seq
foreach g with g ∈ flow.guards
  r := r ⊕ CREATEGUARDPRECONDITION(g)
seq
foreach o with o ∈ flow.objectNodesBuffering
  r := r ⊕ CREATEBOUNDPRECONDITION(o, 1)
seq
forall n1, n2 with (n1, n2) ∈ flow.dataFlows
  if dataSinkSum(n2) = undef then
    dataSinkSum(n2) = 1
  else
    dataSinkSum(n2) = dataSinkSum(n2) + 1
seq
foreach o with o ∈ ObjectNode
  if dataSinkSum(o) ≠ undef then
    r := r ⊕ CREATEBOUNDPRECONDITION(o, dataSinkSum(o))
forall o with o ∈ ObjectNode
  dataSinkSum(o) = undef
seq

nodesInterrupted := {n ∈ ActivityNode | ∃r ∈ n.inInterruptibleRegion : r ∈ flow.regionsInterrupted}
nodesEntered := {n ∈ AcceptEventAction | predecessors(n) = ∅ ∧
  immediateRegion(n) ∈ flow.regionsEntered}

seq
forall region with region ∈ flow.regionsInterrupted
  if tagValue(region, IARHandling, ignoreFlowIntoInterruptedRegion) = false then
    forall n with n ∈ flow.actionsStart ∧ region ∈ n.inInterruptibleRegion
      remove n from nodesInterrupted
forall region with region ∈ flow.regionsEntered
  forall c with c ∈ children(region)
    if ∀r ∈ regionsFromTo(c, region) :
      tagValue(r, IARHandling, regionActivationPolicy) = OnParentFlow then
      foreach n with n ∈ AcceptEventAction ∧ c ∈ n.inInterruptibleRegion ∧
        n ∉ nodesInterrupted
        add n to nodesEntered
seq

if ∃a ∈ ActivityFinalNode | a ∈ flow.actionsStart \ nodesInterrupted then
  foreach n with n ∈ ActivityNode ∧ a.activity = n.activity
    r := r ⊕ CREATEACTIONTRANSITION(n, inactive)
    seq
    if n ∈ AcceptEventAction then
      r := r ⊕ CREATEAEABUFFERTRANSITION(n, inactive)
    seq
    if callingAction(a.activity) ≠ undef then
      r := r ⊕ CREATEReturn(a.activity)
    activityfinal := true

```

```

else
  foreach a with a ∈ (flow.actionsStart \ nodesInterrupted) ∪ nodesEntered
    if a ∈ CallBehaviorAction then
      foreach p with p ∈ a.input
        let
          dataFlow = d : d ∈ flow.dataFlows ∧ d.Second = p
          parameter = PINTOPARAMETER(p)
        in
          r := r ⊕ CREATEDATAFLOW(dataFlow.First, parameter)
      seq
      add a to callingAction(a.behavior)
      foreach n with n ∈ ActivityNodes ∧ n.activity = a.behavior
        if n ∈ InitialNode then
          r := r ⊕ CREATEACTIONTRANSITION(n, offeringToken)
        if n ∈ Action then
          if predecessors(n) = ∅ ∧ n.inGroup = ∅ then
            r := r ⊕ CREATEACTIONTRANSITION(n, running)
      seq
      if a.isSynchronous = false then
        r := r ⊕ CREATEACTIONTRANSITION(a, offeringToken)
      else
        r := r ⊕ CREATEACTIONTRANSITION(a, running)
    else
      if a ∉ FlowFinalNode then
        r := r ⊕ CREATEACTIONTRANSITION(a, running)
seq

if activityfinal = false then
  foreach d with d ∈ flow.dataFlows
    r := r ⊕ CREATEDATAFLOWWITHOUTREMOVE(d.First, d.Second)
  seq
  foreach d with d ∈ flow.dataFlows
    r := r ⊕ CREATEDATAREMOVE(d.First)
  seq
  foreach n, e with (n, e) ∈ flow.forksBuffering
    if e ∈ ControlFlow ∧ n ∉ nodesInterrupted then
      r := r ⊕ CREATENULLDATAADD(forkEdgeToObjectNodes(n, e))
  seq
  foreach n, e with (n, e) ∈ flow.forksOffering
    if e ∈ ControlFlow then
      r := r ⊕ CREATEDATAREMOVE(forkEdgeToObjectNodes(n, e))
  seq
  foreach n with n ∈ (flow.actionsOffering ∪ nodesInterrupted) \ nodesEntered ∧
    n ∉ flow.actionsStart
    if n ∈ Action then
      r := r ⊕ CREATEACTIONTRANSITION(n, inactive)
  seq

```

```

    if n ∈ AcceptEventAction then
      r := r ⊕ CREATEAEABUFFERTRANSITION(n, inactive)
    if n ∈ ObjectNode then
      r := r ⊕ CREATEEMPTYLIST(n)
seq
result := r

```

```

CREATERETURN : Activity → ModelCheckerInput*
CREATERETURN(activity) ≡
  local r : ModelCheckerInput* = []
  foreach ca with ca ∈ callingAction(activity)
    if ca.isSynchronous = true then
      foreach p with p ∈ activity.ownedparameter
        let
          pin = PARAMETERTOPIN(p, ca)
        in
          r := r ⊕ CREATEDATAFLOWWITHOUTREMOVE(p, pin)
      seq
      r := r ⊕ CREATEACTIONTRANSITION(ca, offeringToken)
    seq
  foreach o with o ∈ ObjectNode ∧ o.activity = activity
    r := r ⊕ CREATEEMPTYLIST(o)
  seq
  r := r ⊕ CREATESIGNALBUFFERREMOVEALL(activity)
  seq
  callingAction(activity) := undef
  seq
  result := r

```

Erläuterungen zu den weiteren Makros finden sich in Abschnitt 3.2.5.3.

```

CREATETRANSITIONSRUNNINGTOOFFERING : void → ModelCheckerInput*
CREATETRANSITIONSRUNNINGTOOFFERING ≡
  local r : ModelCheckerInput* = []
  foreach coa with coa ∈ CallOperationAction
    let
      class = getContextClass(coa.activity)
      code = locateCode(class, coa.operation)
    in
      r := r ⊕ CREATESTEPHEADER
      seq
      r := r ⊕ CREATEMODEPRECONDITION(coa, running)
      seq
      r := r ⊕ CREATEOUTPUTPRECONDITIONS(coa)
      seq
      r := r ⊕ CREATECODE(code)
      seq

```

```

     $r := r \oplus \text{CREATEACTIONTRANSITION}(\text{coa}, \text{offeringToken})$ 
foreach ssa with ssa  $\in \text{SendSignalAction}$ 
  if  $\text{tagValue}(\text{ssa}, \text{SignalBuffering}, \text{buffer}) = \text{Yes} \vee$ 
    ( $\text{tagValue}(\text{ssa}, \text{SignalBuffering}, \text{buffer}) = \text{Unspecified} \wedge$ 
     $\text{tagValue}(\text{ssa.} \textit{activity}, \text{SignalBuffering}, \text{buffer}) = \text{Yes}$ ) then
     $r := r \oplus \text{CREATESTEPHEADER}$ 
    seq
     $r := r \oplus \text{CREATEMODEPRECONDITION}(\text{ssa}, \text{running})$ 
    seq
     $r := r \oplus \text{CREATESIGNALDATAFROM}(\text{ssa})$ 
    seq
    foreach a with a  $\in \text{Activity}$ 
      if  $\text{tagValue}(\text{ssa}, \text{SignalBuffering}, \text{replace}) = \text{No} \wedge$ 
         $\text{tagValue}(a, \text{SignalBuffering}, \text{replace}) = \text{No}$  then
           $r := r \oplus \text{CREATESIGNALBUFFERCONCAT}(a, \text{ssa.} \textit{signal})$ 
        else
           $r := r \oplus \text{CREATESIGNALBUFFERREPLACE}(a, \text{ssa.} \textit{signal})$ 
        seq
         $r := r \oplus \text{CREATEACTIONTRANSITION}(\text{ssa}, \text{offeringToken})$ 
      else
        let
          rules =  $\text{CREATECOMBINATIONSFORHANDLINGSIGNALLMEDIATELY}(\text{ssa})$ 
        in
          foreach res with res  $\in \text{rules}$ 
             $r := r \uplus \text{res}$ 
    foreach signal with signal  $\in \text{Signal}$ 
       $r := r \uplus \text{CREATECOMBINATIONSFORHANDLINGSIGNAL}(\text{signal})$ 
    seq
    result :=  $r$ 

```

$\text{CREATECOMBINATIONSFORHANDLINGSIGNAL} : \text{Signal} \rightarrow \text{ModelCheckerInput}^*$

$\text{CREATECOMBINATIONSFORHANDLINGSIGNAL}(\text{signal}) \equiv$

```

local  $r : \text{ModelCheckerInput}^* = []$ 
let
  powerset =  $\text{POWERSET}(\{a \in \text{AcceptEventAction} \mid a.\textit{trigger.event.signal} = \text{signal}\})$ 
in
  foreach set with set  $\in \text{powerset}$ 
     $r := r \oplus \text{CREATESTEPHEADER}$ 
    seq
    foreach aea with aea  $\in \text{set}$ 
       $r := r \oplus \text{CREATEMODEPRECONDITION}(\text{aea}, \text{running})$ 
      seq
      if  $\text{internal}(\text{signal})$  then
         $r := r \oplus \text{CREATESIGNALBUFFERPRECONDITION}(\text{aea.} \textit{activity}, \text{signal})$ 
      seq
      foreach aea with aea  $\in \text{set}$ 

```

```

     $r := r \oplus \text{CREATE\_SIGNAL\_DATA\_TO}(signal, aea)$ 
  seq
   $r := r \uplus \text{CREATE\_MUST\_STAY\_ACTIVE\_CHECK}(aea)$ 
  seq
  if internal(signal) then
     $r := r \oplus \text{CREATE\_SIGNAL\_BUFFER\_REMOVE}(aea.activity, signal)$ 
seq
result :=  $r$ 

```

$\text{CREATE_MUST_STAY_ACTIVE_CHECK} : \text{AcceptEventAction} \rightarrow \text{ModelCheckerInput}^*$

```

CREATE\_MUST\_STAY\_ACTIVE\_CHECK(action)  $\equiv$ 
  local  $r : \text{ModelCheckerInput}^* = []$ 
  if  $action.incoming \cup action.input = \emptyset$  then
    if  $action.inInterruptibleRegion = \emptyset$  then
       $r := \text{CREATE\_ACTION\_TRANSITION}(action, running)$ 
      seq
       $r := r \oplus \text{CREATE\_AEA\_BUFFER\_TRANSITION}(action, offeringToken)$ 
    else
      let
        set =  $\text{CREATE\_MUST\_STAY\_ACTIVE\_CHECK\_SET}(action)$ 
      in
         $r := r \oplus \text{CREATE\_AEA\_BUFFER\_TRANSITION}(action, offeringToken)$ 
        seq
         $r := r \oplus \text{CREATE\_IF\_MODE\_FOR\_NODES}(set, inactive)$ 
        seq
         $r := r \oplus \text{CREATE\_ACTION\_TRANSITION}(action, inactive)$ 
        seq
        let
          elsecode =  $\text{CREATE\_ACTION\_TRANSITION}(action, running)$ 
        in
           $r := r \oplus \text{CREATE\_ELSE}(elsecode)$ 
    else
       $r := \text{CREATE\_ACTION\_TRANSITION}(action, offeringToken)$ 
  seq
  result :=  $r$ 

```

$\text{CREATE_MUST_STAY_ACTIVE_CHECK_SET} : \text{AcceptEventAction} \rightarrow \mathcal{P}(\text{ActivityNode})$

```

CREATE\_MUST\_STAY\_ACTIVE\_CHECK\_SET(action)  $\equiv$ 
  local  $res : \mathcal{P}(\text{ActivityNode}) = \emptyset$ 
  let
     $ir = r : r \in \text{InterruptibleActivityRegion} \wedge action \in r.containedNode$ 
  in
    for all  $r$  with  $r \in action.inInterruptibleActivityRegion$ 
      if  $ir = r \vee \forall r' \in \text{regionsFromTo}(ir, r) :$ 
         $tagValue(r', \text{IARHandling}, \text{regionReactivationPolicy}) = \text{OnParentActive}$ 
      then

```

```

forall n with n ∈ Action ∪ InitialNode ∪ OutputPin ∪ CentralBufferNode ∧
  n ∈ r.node
  add n to res
forall n with n ∈ ForkNode ∧ n ∈ r.node
  foreach e with e ∈ n.outgoing
    add forkEdgeToObjectNode(n, e) to res
seq
result := res

```

CREATECOMBINATIONSFORHANDLINGSIGNALLMEDIATELY :

SendSignalAction → *ModelCheckerInput**

CREATECOMBINATIONSFORHANDLINGSIGNALLMEDIATELY(*ssa*) ≡

```

local r : ModelCheckerInput* = []
let
  powerset = POWERSET({a ∈ AcceptEventAction | a.trigger.event.signal = ssa.signal})
in
  foreach set with set ∈ powerset
    r := r ⊕ CREATESTEPHEADER
    seq
    r := r ⊕ CREATEMODEPRECONDITION(ssa, running)
    seq
    foreach aea with aea ∈ set
      r := r ⊕ CREATEMODEPRECONDITION(aea, running)
    seq
    if |ssa.input| > 0 then
      foreach aea with aea ∈ set
        r := r ⊕ CREATEDATAFLOWWITHOUTREMOVE
          (first(asList(ssa.input)), first(asList(aea.output)))
        seq
        r := r ⊕ CREATEDATAREMOVE(first(asList(ssa.input)))
      seq
      foreach aea with aea ∈ set
        r := r ⊕ CREATEMUSTSTAYACTIVECHECK(aea)
      seq
      r := r ⊕ CREATEACTIONTRANSITION(ssa, offeringToken)
    seq
    result := r

```

Das folgende Makro ist im Text nicht ausführlich beschrieben und wird daher hier kurz erläutert.

Das Makro erstellt pro Aktivität einen Zustandsübergang, der schalten kann, wenn alle Knoten der Aktivität sich im Modus *inactive* befinden und keine Objekte mehr in irgendwelchen Objektknoten (außer den *ActivityParameterNodes*) gepuffert sind. Ist dies der Fall, ist die Aktivität beendet und springt — falls sie über eine *CallBehaviorAction* synchron aufgerufen wurde — wieder zur aufrufenden Aktion zurück.

CREATEFINISHCHECKS : *void* → *ModelCheckerInput**

```

CREATEFINISHCHECKS ≡
  local  $r$  : ModelCheckerInput* = []
  foreach activity with activity ∈ Activity ∧ callingAction(activity) ≠ undef
    foreach n with n ∈ Action ∪ InitialNode ∧ n.activity = activity
       $r := r \oplus$  CREATEMODEPRECONDITION(n, inactive)
    seq
    if n ∈ AcceptEventAction then
       $r := r \oplus$  CREATEAEAMODEPRECONDITION(n, inactive)
    seq
    foreach n with n ∈ ForkNode ∧ n.activity = activity
      foreach e with e ∈ n.outgoing
         $r := r \oplus$  CREATELISTPRECONDITION(forkEdgeToObjectNode(n, e), true)
    seq
    foreach o with o ∈ OutputPin ∪ CentralBufferNode
      if o.activity = activity then
         $r := r \oplus$  CREATELISTPRECONDITION(o, true)
    seq
    foreach n with n ∈ ActivityFinalNode ∧ n.activity = activity
       $r := r \oplus$  CREATEACTIONTRANSITION(n, inactive)
    seq
    if n ∈ AcceptEventAction then
       $r := r \oplus$  CREATEAEABUFFERTRANSITION(n, inactive)
    seq
    if callingAction(activity) ≠ undef then
       $r := r \uplus$  CREATEReturn(activity)
  seq
  result :=  $r$ 

```

Die Anforderungen an die folgenden abstrakten Makros werden in Abschnitt 4.1.2 textuell beschrieben.

```

DEFINEMODES : void → ModelCheckerInput
DEFINEATTRIBUTE : Attribute → ModelCheckerInput
DEFINENODEVARIABLE : Node × NodeMode → ModelCheckerInput
DEFINEAEABUFFERVARIABLE : Node × NodeMode → ModelCheckerInput
DEFINELIST : ObjectNode → ModelCheckerInput
DEFINESIGNAL : Signal → ModelCheckerInput
DEFINESIGNALLIST : Activity → ModelCheckerInput
CREATESTEPHEADER : void → ModelCheckerInput
CREATEMODEPRECONDITION : ActivityNode × NodeMode → ModelCheckerInput
CREATEAEAMODEPRECONDITION : AcceptEventAction × NodeMode → ModelCheckerInput
CREATELISTPRECONDITION : ObjectNode × Boolean → ModelCheckerInput
CREATEGUARDPRECONDITION : GuardSpec → ModelCheckerInput
CREATESIGNALBUFFERPRECONDITION : Activity × Signal → ModelCheckerInput
CREATEOUTPUTPRECONDITION : Action → ModelCheckerInput
CREATEBOUNDPRECONDITION : ObjectNode × Nat → ModelCheckerInput

```

CREATEACTIONTRANSITION : *ActivityNode* × *NodeMode* → *ModelCheckerInput*
CREATEAEABUFFERTRANSITION : *AcceptEventAction* × *NodeMode* → *ModelCheckerInput*
CREATEDATAFLOW : *ObjectNode* × *ObjectNode* → *ModelCheckerInput*
CREATEDATAFLOWWITHOUTREMOVE : *ObjectNode* × *ObjectNode* → *ModelCheckerInput*
CREATEEMPTYLIST : *ObjectNode* → *ModelCheckerInput*
CREATENULLDATAADD : *ObjectNode* → *ModelCheckerInput*
CREATEDATAREMOVE : *ObjectNode* → *ModelCheckerInput*
CREATECODE : *Code* → *ModelCheckerInput**
CREATE SIGNALBUFFERCONCAT : *Activity* × *Signal* → *ModelCheckerInput*
CREATE SIGNALBUFFERREPLACE : *Activity* × *Signal* → *ModelCheckerInput*
CREATE SIGNALBUFFERREMOVE : *Activity* × *Signal* → *ModelCheckerInput*
CREATE SIGNALBUFFERREMOVEALL : *Activity* → *ModelCheckerInput*
CREATE SIGNALDATAFROM : *SendSignalAction* → *ModelCheckerInput*
CREATE SIGNALDATATO : *Signal* × *AcceptEventAction* → *ModelCheckerInput*
CREATEIFMODEFORNODES : $\mathcal{P}(\textit{ActivityNode})$ × *NodeMode* → *ModelCheckerInput**
CREATEELSE : *ModelCheckerInput** → *ModelCheckerInput*

A.4 Hilfsfunktionen und -makros

In diesem Abschnitt sind verschiedene Makros zusammengefasst, die für den eigentlichen Algorithmus als Hilfsfunktionen fungieren. Komplexere Aufgaben sind in diese Makro ausgelagert, um die obigen Beschreibungen etwas übersichtlicher zu halten. Da sie deshalb auch nicht im Text erläutert werden, wird ihre jeweilige Funktion im Folgenden kurz beschrieben.

Das Makro **CREATEFLOWCOMBINATIONSFORFORKNODE** wird während der Flussberechnung für jeden *ForkNode* aufgerufen. Das Makro erstellt für jede Kombination der \langle Guards \rangle aller ausgehenden Kanten einen Fluss, da die UML-Spezifikation vorsieht, dass keine Tokens an den Kanten eines *ForkNodes* angeboten (und damit auch nicht gepuffert) werden, falls der \langle Guard \rangle der entsprechenden Kante *false* ist.

Dazu werden die Kanten von „links“ nach „rechts“ (bzw. von 0 bis *n*) rekursiv durchlaufen. Jede Kante wird mit dem \langle Guard \rangle und dessen Negation mit den Flüssen der vorherigen Kanten zu neuen Flüssen kombiniert und der Gesamtmenge hinzugefügt. Dabei wird neben der korrekten Initialisierung der Datenflüsse auch beachtet, ob überhaupt ein \langle Guard \rangle vorhanden ist oder nicht.

Zur Negierung des \langle Guards \rangle wird die abstrakte externe Funktion *negateGuard* definiert, die genau diese Aufgabe erledigt.

monitored *negateGuard* : *GuardSpecification* → *GuardSpecification*

CREATEFLOWCOMBINATIONSFORFORKNODE : *ActivityNode* × *Flow* × *Nat* → $\mathcal{P}(\textit{Flow})$
CREATEFLOWCOMBINATIONSFORFORKNODE(*node*, *flow*, *pos*) ≡
local *r* : $\mathcal{P}(\textit{Flow}) = \emptyset$
if *pos* ≤ |*node.outgoing*| **then**
let
edge = *outgoing*(*node*, *pos*)

```

rekFlows = CREATEFLOWCOMBINATIONSFORFORKNODE(node, flow, pos + 1)
positiveFlow = CLONEFLOW(flow)
negativeFlow = CLONEFLOW(flow)
negGuard = negateGuard(flow.guard)
in
  add (node, edge) to positiveFlows.forksBuffering
  if edge ∈ ObjectFlow then
    first(asList(positiveFlow.dataFlows)).Second := forkEdgeToObjectNode(node, edge)
    negativeFlows.dataFlows := ∅
  add edge.guard to positiveFlow.guards
  add negGuard to negativeFlow.guards
  seq
  add positiveFlow to r
  if negGuard ≠ undef then
    add negativeFlow to r
  seq
  r := ELEMENTWISEUNION(rekFlows, r)
seq
result := r

```

Das folgende Makro liefert `true` zurück, falls die übergebene Kante von einem Join gefolgt wird. Dabei dürfen verschiedene Kontrollknoten passiert werden. Es werden auch implizite Joins berücksichtigt.

Für einen *MergeNode* wird rekursiv die einzige ausgehende Kante verfolgt, für *Fork-* und *DecisionNodes* wird die Funktion für alle ausgehenden Kanten rekursiv aufgerufen.

```

ISFOLLOWEDBYJOIN : ActivityEdge → Boolean
ISFOLLOWEDBYJOIN(edge) ≡
  local r : Boolean = false
  let
    node = edge.target
  in
    if (node ∈ Action ∧ multipleInFlows(node)) ∨
      (node ∈ Pin ∧ multipleInFlows(node.owner)) ∨
      (node ∈ JoinNode) then
      r := true
    if node ∈ MergeNode then
      r := ISFOLLOWEDBYJOIN(outgoing(node,1))
    if node ∈ ForkNode ∨ node ∈ DecisionNode then
      foreach e with e ∈ node.outgoing
        r := r ∨ ISFOLLOWEDBYJOIN(e)
  seq
  result := r

```

Diese Makro kombiniert zwei übergebene Flussmengen elementweise. Dabei werden nicht die Elemente der beiden Mengen vereinigt, sondern die Elemente der Mengen die in der Datenstruktur *Flow* zusammengefasst sind, jeweils „kreuzweise“ verknüpft.

Ein Beispiel für eine elementweise Vereinigung zweier Mengen ist folgendes:

$s_1 = \{\{1,2\}, \{2,3\}, \{4,5\}\}$

$s_2 = \{\{2,3,4\}, \{1,5\}\}$

Ergebnismenge: $\{\{1,2,3,4\}, \{1,2,5\}, \{2,3,4\}, \{1,2,3,5\}, \{2,3,4,5\}, \{1,4,5\}\}$

ELEMENTWISEUNION : $\mathcal{P}(\text{Flow}) \times \mathcal{P}(\text{Flow}) \rightarrow \mathcal{P}(\text{Flow})$

ELEMENTWISEUNION(s_1, s_2) \equiv

```

local  $r : \mathcal{P}(\text{Flow})$ 
if  $s_1 = \emptyset$  then
   $r := s_2$ 
if  $s_2 = \emptyset$  then
   $r := s_1$ 
forall  $f_1 \in s_1$ 
  forall  $f_2 \in s_2$ 
  let
     $\text{res} = \text{COMBINEFLOWS}(f_1, f_2)$ 
     $\text{resred} = \text{REDUCEELEMENTS}(\text{res})$ 
  in
    add  $\text{res}$  to  $r$ 
seq
result :=  $r$ 

```

Die Reduktion der Flüsse wird von folgendem Makro erledigt. Es werden immer korrespondierende Mengen eines Flusses (*actionsOffering* ↔ *actionsStart*, *forksOffering* ↔ *forksBuffering*, ...) nach gleichen Elementen durchsucht und diese aus beiden Mengen entfernt.

REDUCEELEMENTS : $\text{Flow} \rightarrow \text{Flow}$

REDUCEELEMENTS(flow) \equiv

```

forall  $n_1$  with  $n_1 \in \text{flow.actionsOffering}$ 
  forall  $n_2$  with  $n_2 \in \text{flow.actionsStart}$ 
  if  $n_1 = n_2$  then
    remove  $n_1$  from  $\text{flow.actionsOffering}$ 
    remove  $n_2$  from  $\text{flow.actionsStart}$ 
forall  $n_1$  with  $n_1 \in \text{flow.forksOffering}$ 
  forall  $n_2$  with  $n_2 \in \text{flow.forksBuffering}$ 
  if  $n_1 = n_2$  then
    remove  $n_1$  from  $\text{flow.forksOffering}$ 
    remove  $n_2$  from  $\text{flow.forksBuffering}$ 
forall  $n_1$  with  $n_1 \in \text{flow.objectNodesOffering}$ 
  forall  $n_2$  with  $n_2 \in \text{flow.objectNodesBuffering}$ 
  if  $n_1 = n_2$  then
    remove  $n_1$  from  $\text{flow.objectNodesOffering}$ 
    remove  $n_2$  from  $\text{flow.objectNodesBuffering}$ 

```

Ähnlich zum vorherigen Makro werden transitive Datenflüsse über *ForkNode*-Puffer aus der Menge *dataFlows* eines Flusses entfernt.

REDUCEDATAFLOWS : $\text{Flow} \rightarrow \text{Flow}$

```

REDUCEDATAFLOWS(flow) ≡
  forall t1 with t1 ∈ flow.dataFlows
    forall t2 with t2 ∈ flow.dataFlows
      if ¬(t1.Second ∈ CentralBufferNode ∪ ActivityParameterNode) then
        if t1.Second = t2.First then
          remove t1 from flow.dataFlows
          remove t2 from flow.dataFlows
          seq
            t1.Second := t2.Second
          add t1 to flow.dataFlows

```

Dieses Makro wird unter anderem von **ELEMENTWISEUNION** benutzt, um alle Teilmengen eines Flusses jeweils miteinander zu kombinieren.

COMBINEFLOWS : *Flow* × *Flow* → *Flow*

COMBINEFLOWS(*f*₁, *f*₂) ≡

```

let
  r = new(Flow)
in
  r.actionsOffering := f1.actionsOffering ∪ f2.actionsOffering
  r.forksOffering := f1.forksOffering ∪ f2.forksOffering
  r.objectNodesOffering := f1.objectNodesOffering ∪ f2.objectNodesOffering
  r.guards := f1.guards ∪ f2.guards
  r.actionsStart := f1.actionsStart ∪ f2.actionsStart
  r.forksBuffering := f1.forksBuffering ∪ f2.forksBuffering
  r.objectNodesBuffering := f1.objectNodesBuffering ∪ f2.objectNodesBuffering
  r.regionsEntered := f1.regionsEntered ∪ f2.regionsEntered
  r.regionsInterrupted := f1.regionsInterrupted ∪ f2.regionsInterrupted
  r.dataFlows := f1.dataFlows ∪ f2.dataFlows
  result := r

```

Das Makro **CHECKDATAFLOWS** überprüft die Menge *dataFlows* eines Flusses auf Inkonsistenzen. Inkonsistente Datenflüsse ergeben sich bei der Kombination der einfachen Flüsse, wenn mehrere Datenflüsse aus der gleichen Quelle auf unterschiedliche *InputPins* zeigen.

CHECKDATAFLOWS : *Flow* → *Boolean*

CHECKDATAFLOWS(*flow*) ≡

```

local r := true
forall n1, n2 with (n1, n2) ∈ flow.dataFlows
  if n2 ∈ InputPin then
    if ∃(ns, nt) ∈ flow.dataFlows : ns = n1 ∧ n2 ≠ nt then
      r := false
seq
  result := r

```

CLONEFLOW erzeugt eine Kopie des übergebenen Flusses.

CLONEFLOW : *Flow* → *Flow*

```

CLONEFLOW(flow)  $\equiv$ 
  let
    r = new(Flow)
  in
    r.actionsOffering := flow.actionsOffering
    r.forksOffering := flow.forksOffering
    r.objectNodesOffering := flow.objectNodesOffering
    r.guards := flow.guards
    r.actionsStart := flow.actionsStart
    r.forksBuffering := flow.forksBuffering
    r.objectNodesBuffering := flow.objectNodesBuffering
    r.regionsEntered := flow.regionsEntered
    r.regionsInterrupted := flow.regionsInterrupted
    r.dataFlows := flow.dataFlows
  seq
  result := r

```

CLONEFLOWS ruft für jedes Element der übergebenen Menge **CLONEFLOW** auf und erzeugt somit eine Kopie der gesamten Menge.

```

CLONEFLOWS :  $\mathcal{P}(\textit{Flow}) \rightarrow \mathcal{P}(\textit{Flow})$ 
CLONEFLOWS(flows)  $\equiv$ 
  local r :  $\mathcal{P}(\textit{Flow})$ 
  forall f with f  $\in$  flows
    add CLONEFLOW(f) to r
  seq
  result := r

```

Das folgende Makro ruft für jeden *OutputPin* des übergebenen Knotens das Makro **CREATEOUTPUTPRECONDITION** auf.

```

CREATEOUTPUTPRECONDITIONS : ActivityNode  $\rightarrow$  ModelCheckerInput*
CREATEOUTPUTPRECONDITIONS(node)  $\equiv$ 
  local r : ModelCheckerInput* = []
  foreach o with o  $\in$  node.output
    r := r  $\oplus$  CREATEOUTPUTPRECONDITION(o)
  seq
  result := r

```

Dieses Makro bestimmt die Potenzmenge der übergebenen Menge rekursiv:

```

POWERSET :  $\mathcal{P}(\textit{Element}) \rightarrow \mathcal{P}(\mathcal{P}(\textit{Element}))$ 
POWERSET(set)  $\equiv$ 
  if set =  $\emptyset$  then
    result :=  $\{\emptyset\}$ 
  else
    choose a with a  $\in$  set
    let

```

```

    powerset = POWERSET(set \ a)
in
    result := powerset  $\cup$  { X  $\cup$  {a} | X  $\in$  powerset }

```

Die beiden folgenden Makros bilden *ActivityParameterNodes* auf Output- bzw. *InputPins* ab. Dabei wird einerseits auf den Typ des *ActivityParameterNodes* geachtet, andererseits wird davon ausgegangen, dass die *ActivityParameterNodes* und die *Pins* in der gleichen Reihenfolge stehen, so dass sie bezüglich ihrer Position abgebildet werden können.

```

PARAMETERTOPIN : ActivityParameterNode  $\times$  CallBehaviorAction  $\rightarrow$  OutputPin
PARAMETERTOPIN(n, callAction)  $\equiv$ 
let
    outParameters = [p  $\in$  callAction.behavior.ownedParameters | p.direction  $\in$  {inout, out, return}]
in
    choose o with o  $\in$  callAction.result  $\wedge$ 
        indexOf(callAction.result, o) = indexOf(outParameters, n.parameter)
    result := o

```

```

PINTOPARAMETER : InputPin  $\rightarrow$  ActivityParameterNode
PINTOPARAMETER(inputPin)  $\equiv$ 
choose callAction with callAction  $\in$  CallBehaviorAction  $\wedge$  inputPin  $\in$  callAction.argument
let
    inParameters = [p  $\in$  callAction.behavior.ownedParameters | p.direction  $\in$  {in, inout}]
in
    choose n with n  $\in$  ActivityParameterNode  $\wedge$ 
        indexOf(callAction.argument, inputPin) = indexOf(inParameters, n.parameter)
    result := n

```

Die Funktion *predecessors* liefert eine Menge aller Vorgänger eines Knotens zurück:

```

derived predecessors: ActivityNode  $\rightarrow$   $\mathcal{P}$ (ActivityNode)
predecessors(node) =def
    {p  $\in$  ActivityNode |  $\exists$  e  $\in$  ActivityEdge: e.source = p  $\wedge$  e.target = node}

```

Die folgenden Funktionen bieten lediglich eine kürzere Schreibweise für den Zugriff auf die Elemente der entsprechenden Listen.

```

derived input: Action  $\times$  Nat  $\rightarrow$  InputPin
input(action, i) =def elementAt(action.input, i)
derived output: Action  $\times$  Nat  $\rightarrow$  OutputPin
output(action, i) =def elementAt(action.output, i)
derived incoming, outgoing: ActivityNode  $\times$  Nat  $\rightarrow$  ActivityEdge

```

Diese Funktion liefert true, falls die übergebene Aktion mehr als eine ausgehende Kante besitzt.

```

derived multipleOutFlows: Action  $\rightarrow$  Boolean
multipleOutFlows(action) =def |action.outgoing| + |action.output| > 1

```

Diese Funktion liefert `true`, falls die übergebene Aktion mehr als eine eingehende Kante besitzt.

derived *multipleInFlows*: *Action* \rightarrow *Boolean*

multipleInFlows(*action*) =_{def} $|action.incoming| + |action.input| > 1$

In der Funktion *callingAction* werden die Aufrufer einer Aktivität gespeichert, damit durch das Makro `CREATERETURN` wieder zu den entsprechenden Aktionen zurück gesprungen werden kann.

controlled *callingAction*: *Activity* \rightarrow $\mathcal{P}(\text{CallBehaviorAction})$

Gemäß der Definition eines internen Signals überprüft diese Funktion, ob das übergebene Signal durch eine *SendSignalAction* erzeugt wird.

derived *internal*: *Signal* \rightarrow *Boolean*

internal(*signal*) =_{def} $\exists ssa \in \text{SendSignalAction} : ssa.signal = signal$

Die abgeleitete Funktion *children* liefert die Menge der *InterruptibleActivityRegions*, die sich innerhalb der übergebenen befinden.

derived *children*: *InterruptibleActivityRegion* \rightarrow $\mathcal{P}(\text{InterruptibleActivityRegion})$

children(*r*) =_{def}
$$\begin{cases} subGroup(r) \cup \bigcup_{r' \in subGroup(r)} children(r'), & \text{if } subGroup(r) \neq \emptyset \\ \emptyset, & \text{otherwise} \end{cases}$$

Diese Funktion liefert alle Unterbrechungsbereiche zurück, die sich auf den Ebenen zwischen den beiden übergebenen befinden.

derived *regionsFromTo*: *InterruptibleActivityRegion* \times *InterruptibleActivityRegion* \rightarrow $\mathcal{P}(\text{InterruptibleActivityRegion})$

regionsFromTo(*r*,*s*) =_{def}
$$\begin{cases} \{r\}, & \text{if } r.superGroup = s \\ \{r\} \cup regionsFromTo(r.superGroup, s), & \text{otherwise} \end{cases}$$

Der einen Knoten unmittelbar umgebende Unterbrechungsbereich wird durch folgende Funktion zurückgeliefert:

derived *immediateRegion*: *ActivityNode* \rightarrow *InterruptibleActivityRegion*

immediateRegion(*n*) =_{def} $r \in n.inInterruptibleRegion \mid \forall s \in children(r) : s \notin n.inInterruptibleRegion$

Abbildungsverzeichnis

2.1	Mächtigkeit verschiedener Varianten der Temporallogik.	9
2.2	Beispiel für ein Zustandsübergangssystem und der Anfang des dazugehörigen Ausführungsbaums.	10
2.3	Ein RBDD für $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$	14
2.4	Das ROBDD für $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$ mit der Variablenordnung $x_1 < x_2 < y_1 < y_2$	15
2.5	Klassifikation von ASM Funktionen.	18
2.6	In dieser Arbeit verwendeter Ausschnitt aus dem Metamodell der UML 2 . . .	22
2.7	Syntax der UML 2 Aktivitätsdiagramme	25
2.8	Beispiel für Tokenpufferung an <i>ForkNodes</i>	26
2.9	Definition der Variationspunkte für <i>InterruptibleActivityRegions</i>	27
2.10	Definition der Variationspunkte zur Signalpufferung.	28
3.1	Beispiel für einen Zyklus zwischen zwei Kontrollknoten.	30
3.2	Beispiel für eine Überabschätzung durch Abstraktion von Zeit.	31
3.3	Definition des Variationspunkts zur Kennzeichnung von Startaktivitäten. . .	32
3.4	Übersicht über den Ablauf der Verifikation von Aktivitätsdiagrammen	34
3.5	Ausschnitt aus dem Metamodell für UML 2 Aktivitätsdiagramme.	36
3.6	Beispiel für einen Zustandsübergang.	39
3.7	Einzelne Schritte der Flussberechnung.	40
3.8	Aktivitätsdiagrammbeispiel zur Erläuterung der Flussberechnung.	44
3.9	Ausschnitt aus dem Aufrufgraphen für das Makro COMPSIMPLEFLOWS	45
3.10	Mehrere aus- bzw. eingehende Flüsse werden implizit wie ein <i>Fork-</i> bzw. <i>Join-Node</i> behandelt.	47
3.11	Ausschnitt aus dem Aufrufgraphen für die Makros FLOWDOWN und FLOWUP . 50	50
3.12	Beispiel von unterbrochenen und betretenen <i>InterruptibleActivityRegions</i> . . .	51
3.13	Ausschnitt aus Abb. 3.8 mit Darstellung der möglichen Flüsse.	52
3.14	Aktivitätsdiagrammausschnitt mit <i>Join-</i> und <i>ForkNodes</i>	53
3.15	Einzelne Schritte der Zustandsübergangserzeugung.	65
3.16	Einige Kombinationen von Kontrollknoten und Aktionen zur strukturellen Zerlegung von Aktivitätsdiagrammen.	84
4.1	Beispiel für ein Model Program.	89
4.2	Aktivitätsdiagramm einer Alarmanlage (entnommen aus [FMS08]).	97
4.3	Aktivitätsdiagramm der Zweihandpressensteuerung.	100
4.4	Aktivitätsdiagramm der Zweihandpressensteuerung.	102
4.5	Sequenzdiagramm zur Umgebungsspezifikation.	106

4.6	Aktivitätsdiagramm mit \langle Guards \rangle an ausgehenden Kanten des <i>ForkNodes</i> . . .	108
4.7	Beispiel zur Optimierung durch Zusammenfassen von Aktionen.	111
4.8	Ergebnis der Optimierung von Abb. 4.7.	112
5.1	Beispiel für ein mit Ausführungszeiten versehenes Aktivitätsdiagramm.	118
A.1	In dieser Arbeit betrachteter Ausschnitt aus dem Metamodell der UML 2. . .	132

Tabellenverzeichnis

1.1	Mathematische Konventionen.	6
2.1	Wichtigste Klassen von Systemeigenschaften.	12
2.2	Für die Arbeit relevante ASM-Konstrukte.	20
3.1	Übersicht der Einschränkungen.	33
3.2	Wertebelegung der <i>Flow</i> -Datenstruktur für Beispiel aus Abb. 3.6.	42
4.1	Mögliche Umsetzungen der Variablen in Model Programs	91
5.1	Gegenüberstellung der Arbeiten zur Verifikation von Aktivitätsdiagrammen. .	126

Glossar

- ASM** Abkürzung für Abstract State Machine. Ein semantisch fundierter Formalismus, der auf evolving algebras aufbaut. Details siehe [BS03].
- ASM-Makro** Zusammenfassung mehrerer \rightarrow ASM-Regeln zu einer aufrufbaren, parametrisierten Einheit. Ähnlich zu Funktionen in Programmiersprachen.
- ASM-Regel** Bezeichnung für Konstrukte des Formalismus \rightarrow ASM.
- Datenfluss** Fluss von Daten entlang Kanten innerhalb eines Aktivitätsdiagramms. Siehe auch \rightarrow Objektfluss.
- Datenquelle** Knoten im Aktivitätsdiagramm, der Daten bzw. Objekte abgeben kann. Z.B. *OutputPin*, *CentralBuffer*, ...
- Datensenke** Knoten im Aktivitätsdiagramm, der Daten bzw. Objekte erhalten kann. Z.B. *InputPin*, *CentralBuffer*, ...
- einfacher Fluss** Fluss, der zwischen zwei Knoten fließt, ohne einen *ForkNode* zu traversieren.
- externes Signal** Signal, welches nicht innerhalb des Systems bzw. Aktivitätsdiagramms, sondern von der Umgebung erzeugt wird. Kann jederzeit auftreten und wird daher nicht gepuffert.
- Fluss** Bewegung eines oder mehrerer \rightarrow Tokens über Kanten eines Aktivitätsdiagramms.
- Flussbedingung** Gesamtheit der Bedingungen, die gelten müssen, damit ein \rightarrow Fluss fließen kann. Betrifft hauptsächlich \rightarrow Modi von Aktionen und Datenmengen in Objektknoten.
- Fluss-Datenstruktur** Datenstruktur zur Erfassung und Speicherung der möglichen \rightarrow einfachen und \rightarrow kombinierten Flüsse. Beschreibung in Abschnitt 3.2.4.1.
- Flusseffekt** Gesamtheit der Zustandsänderungen, die sich durch einen \rightarrow Fluss ergeben. Betrifft im Wesentlichen \rightarrow Modi von Aktionen und Daten in Objektknoten.
- ForkNode-Pufferung** Situation, dass Tokens an ausgehenden Kanten von *ForkNodes* gepuffert werden müssen, da nachfolgende Aktionen das Token noch nicht aufnehmen können.
- Guarded Command** Beschreibung eines Zustandsübergangs in der Form *guard* \rightarrow *command*.
- impliziter Fork** Mehrere ausgehende Kanten an Aktionen sind wie spezielle *ForkNodes* zu betrachten.
- impliziter Join** Mehrere eingehende Kanten an Aktionen sind wie spezielle *JoinNodes* zu betrachten.

- internes Signal** Signal, welches innerhalb des Systems als Nachricht oder Botschaft erzeugt wird. Kann gepuffert oder auch direkt verarbeitet werden.
- Knoten** Allgemeine Bezeichnung für alle Elemente eines Aktivitätsdiagramms, die keine Kante darstellen. Z. B. Aktion, Kontrollknoten, ...
- Knotentyp** Spezielle Ausprägung eines Knotens wie z. B. *CallOperationAction*, *DecisionNode*, *ForkNode*, *JoinNode*, ...
- kombinierter Fluss** Über *ForkNodes* hinwegführende, aus \rightarrow einfachen Flüssen kombinierte \rightarrow Flüsse.
- Kontrollfluss** \rightarrow Fluss, der keine Daten bzw. Objekte, sondern nur \rightarrow Kontrolltokens transportiert.
- Kontrolltoken** \rightarrow Token, das keine Daten bzw. Objekte transportiert.
- MDS** Modeldriven-Softwaredevelopment = Modellgetriebene Softwareentwicklung mit dem Ziel, Systeme graphisch zu modellieren und Code zu generieren.
- Model Program** Spezifikationsprache für Model Based Testing oder Model Checking
- Model Program Checker** Bounded Model Checker für den Formalismus \rightarrow Model Programs.
- Modus** Bezeichnung für die Zustände, in denen sich eine Aktion befinden kann: inactive, running oder offeringToken.
- Objektfluss** \rightarrow Fluss, der Daten bzw. Objekte transportiert. Kann nur zwischen Objekt-knoten (*Pins*, *ParameterNodes*, *CentralBuffers*) stattfinden.
- Objekttoken** Ein \rightarrow Token, welches Daten bzw. Objekte aufnimmt und nur über \rightarrow Objektflüsse fließen kann.
- OMG** Object Management Group. Internationales offenes Konsortium. Entwickelt unter anderem \rightarrow UML.
- Token** Konstrukt, ähnlich zu Marken in Petri-Netzen, welches zwischen den \rightarrow Knoten eines Aktivitätsdiagramms fließen kann.
- Tokenquelle** Menge der \rightarrow Aktivitätsknoten, die Token erzeugen bzw. anbieten können.
- Token Senke** Menge der \rightarrow Aktivitätsknoten, die Tokens konsumieren können.
- traverse-to-completion** Prinzip der Aktivitätsdiagramme, welches besagt, dass \rightarrow Tokens nicht zwischen Aktionen „geparkt“ werden können, sondern immer von einer Tokenquelle zu einer Token Senke fließen. Ausnahme: \rightarrow ForkNode-Pufferung.
- UML** Unified Modeling Language. Eine graphische Spezifikationsprache, die von der \rightarrow OMG entwickelt und gepflegt wird.
- Zustand** Gesamtheit der \rightarrow Modi von Aktionen kombiniert mit weiteren Systemvariablen. Definition siehe 3.2.3.
- Zustandsübergangssystem** System, welches aus einem Initialzustand und definierten Zustandsübergängen besteht. Grundlage für Model Checking.

Literaturverzeichnis

- [Alk07] ALKSNIS, GUNDARS: *The Analysis of UML State Machine Formal Checking Methods*. In: ZENDULKA, JAROSLAV (Herausgeber): *Proceedings of the 10th International Conference on Information System Implementation and Modeling (ISIM '07)*, Band 252 der Reihe *CEUR Workshop Proceedings*. CEUR-WS.org, April 2007.
- [AMP09] ARMANDO, ALESSANDRO, JACOPO MANTOVANI und LORENZO PLATANIA: *Bounded Model Checking of Software using SMT Solvers instead of SAT Solvers*. *International Journal on Software Tools for Technology Transfer*, 11(1):69–83, 2009.
- [And99] ANDERSEN, HENRIK REIF: *An Introduction to Binary Decision Diagrams*. Lecture Notes, 1999. IT University of Copenhagen.
- [BAMP81] BEN-ARI, MORDECHAI, ZOHAR MANNA und AMIR PNUELI: *The Temporal Logic of Branching Time*. In: *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '81)*, Seiten 164–176. ACM, 1981.
- [BBB⁺99] BIENMÜLLER, TOM, JÜRGEN BOHN, HENNING BRINKMANN, UDO BROCKMEYER, WERNER DAMM, HARDI HUNGAR und PETER JANSEN: *Verification of Automotive Control Units*. In: *Proceedings of Correct System Design, Recent Insight and Advances*, Seiten 319–341. Springer, 1999.
- [BBC⁺] BERGLUND, ANDERS, SCOTT BOAG, DON CHAMBERLIN, MARY F. FERNÁNDEZ, MICHAEL KAY, JONATHAN ROBIE und JÉRÔME SIMÉON: *XML Path Language (XPath) 2.0, W3C Recommendation 23 January 2007*. neueste Fassung: <http://www.w3.org/TR/xpath20/>.
- [BBF⁺01] BÉRARD, BÉATRICE, MICHEL BIDOIT, ALAIN FINKEL, FRANÇOIS LAROUSSE, ANTOINE PETIT, LAURE PETRUCCI, PHILIPPE SCHNOEBELEN und PIERRE MCKENZIE: *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer, 2001.
- [BBG05] BEYDEDA, SAMI, MATTHIAS BOOK und VOLKER GRUHN: *Model-Driven Software Development*. Springer, 2005.
- [BCCZ99] BIERE, ARMIN, ALESSANDRO CIMATTI, EDMUND M. CLARKE und YUNSHAN ZHU: *Symbolic Model Checking without BDDs*. In: *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '99)*, Seiten 193–207. Springer, 1999.

- [BCR04] BÖRGER, EGON, ALESSANDRA CAVARRA und ELVINIA RICCOBENE: *On Formalizing UML State Machines Using ASMs*. Information and Software Technology, 46(5):287–292, April 2004.
- [BDK⁺04] BRILL, MATTHIAS, WERNER DAMM, JOCHEN KLOSE, BERND WESTPHAL und HARTMUT WITTKE: *Live Sequence Charts: An Introduction to Lines, Arrows, and Strange Boxes in the Context of Formal Verification*. In: EHRIG, HARTMUT, WERNER DAMM, JÖRG DESEL, MARTIN GROSSE-RHODE, WOLFGANG REIF, ECKEHARD SCHNIEDER und ENGELBERT WESTKÄMPER (Herausgeber): *Integration of Software Specification Techniques for Applications in Engineering, Priority Program SoftSpez of the German Research Foundation (DFG), Final Report*, Band 3147 der Reihe LNCS, Seiten 374–399, 2004.
- [BDL04] BEHRMANN, GERD, ALEXANDRE DAVID und KIM G. LARSEN: *A Tutorial on UPPAAL*. In: BERNARDO, MARCO und FLAVIO CORRADINI (Herausgeber): *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT '04)*, Nummer 3185 in LNCS, Seiten 200–236. Springer, September 2004.
- [BGM02] BOZGA, MARIUS, SUSANNE GRAF und LAURENT MOUNIER: *IF-2.0: A Validation Environment for Component-Based Real-Time Systems*. In: BRINKSMA, D. und KIM G. LARSEN (Herausgeber): *Computer Aided Verification (CAV '02)*, Band 2404 der Reihe LNCS, Seiten 343–348. Springer, 2002.
- [BKKS08] BECKERS, JÖRG, DANIEL KLÜNDER, STEFAN KOWALEWSKI und BASTIAN SCHLICH: *Direct Support for Model Checking Abstract State Machines by Utilizing Simulation*. In: *Proceedings of the 1st international conference on Abstract State Machines, B and Z (ABZ '08)*, Seiten 112–124. Springer, 2008.
- [BLN03] BEYER, DIRK, CLAUS LEWERENTZ und ANDREAS NOACK: *Rabbit: A Tool for BDD-based Verification of Real-Time Systems*. In: HUNT, W. A. und F. SOMENZI (Herausgeber): *Proceedings of the 15th International Conference on Computer Aided Verification (CAV '03)*, Band 2725 der Reihe LNCS, Seiten 122–125. Springer, 2003.
- [Bry86] BRYANT, RANDAL E.: *Graph-Based Algorithms for Boolean Function Manipulation*. IEEE Transactions on Computers, 35(8):677–691, 1986.
- [Bry91] BRYANT, RANDAL E.: *On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication*. IEEE Transactions on Computers, 40(2):205–213, 1991.
- [Bry92] BRYANT, RANDAL E.: *Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams*. ACM Computing Surveys, 24(3):293–318, 1992.
- [BS03] BÖRGER, E. und R. STÄRK: *Abstract State Machines*. Springer, 2003.
- [BT08] BÖRGER, EGON und BERNHARD THALHEIM: *Advances in Software Engineering: Lipari Summer School 2007*, Kapitel A Method for Verifiable and Validatable Business Process Modeling, Seiten 59–115. Springer, 2008.

- [Büc62] BÜCHI, JULIUS RICHARD: *On a Decision Method in Restricted Second Order Arithmetic*. In: *Proceedings of International Congress Logic, Methodology and Philosophy of Science*, Seiten 1–12. Stanford University Press, 1962.
- [Bus] *Object Management Group, Business Process Modeling Notation (BPMN), Version 1.2, Document 09-01-03, Januar 2009.*
- [CCG⁺02] CIMATTI, ALESSANDRO, EDMUND M. CLARKE, ENRICO GIUNCHIGLIA, FAUSTO GIUNCHIGLIA, MARCO PISTORE, MARCO ROVERI, ROBERTO SEBASTIANI und ARMANDO TACCHELLA: *NuSMV 2: An OpenSource Tool for Symbolic Model Checking*. In: *Proceedings of the 14th International Conference on Computer Aided Verification (CAV '02)*, Seiten 359–364. Springer, 2002.
- [CD89] CLARKE, EDMUND M. und I. A. DRAGHICESCU: *Expressibility Results for Linear-Time and Branching-Time Logics*. In: *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, Seiten 428–437. Springer, 1989.
- [CD08] CRANE, MICHELLE L. und JUERGEN DINGEL: *Towards a UML Virtual Machine: implementing an Interpreter for UML 2 Actions and Activities*. In: *Proceedings of the 2008 conference of the center for advanced studies on collaborative research (CASCON '08)*, Seiten 96–110. ACM, 2008.
- [CES86] CLARKE, EDMUND M., E. A. EMERSON und A. P. SISTLA: *Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications*. *ACM Transaction on Programming Languages and Systems*, 8(2):244–263, 1986.
- [CGJ⁺03] CLARKE, EDMUND, ORNA GRUMBERG, SOMESH JHA, YUAN LU und HELMUT VEITH: *Counterexample-Guided Abstraction Refinement for Symbolic Model Checking*. *Journal of the ACM*, 50(5):752–794, 2003.
- [CGP99] CLARKE, EDMUND M. JR., ORNA GRUMBERG und DORON A. PELED: *Model Checking*. The MIT Press, 1999.
- [CK04] CENGARLE, MARÍA V. und ALEXANDER KNAPP: *UML 2.0 Interactions: Semantics and Refinement*. In: JÜRJENS, JAN, EDUARDO B. FERNANDEZ, ROBERT FRANCE und BERNHARD RUMPE (Herausgeber): *Proceedings of 3rd International Workshop on Critical Systems Development with UML (CSDUML '04)*, Seiten 85–99. Technische Universität München, 2004.
- [DAC99] DWYER, MATTHEW B., GEORGE S. AVRUNIN und JAMES C. CORBETT: *Patterns in Property Specifications for Finite-State Verification*. In: *Proceedings of the 21st International Conference on Software engineering (ICSE '99)*, Seiten 411–420. ACM, 1999.
- [DDHY92] DILL, DAVID L., ANDREAS J. DREXLER, ALAN J. HU und C. HAN YANG: *Protocol Verification as a Hardware Design Aid*. In: *Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors (ICCD '92)*, Seiten 522–525. IEEE Computer Society, 1992.

- [Dij75] DIJKSTRA, EDSGER W.: *Guarded Commands, Nondeterminacy and Formal Derivation of Programs*. Communications of the ACM, 18(8):453–457, August 1975.
- [Dij76] DIJKSTRA, EDSGER W.: *A Discipline of Programming*. Prentice Hall, Inc., Oktober 1976.
- [dMB08] MOURA, LEONARDO MENDONÇA DE und NIKOLAJ BJØRNER: *Z3: An Efficient SMT Solver*. In: RAMAKRISHNAN, C. R. und JAKOB REHOF (Herausgeber): *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*, Band 4963 der Reihe LNCS, Seiten 337–340. Springer, 2008.
- [EH86] EMERSON, E. ALLEN und JOSEPH Y. HALPERN: “*Sometimes*” and “*Not Never*” Revisited: *On Branching versus Linear Time Temporal Logic*. Journal of the ACM, 33(1):151–178, 1986.
- [EL85] EMERSON, E. ALLEN und CHIN-LAUNG LEI: *Modalities for Model Checking: Branching Time Strikes Back*. In: *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages (POPL '85)*, Seiten 84–96. ACM Press, Januar 1985.
- [Esh06] ESHUIS, RIK: *Symbolic Model Checking of UML Activity Diagrams*. ACM Transactions on Software Engineering and Methodology, 15(1):1–38, 2006.
- [ESW07] ENGELS, G., C. SOLTENBORN und H. WEHRHEIM: *Analysis of UML Activities Using Dynamic Meta Modeling*. In: BOSANGUE, M. M. und E. BROCH JOHNSEN (Herausgeber): *Proceedings of the 9th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS '07)*, Band 4468 der Reihe LNCS, Seiten 76–90. Springer, 2007.
- [FGG07] FARAHBOD, ROOZBEH, VINCENZO GERVASI und UWE GLÄSSER: *CoreASM: An Extensible ASM Execution Engine*. Fundamenta Informaticae, 77(1-2):71–103, Mai 2007.
- [FGM08] FARAHBOD, ROOZBEH, UWE GLÄSSER und GEORGE MA: *Model Checking Core-ASM Specifications*. In: *Proceedings of 14th International Workshop on Abstract State Machines (ASM '07)*, 2008.
- [FMS08] FRIEDENTHAL, SANFORD, ALAN MOORE und RICK STEINER: *OMG Systems Modeling Language Tutorial*. online, Juni 2008. <http://www.omgsysml.org/INCOSE-2008-OMGSysML-Tutorial-Final-revb.pdf>.
- [FS07] FECHER, HARALD und JENS SCHÖNBORN: *UML 2.0 State Machines: Complete Formal Semantics via Core State Machines*. In: *Proceedings of Formal Methods: Applications and Technology, 11th International Workshop, FMICS '06 and 5th International Workshop PDMC '06*, Band 4346 der Reihe Lecture Notes in Computer Science, Seiten 244–260. Springer, 2007.
- [GHJ95] GAMMA, ERICH, RICHARD HELM und RALPH E. JOHNSON: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

- [GMB08] GAGNON, PATRICE, FARID MOKHATI und MOURAD BADRI: *Applying Model Checking to Concurrent UML Models*. Journal of Object Technology, 7(1):59–84, Januar-Februar 2008.
- [GMP02] GALLARDO, MARÍA DEL MAR, PEDRO MERINO und ERNESTO PIMENTEL: *Debugging UML Designs with Model Checking*. Journal of Object Technology, 1(2):101–117, Juli-August 2002.
- [Gor88] GORDON, M. J. C.: *VLSI Specification, Verification and Synthesis*, Kapitel HOL: A Proof Generating System for Higher Order Logic, Seiten 73–128. Kluwer Academic Publishers, 1988.
- [GRR03] GARGANTINI, ANGELO, ELVINIA RICCOBENE und SALVATORE RINZIVILLO: *Using Spin to Generate Tests from ASM Specifications*. In: BÖRGER, EGON, ANGELO GARGANTINI und ELVINIA RICCOBENE (Herausgeber): *Proceedings of the 10th International Workshop on Abstract State Machines (ASM '03)*, Band 2589 der Reihe LNCS, Seiten 263–277. Springer, 2003.
- [GRS05] GUREVICH, YURI, BENJAMIN ROSSMAN und WOLFRAM SCHULTE: *Semantic Essence of AsmL*. Theoretical Computer Science, 343(3):370–412, 2005.
- [GT01] GUREVICH, Y. und N. TILLMANN: *Partial Updates: Exploration*. Journal of Universal Computer Science, 7(11):917–951, 2001.
- [Gur84] GUREVICH, YURI: *Reconsidering Turing's Thesis (Toward More Realistic Semantics of Programs)*. Technischer Bericht CRL-TR-36-84, University of Michigan, September 1984.
- [Gur85] GUREVICH, YURI: *A New Thesis*. Abstracts, American Mathematical Society, 6(4):317, August 1985.
- [Gur95] GUREVICH, YURI: *Specification and Validation Methods*, Kapitel Evolving Algebras 1993: Lipari Guide, Seiten 9–36. Oxford University Press, 1995.
- [HKNP06] HINTON, A., M. KWIATKOWSKA, G. NORMAN und D. PARKER: *PRISM: A Tool for Automatic Verification of Probabilistic Systems*. In: HERMANN, H. und J. PALSBERG (Herausgeber): *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '06)*, Band 3920 der Reihe LNCS, Seiten 441–444. Springer, 2006.
- [HMU02] HOPCROFT, JOHN E., RAJEEV MOTWANI und JEFFREY D. ULLMAN: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Pearson Studium, 2. Auflage, 2002.
- [Hol97] HOLZMANN, GERARD J.: *The Model Checker SPIN*. IEEE Transactions on Software Engineering, 23(5):279–295, 1997.
- [Inf] *Object Management Group, UML 2.2 Infrastructure, Document 09-02-04, Februar 2009*.

- [JVCS08] JACKY, JONATHAN, MARGUS VEANES, COLIN CAMPBELL und WOLFRAM SCHULTE: *Model-Based Software Testing and Analysis with C#*. Cambridge University Press, 2008.
- [Kar05] KARDOS, MARTIN: *An Approach to Model Checking AsmL Specifications*. In: *Proceedings of the 12th International Workshop on Abstract State Machines (ASM '05)*, Seiten 289–304, März 2005.
- [Kec06] KECHER, CHRISTOPH: *UML 2.0 - Das umfassende Handbuch*. Galileo Computing, 2006.
- [KG09] KOHLMAYER, JENS und WALTER GUTTMANN: *Unifying the Semantics of UML 2 State, Activity and Interaction Diagrams*. In: *Proceedings of the 7th International Andrei Ershov Memorial Conference on Perspectives of System Informatics (PSI '09)*. Springer, 2009.
- [KK06] KREUZER, MARTIN und STEFAN KÜHLING: *Logik für Informatiker*. Pearson Studium, März 2006.
- [KMR02] KNAPP, ALEXANDER, STEPHAN MERZ und CHRISTOPHER RAUH: *Model Checking - Timed UML State Machines and Collaborations*. In: *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT '02)*, Seiten 395–416. Springer, 2002.
- [Koh08] KOHLMAYER, JENS: *Executing UML 2 Diagrams in ActiveCharts: A formal Semantics for the Combination of Behavior Specifications in the UML 2*. In: BERTELLE, C. und A. AYESH (Herausgeber): *Proceedings of the 2008 European Simulation and Modeling Conference (ESM '08)*, Seiten 94–101, Oktober 2008.
- [Koh09] KOHLMAYER, JENS: *Eine formale Semantik für die Verknüpfung von Verhaltensbeschreibungen in der UML 2*. Doktorarbeit, Universität Ulm, 2009. noch nicht veröffentlicht.
- [Kri63] KRIPKE, SAUL AARON: *Semantical Considerations on Modal Logic*. Acta Philosophica Fennica, 16:83–94, 1963.
- [KS06] KUNTZ, MATTHIAS und MARKUS SIEGLE: *CASPA: Symbolic Model Checking of Stochastic Systems*. In: GERMAN, REINHARD und ARMIN HEINDL (Herausgeber): *Proceedings of the 13th GI/ITG Conference on Measuring, Modelling and Evaluation of Computer and Communication Systems (MMB '06)*, Seiten 465–468. VDE Verlag, März 2006.
- [KW06] KNAPP, ALEXANDER und JOCHEN WUTTKE: *Model Checking of UML 2.0 Interactions*. In: *Models in Software Engineering, Workshops and Symposia at MoDELS '06*, Seiten 42–51, 2006.
- [KWB03] KLEPPE, ANNEKE, JOS WARMER und WIM BAST: *MDA Explained: The Model Driven Architecture — Practice and Promise*. Addison-Wesley, 2003.
- [LLJ04] LI, XIAOSHAN, ZHIMING LIU und HE JIFENG: *A Formal Semantics of UML Sequence Diagram*. In: *Proceedings of the 2004 Australian Software Engineering Conference (ASWEC '04)*, Seite 168. IEEE Computer Society, 2004.

- [LNR⁺08] LETTNIN, DJONES, PRADEEP K. NALLA, JÜRGEN RUF, THOMAS KROPP, WOLFGANG ROSENSTIEL, TOBIAS KIRSTEN, VOLKER SCHÖNKNECHT und STEPHAN REITEMEYER: *Verification of Temporal Properties in Automotive Embedded Software*. In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '08)*, Seiten 164–169. ACM, 2008.
- [MDAa] *Object Management Group, MDA Guide 1.0.1, Document 03-06-01, Juni 2003.*
- [MDAb] *Object Management Group, OMG Systems Modeling Language 1.1, Document 08-11-02, November 2008.*
- [MKB95] MOORE, J STROTHER, M. KAUFMANN und R.S. BOYER: *The Boyer-Moore Theorem Prover and Its Interactive Enhancement*. *Computers and Mathematics with Applications*, 29(2):27–62, 1995.
- [MOF] *Object Management Group, Meta Object Facility (MOF) 2.0, Document 06-01-01, Januar 2006.*
- [Moo06] *IEEE Solid-State Circuits Society Newsletter*, September 2006.
- [Mor08] MORIMOTO, SHOICHI: *A Survey of Formal Verification for Business Process Modeling*. In: *Proceedings of the 8th International Conference on Computational Science, Part II (ICCS '08)*, Seiten 514–522. Springer, 2008.
- [NPW02] NIPKOW, TOBIAS, LAWRENCE C. PAULSON und MARKUS WENZEL: *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, Band 2283 der Reihe LNCS. Springer, 2002.
- [NR69] NAUR, P. und B. RANDELL (Herausgeber): *Software Engineering: Report of a conference sponsored by the NATO Science Committee*, Garmisch, Deutschland, 1969. NATO, Scientific Affairs Division.
- [NS07] NORIRISH, MICHAEL und KONRAD SLIND: *The HOL System - Description*, Januar 2007.
- [OGO04] OBER, IULIAN, SUSANNE GRAF und ILEANA OBER: *Model checking of UML models via a mapping to communicating extended timed automata*. In: *Proceedings of the 11th International SPIN Workshop on Model Checking of Software*, Band 2989 der Reihe LNCS. Springer, 2004.
- [ORS92] OWRE, S., J. M. RUSHBY, und N. SHANKAR: *PVS: A Prototype Verification System*. In: KAPUR, DEEPAK (Herausgeber): *Proceedings of the 11th International Conference on Automated Deduction (CADE)*, Band 607 der Reihe *Lecture Notes in Artificial Intelligence*, Seiten 748–752. Springer, Juni 1992.
- [Pnu81] PNUELI, AMIR: *A Temporal Logic of Concurrent Programs*. *Theoretical Computer Science*, 13:45–60, 1981.
- [PTN⁺07] PIETREK, GEORG, JENS TROMPETER, BENEDIKT NIEHUES, THORSTEN KAMMANN, BORIS HOLZER, MICHAEL KLOSS, KARSTEN THOMS, JUAN CARLOS FLORES BELTRAN und STEFFEN MORK: *Modellgetriebene Software Entwicklung. MDA und MDS in der Praxis*. Entwickler.Press, 2007.

- [Ras09] RASCHKE, ALEXANDER: *Translation of UML 2 Activity Diagrams into Finite State Machines for Model Checking*. In: *Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA '09)*, 2009.
- [RB70] RANDELL, B. und J.N. BUXTON (Herausgeber): *Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee*, Rom, Italien, 1970. NATO, Scientific Affairs Division.
- [RQZ07] RUPP, CHRIS, STEFAN QUEINS und BARBARA ZENGLER: *UML 2 glasklar. Praxiswissen für die UML-Modellierung*. Hanser, 3. Auflage Auflage, August 2007.
- [Sar05] SARSTEDT, S.: *Overcoming The Limitations of Signal Handling when Simulating UML 2 Activity Charts*. In: FELIZ-TEIXEIRA, J.M. und A.E. CARVALHO BRITO (Herausgeber): *Proceedings of the 2005 European Simulation and Modelling Conference (ESM '05)*, Seiten 61–65, Oktober 2005.
- [Sar06a] SARSTEDT, S.: *Semantic Foundation and Tool Support for Model-Driven Development with UML 2 Activity Diagrams*. Doktorarbeit, Universität Ulm, 2006.
- [Sar06b] SARSTEDT, STEFAN: *Model-Driven Development with ActiveCharts — Tutorial*. Technischer Bericht 2006-01, Universität Ulm, März 2006.
- [SCH02] SHEN, WUWEI, KEVIN COMPTON und JAMES HUGGINS: *A Toolset for Supporting UML Static and Dynamic Model Checking*. In: *Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment (COMPSAC '02)*, Seiten 147–152. IEEE Computer Society, 2002.
- [SG07] SARSTEDT, S. und W. GUTTMANN: *An ASM Semantics of Token Flow in UML 2 Activity Diagrams*. In: VIRBITSKAITE, I. und A. VORONKOV (Herausgeber): *Proceedings of the 6th International Andrei Ershov Memorial Conference on Perspectives of System Informatics (PSI '07)*, Band 4378 der Reihe LNCS, Seiten 207–213, Novosibirsk, Juni 2007. A. P. Ershov Institute of Informatics Systems.
- [SGK⁺05] SARSTEDT, STEFAN, DOMINIK GESSENHARTER, JENS KOHLMAYER, ALEXANDER RASCHKE und MATTHIAS SCHNEIDERHAN: *ActiveChartsIDE: An Integrated Software Development Environment Comprising a Component for Simulating UML 2 Activity Charts*. In: FELIZ-TEIXEIRA, J. M. und A. E. CARVALLO BRITO (Herausgeber): *Proceedings of the 2005 European Simulation and Modeling Conference (ESM '05)*, Seiten 66–73, Oktober 2005.
- [SH05] STÖRRLE, H. und J.H. HAUSMANN: *Towards a Formal Semantics of UML 2.0 Activities*. In: LIGGESMEYER, P., K. POHL und M. GOEDICKE (Herausgeber): *Software Engineering 2005 (SE '05)*, Band P-64 der Reihe *Lecture Notes in Informatics*, Seiten 117–128. Gesellschaft für Informatik, 2005.
- [SKRG07] SARSTEDT, STEFAN, JENS KOHLMAYER, ALEXANDER RASCHKE und DOMINIK GESSENHARTER: *ActiveChartsIDE: Eine pragmatische Umsetzung der MDA mit UML2-Aktivitätsdiagrammen*. Objektspektrum, 6, 2007.

- [SRKS05] SARSTEDT, S., A. RASCHKE, J. KOHLMAYER und M. SCHNEIDERHAN: *A New Approach to Combine Models and Code in Model Driven Development*. In: *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '05), International Workshop on Applications of UML/MDA to Software Systems*, 2005.
- [Stö05] STÖRRLE, H.: *Semantics and Verification of Data Flow in UML 2.0 Activities*. In: MINAS, M. (Herausgeber): *Proceedings of the Workshop on Visual Languages and Formal Methods (VLFM '04)*, Band 127 der Reihe *Electronic Notes in Theoretical Computer Science*, Seiten 35–52. Elsevier, 2005.
- [SVEH07] STAHL, THOMAS, MARKUS VÖLTER, SVEN EFFTINGE und ARNO HAASE: *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. Dpunkt Verlag, Mai 2007.
- [tBGKM08] BEEK, MAURICE H. TER, STEFANIA GNESI, NORA KOCH und FRANCO MAZ-ZANTI: *Formal Verification of an Automotive Scenario in Service-Oriented Computing*. In: *Proceedings of the 30th international conference on Software engineering (ICSE '08)*, Seiten 613–622. ACM, 2008.
- [tHvdA05] HOFSTEDE, ARTHUR H. TER und WIL M. VAN DER AALST: *YAWL: Yet Another Workflow Language*. *Information Systems*, 30(4):245–275, 2005.
- [TT07] TANG, CALVIN KAI FAN und EUGENIA TERNOVSKA: *Model Checking Abstract State Machines with Answer Set Programming*. *Fundamenta Informaticae*, 77(1-2):105–141, 2007.
- [Tur36] TURING, ALAN M.: *On Computable Numbers, with an Application to the Entscheidungsproblem*. *Proceedings of the London Mathematical Society. Second Series*, 42:230–265, 1936.
- [UML] *Object Management Group, UML 2.2 Superstructure Specification, Document 09-02-02, Februar 2009*.
- [VBR08] VEANES, MARGUS, NIKOLAJ BJØRNER und ALEXANDER RASCHKE: *An SMT Approach to Bounded Reachability Analysis of Model Programs*. In: *Proceedings of the 28th IFIP WG 6.1 international conference on Formal Techniques for Networked and Distributed Systems (FORTE '08)*, Band 5048 der Reihe *LNCS*, Seiten 53–68. Springer, 2008.
- [VCG⁺08] VEANES, MARGUS, COLIN CAMPBELL, WOLFGANG GRIESKAMP, WOLFRAM SCHULTE, NIKOLAI TILLMANN und LEV NACHMANSON: *Formal Methods and Testing*, Band 4949 der Reihe *LNCS*, Kapitel Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer, Seiten 39–76. Springer, 2008.
- [VSB08] VEANES, MARGUS, ANDO SAABAS und NIKOLAJ BJØRNER: *Bounded Reachability of Model Programs*. *Technischer Bericht MSR-TR-2008-81*, Microsoft Research, Mai 2008.
- [Whi04] WHITE, STEPHEN A.: *Workflow Handbook 2004*, Kapitel Process modeling notations and workflow patterns, Seiten 265–294. Future Strategies Inc., 2004.

- [Win97] WINTER, K.: *Model Checking for Abstract State Machines*. Journal of Universal Computer Science, 3(5):689–701, 1997.
- [Win01] WINTER, KIRSTEN: *Model Checking Abstract State Machines*. Doktorarbeit, Technische Universität Berlin, 2001.
- [WM97] WALICKI, MICHAŁ und SIGURD MELDAL: *Algebraic Approaches to Nondeterminism — An Overview*. ACM Computing Surveys, 29(1):30–81, 1997.
- [Wyn06] WYNN, MOE THANDAR KYAW: *Semantics, Verification, and Implementation of Workflows with Cancellation Regions and OR-joins*. Doktorarbeit, Queensland University of Technology, November 2006.
- [Yov97] YOVINE, SERGIO: *KRONOS: a verification tool for real-time systems*. International Journal on Software Tools for Technology Transfer, 1(1–2):123–133, Dezember 1997.
- [ZJ97] ZAVE, PAMELA und MICHAEL JACKSON: *Four Dark Corners of Requirements Engineering*. ACM Transactions on Software Engineering and Methodology, 6(1):1–30, 1997.