Entwicklung und Gegenüberstellung von Methoden zur automatisierten Verifikation von ausführbaren Systemspezifikationen

Dissertation

Zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt der
Fakultät für Informatik und Automatisierung
der
Technischen Universität Ilmenau

von

Dipl.-Ing. Alexander Pacholik
geboren am 3. April 1979 in Ludwigsfelde

Tag der wissenschaftlichen Aussprache: 23.04.2010

Gutachter:
1.) Uni.-Prof. Wolfgang Fengler, TU Ilmenau
2.) Uni.-Prof. Horst Salzwedel, TU Ilmenau
3.) Uni.-Prof. Kurt Lautenbach,
Universität Koblenz-Landau

urn:nbn:de:gbv:ilm1-2010000221

Kurzfassung

Für die Entwicklung komplexer eingebetteter Systeme werden Techniken eingesetzt, die eine frühe Validierung der zu entwickelnden Systeme in Bezug auf funktionale Aspekte ermöglichen. Diese Techniken greifen in der Regel auf ausführbare Spezifikationsmodelle zurück. Eingebettete Systeme stellen meist auch Echtzeitsysteme dar. Dabei sind funktionale Anforderungen in der Regel zeitvariant, also zustandsabhängig und besitzen zeitliche Randbedingungen, so dass zeitliche und funktionale Aspekte gemeinsam betrachtet werden müssen. Für die Modellierung solcher ausführbaren Spezifikationen ist der Discrete-Event Formalismus besonders geeignet, da er eine vergleichsweise abstrakte parametrisierbare Beschreibung, unabhängig von Implementierungsdetails, ermöglicht. Für den systematischen Einsatz solcher Discrete-Event Modelle als ausführbare Spezifikationen, werden Methoden und Techniken zur Verifikation benötigt, um eine möglichst frühe Fehlererkennung bei der Systementwicklung zu ermöglichen.

Gegenstand der Arbeit ist die Entwicklung und Gegenüberstellung von Methoden, die eine automatisierte Verifikation zeitbeschränkter funktionaler Eigenschaften in ausführbaren Spezifikation ermöglichen. Solche Methoden stehen für komplexe Multi-Domänen Modelle, wie sie in der Arbeit betrachtet werden, noch nicht zur Verfügung. Bei den Betrachtungen werden insbesondere die Spezifika der zugrunde liegenden Modelle und Eigenschaften sowie die möglichst automatisierte Anwendbarkeit der Verifikationsmethoden berücksichtigt.

Ausgehend von grundlegenden Erwägungen werden zwei Anwendungsszenarien entwickelt, wobei im ersten Fall die vollständige formale Verifikation im Vordergrund steht und im zweiten Fall eine dynamische Überprüfung der Eigenschaften während der szenariobasierten Simulation favorisiert wird. Für die formale Verifikation wird ein Transformation-basierter Ansatz entwickelt, der eine Transformation des Verifikationsproblems in eine mathematische Beschreibung realisiert. Für die dynamische Überprüfung der Eigenschaften wird ein Assertion-basierter Ansatz benutzt, bei dem die geforderten Eigenschaften als temporallogische Formeln notiert werden. Es werden zur Prüfung der Eigenschaften unterschiedliche Techniken zur algorithmischen Auswertung bzw. zur symbolischen Cosimulation entwickelt.

Beide Ansätze sind prototypisch für die Entwicklungsumgebung MLDesigner realisiert worden. Ausgehend von den Ergebnissen der Validierung werden Abschätzungen für das weitere Potential der Ansätze abgeleitet. Abschließend werden die Ansätze vergleichend gegenübergestellt und bewertet.

Abstract

During the development of complex embedded systems, techniques for validating functional aspects of the evolving system based on executable specifications are used. Embedded systems normally also represent real time systems. Thereby functional requirements include real time constraints and are functionally time-variant and are hence depending on internal states. Therefore temporal and functional aspects needs to be conjointly analyzed. The discrete-event formalism is particularly suitable for modeling executable specifications. That is because it enables abstract parametrizable descriptions without the need for implementation details. For a more efficient application of discrete-event models for executable specifications verification methods are needed to allow the early detection of faults.

The aim of this work is the development and comparision of methods for automatic verification of time constraint functional properties of executable specifications. Such methods are currently not available for the kind of multi-domain models considered in this work. The descriptions take the peculiarities of the regarded models as well as the automatic application of methods into account.

Based on fundamental considerations two application cases are developed. In the first case the exhaustive formal verification is considered. The second case concerns the dynamic verification of properties during scenario based simulation. For the formal verification a transformation based approach is developed, which maps the verification problem onto a mathematical model. For the dynamic verification of properties an assertion-based approach is developed, for which the required properties are given by formulas. For analyzing the regarded properties, different techniques for the algorithmic evaluation and the symbolic cosimulation are developed.

Both approaches were prototypically realized for the tool MLDesigner. Using the validated results estimations for the potentials of the approaches are made. In conclusion the approaches are compared with each other and appraised.

Danksagung

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter im Fachgebiet Rechnerarchitektur der Technischen Universität Ilmenau. An dieser Stelle möchte ich mich bei bei allen bedanken, die mich während der Erstellung dieser Arbeit sowohl fachlich als auch menschlich unterstützt haben.

In erster Line bedanke ich mich bei Univ.-Prof. Dr.-Ing. habil. Wolfgang Fengler, der mir die Tätigkeit am Fachgebiet Rechnerarchitektur ermöglichte und mich bei der Erstellung der Arbeit betreute.

Weiterhin möchte ich mich bei bei Univ.-Prof. Dr. (USA) Horst Salzwedel für die fachliche Unterstützung und eine Reihe interessanter Diskussionen bedanken.

Bei Univ.-Prof. Dr. Kurt Lautenbach bedanke ich mich für die bereitwillige Übernahme des Gutachtens.

Des weiteren bedanke ich mich bei den Mitarbeitern der Fachgebiete Rechnerarchitektur sowie Softwaresysteme/ Prozessinformatik. Allen voran bei Dipl.-Ing. Patrick Mäder, der mir bei zahlreichen Diskussionen zur Seite stand und mich fortwährend anspornte. Bei Herrn Dr.-Ing. Bernd Däne bedanke ich mich für die interessanten Diskussionen und zahlreichen Hinweise.

Ich möchte mich weiterhin bei den Kollegen des Forschungsprojektes „Funktionale Verifikation von Systemen“ (FEST) bedanken. Vor allem bei Dipl.-Ing. Stefan Lämmermann und Dr. Alexander Jesser für die intensive und fruchtbare Zusammenarbeit, auch weit über Projektende hinaus.

Eine große Unterstützung erfuhr ich durch meine ehemaligen Studenten Dipl.-Ing. Oleg Vinogradov, Dipl.-Inf. Manuela Rapp, Dipl.-Inf. Johannes Klöckner, Dipl.-Inf. Marcus Müller, Dip.-Inf Michael Stöhr, Holger Möller, Ekaterina Bondarenko und Sven Kämpf. Ohne sie die Durchführung meiner Arbeit undenkbar gewesen wäre.

Ein besonderer Dank gilt meiner Schwester Antje Pacholik, die mich fortwährend ermutigte.

Inhaltsverzeichnis

Abkürzungen **xiii**

1 Einleitung und Motivation **1**
1.1 Gegenstand der Arbeit . . . 3
1.2 Gliederung der Arbeit . . . 3

2 Wissenschaftlicher Stand **5**
2.1 Verifikationsprozess . . . 5
2.2 Entwicklungsprozess . . . 7
2.2.1 Allgemeines Entwicklungsprozess-Modell . . . 8
2.2.2 Domänenspezifische Anpassung . . . 10
2.2.3 Verifikation der Spezifikation . . . 12
2.2.4 Schlussfolgerungen . . . 15
2.3 Performancemodellierung . . . 16
2.3.1 Definition Performance . . . 16
2.3.2 Performance im Entwicklungsprozess . . . 16
2.3.3 Aspekte der Modellierung . . . 17
2.3.4 Realzeitmodellierung . . . 18
2.3.5 Modelle zur algebraischen Analyse . . . 19
2.3.6 Modell zur Analyse durch Simulation . . . 21
2.3.7 Schlussfolgerungen . . . 24
2.4 Formale Modellierungssprachen . . . 24
2.4.1 Synchrone Modelle . . . 25
2.4.2 Asynchrone Modelle . . . 26
2.4.3 Zeitbewertete Prozessmodelle . . . 28
2.4.4 Verhaltensmodellierung mit der UML . . . 30
2.5 Formale Eigenschaftsspezifikation . . . 31
2.5.1 Temporale Logiken . . . 31
2.5.2 Eigenschaftstemplates mit temporaler Logik . . . 34
2.5.3 Spezifikationssprache PSL . . . 34

2.5.4 Spezifikationssprache STL . . . 35
2.6 Verifikationstechniken . . . 36
2.6.1 Zustandsbeschreibung als Regionengraph . . . 36
2.6.2 Explizites Model Checking . . . 37
2.6.3 Symbolische Verifikation . . . 37
2.6.4 SAT basiertes Model Checking . . . 38
2.6.5 Bounded Model Checking . . . 38
2.6.6 Semiformale Verifikation . . . 38
2.7 Zusammenfassung zum Stand der Technik . . . 39

3 Ausarbeitung der Problemstellung **41**
3.1 Grundsätzliche Problemstellungen . . . 41
3.2 Abgrenzung . . . 42
3.2.1 Aspekte von Modellen . . . 43
3.3 Zielstellung . . . 43
3.3.1 Use Case Formale Verifikation . . . 44
3.3.2 Use Case Semiformale Verifikation . . . 44
3.4 Ansatz Formale Verifikation . . . 45
3.4.1 Direkte Verwendung des Systemmodells . . . 45
3.4.2 Verwendung von Annotationen zur Modellgenerierung . . . 46
3.4.3 Fragestellungen zur Transformation-basierten Verifikation . . . 47
3.5 Ansatz Semiformale Verifikation . . . 48
3.5.1 Fragestellungen zur Assertion-basierten Verifikation . . . 50
3.6 Vorgehensweise . . . 50

4 Eigenschaftsspezifikation **53**
4.1 Grundkonzept zur Eigenschaftsspezifikation . . . 53
4.1.1 Anforderungen an Eigenschaftsspezifikationen . . . 54
4.1.2 Konzept . . . 54
4.2 Klassifizierung von Eigenschaften . . . 55
4.2.1 Prozessbasierte Eigenschaften . . . 56
4.2.2 Zustandsbasierte Eigenschaften . . . 59
4.2.3 Gegenüberstellung Zustands- und Prozessbasierter Eigenschaften . . . 61
4.3 Domänenspezifische Semantik vom Eigenschaften . . . 61
4.3.1 Synchronous Data Flow (SDF) . . . 61
4.3.2 Discrete-Event (DE) . . . 62
4.3.3 Continuous Time Discrete Event (CTDE) . . . 62
4.3.4 Finite State Machine (FSM) . . . 63

4.3.5 Heterogene Modelle mit mehreren Modellierungsdomänen 63
4.3.6 Interpretation von Systemmodellen 64
4.4 Konzept zur Organisation von heterogenen Eigenschaften 66
4.5 Sprachdefinition heterogener temporaler Eigenschaften 67
4.5.1 Eventbasiertes Zeitmodell 68
4.5.2 Definition von XFLTL 69
4.5.3 Abstraktion heterogener Signale mittels Events 71
4.5.4 Interpretation analoger Eigenschaften in XFLTL 72
4.5.5 Interpretation digitaler Eigenschaften in XFLTL 73
4.5.6 Heterogene Eigenschaften in XFLTL 74
4.5.7 Zusammenfassung heterogene temporale Eigenschaften 74
4.6 Sprachdefinition nebenläufiger Prozess-basierter Eigenschaften 75
4.6.1 Sichtbarkeit der Eigenschaften 75
4.6.2 Semantik der Prozesskette als Eventfluss 75
4.6.3 Aufbau Prozessbasierter Spezifikation 77
4.6.4 Einsatz der Constraint Language 77
4.7 Zusammenfassung Eigenschaftsspezifikation 78

5 Transformation-basierte Verifikation von Systemmodellen 79
5.1 Konzept zur Transformation-basierten Verifikation 79
5.1.1 Kombinierter Transformations-/ Verifikationsalgorithmus 81
5.1.2 Hierarchie und Eigenschaften 82
5.2 Strukturanalyse 83
5.2.1 Definition 85
5.2.2 Normalisierung von kompositional-hierarchischen Modellen 86
5.2.3 Normalisierung und Hierarchische Verifikation 87
5.2.4 Eigenschaften des normalisierten Systems 87
5.3 Eigenschaftstransformation 88
5.3.1 Eigenschaftsannotation 88
5.3.2 Interpretation von DE-Modellen 91
5.3.3 Transformation der FSM Primitive 97
5.3.4 Transformation der Modellstruktur 98
5.3.5 Eigenschaftsprüfung mit Timed Automata 99
5.4 Schlussfolgerungen zur Transformation in Timed Automata 101
5.5 Prototyp zur Transformation-basierten Verifikation 101
5.5.1 Architektur 101
5.5.2 Metamodelle und Templates 102
5.5.3 Constraints und Templates 103

5.5.4 Bedienung des Prototyps . . . 104
5.6 Validierung der Methode zur Transformation-basierten Verifikation am Beispiel 105
5.6.1 Beispiel Prozessleitsystem . . . 106
5.6.2 Umsetzung in MLDesigner . . . 108
5.6.3 Ergebnisse . . . 113
5.6.4 Bewertung . . . 114
5.7 Zusammenfassung Transformation-basierte Verifikation . . . 114

6 Assertion-basierte Verifikation von Systemmodellen 117
6.1 Konzept zur Assertion-basierten Verifikation von Systemmodellen . . . 117
6.1.1 Techniken für diskrete Eigenschaften . . . 118
6.1.2 Partitionierung von XFLTL-Eigenschaften in Intervalle . . . 119
6.2 Algorithmische Prüfung von Eigenschaftstemplates . . . 121
6.2.1 Einschränkungen . . . 121
6.2.2 Operatorinstanzen . . . 123
6.2.3 Signalinstanzen . . . 125
6.2.4 Kopplung mit dem Simulationsmodell . . . 125
6.2.5 Eigenschaftsbewertung . . . 126
6.2.6 Schlussfolgerungen . . . 126
6.3 Symbolische Prüfung mittels Timed-AR-Automata . . . 127
6.3.1 Auflösung des Nichtdeterminismus . . . 127
6.3.2 Normalisierung des Zeitmodells . . . 128
6.3.3 Definition von Timed AR-Automata . . . 131
6.3.4 Konstruktion von Timed AR-Automata . . . 132
6.3.5 Interpretation heterogener Eigenschaften . . . 135
6.3.6 Schlussfolgerungen . . . 136
6.4 Prototyp zur Assertion-basierten Verifikation durch algorithmische Prüfung . . 137
6.4.1 XFLTL-Primitiv . . . 137
6.4.2 XFLTL-Klassenbibliothek . . . 138
6.4.3 Visualisierung . . . 139
6.4.4 Erweiterungsmöglichkeiten . . . 140
6.5 Validierung der Methode zur Assertion-basierten Verifikation am Beispiel . . . 141
6.5.1 Sigma/Delta-Konverter . . . 141
6.5.2 Simulationsumgebung . . . 142
6.5.3 Definition der Assertions . . . 144
6.5.4 Ergebnisse und Bewertung . . . 145
6.6 Zusammenfassung Assertion-basierte Verifikation . . . 147

7 Vergleich und Bewertung der Verifikationsansätze **149**
7.1 Bewertungskriterien . . . 150
7.1.1 Analysemodell . . . 150
7.1.2 Spezifikationssprache . . . 151
7.1.3 Verifikationstechnik . . . 152
7.2 Vergleich der Verifikationsansätze . . . 152
7.2.1 Bewertung Analysemodell . . . 154
7.2.2 Bewertung Spezifikationssprache . . . 155
7.2.3 Bewertung der Verifikationstechnik . . . 157
7.3 Zusammenfassung der Verifikationsansätze . . . 159

8 Schlussbemerkungen und Ausblick **161**
8.1 Zusammenfassung . . . 161
8.2 Diskussion der Ergebnisse . . . 162
8.3 Ausblick . . . 165

A Eigenschaftsbeschreibungssprachen **167**
A.1 Constraint Language des Transformation-basierten Ansatzes . . . 167
A.2 XFLTL Beschreibungssprache des Assertion-basierten Ansatzes . . . 173

B Anleitung Prototyp Transformation **175**

C Anleitung Prototyp Assertion Checking **179**

D Abstraktion und Verfeinerung von Prozessen **183**
D.1 Reduktionsmechanismen . . . 183
D.1.1 Verfeinerungsanalyse . . . 183
D.1.2 Intervallarithmetik . . . 184
D.1.3 Beschränkungen und offene Probleme . . . 186
D.2 Erweiterung der Modellklasse . . . 187
D.2.1 Multiple Transition . . . 187
D.2.2 Ersatzkonstruktion . . . 187
D.3 Erweiterte Verfeinerungsanalyse . . . 189
D.3.1 Umgebungsmodell . . . 189
D.3.2 Sequentielle Prozessketten und Jitter . . . 189
D.3.3 Parallele Prozesse . . . 190
D.3.4 Alternative Prozesse . . . 190
D.4 Synchronisationsbeziehungen von Prozessen . . . 190
D.5 Bewertung der Methodik . . . 191

E Zeitanalyse in nebenläufigen Pfaden - Hierarchische Analyse des Zustandsraums 193
E.1 Definition formales DE-Modell 194
E.2 Simulation des formalen DE-Modells 195
E.2.1 Simulationseigenschaften und Nichtdeterminismus 196
E.2.2 Modelleigenschaften 196
E.3 Abbildung anderer Systemklassen 197
E.4 Analysemethoden für das des DE-Systemmodell 197
E.5 Bewertung der Analysemethodik 198

Literaturverzeichnis 205

Abkürzungen

BDD engl. *Binary Decision Diagram* Kompakte, graphenbasierte Darstellung für boolesche Gleichungen.

CTL engl. *engl. Computation Tree Logic*, temporale Logik für verzweigende Pfade, siehe Abschnitt 2.5.1.

CTL* Obermenge der temporalen Logiken LTL und CTL, siehe Abschnitt 2.5.1.

DAG engl. *Directed Acyclic Graph*, gerichteter azyklischer Graph.

DBM Datenstruktur zur Beschreibung von Regionengraphen (Clock Zones) für das Model Checking zeitbehafteter Systeme.

FSM engl. *Finite State Machine* Domäne, Simulationsdomäne für zustandsdiskrete Modelle, siehe Abschnitt 4.3.4.

HDL engl. *Hardware Description Language*, als HDL kann jede Programmier- und Beschreibungssprache bezeichnet werden, die für eine formale Beschreibung von (digitaler) Hardware geeignet ist.

HOL Ansatz zur Beschreibung und Analyse von Modellen durch eine Logik höherer Ordnung, die durch die Definition von Theoremen und deren Anwendung (logisches Schließen) den Beweis von Aussagen ermöglicht.

KDNF *Konjunktiv Disjunktive Normalform*, Normalform für boolesche Aussagen.

KSD *Kompositionsstrukturdiagramm* (engl. composite structure diagram) der UML 2.0.

LTL *engl. Linear Temporal Logic, temporale Logik für lineare Pfade, siehe Abschnitt 2.5.1.*

MDA engl. *Model Driven Architecture*, Ansatz der Object Management Group zur modellgetriebenen Softwareentwicklung.

MLD *MLDesigner,*Entwicklungsumgebung zur Modellierung und Analyse von Systemen, welche die kombinierte Verwendung mehrerer Berechnungsdomänen unterstützt.

MoC engl. *Model of Computation*, Bezeichnung für eine Klasse von Modellen, welche im mathematischen Sinne die selbe Simulationssemantik aufweisen.

MTBF engl. *Mean Time Between Failure*.

MTTR engl. *Mean Time To Repair*.

OCL engl. *Object Constraint Language*, Sprache zur Beschreibung von Restriktionen und Eigenschaften in UML-Modellen.

ODE engl. *Ordinary Differential Equation*, engl. gewöhnliche Differentialgleichung.

PSL engl. *Property Specification Language*, Beschreibungssprache für temporale Eigenschaften in getakteten digitalen Systemen, wie z.B. VHDL und Verilog [Acc04]..

SDL engl. *Specification and Description Language*, Spezifikations- und Beschreibungssprache, die vor allem im Telekommunikationsbereich eingesetzt wird.

SERE engl. *Sequential Extended Regular Expressions*, Beschreibung linearer sequentieller Signalverläufe in digitaler Logik.

SOI engl. *System Of Interrest*.

STL Sprache zur Beschreibung analoger Eigenschaften. Wurde im Rahmen des Projektes PROSYD als Ergänzung zur PSL entwickelt.

UML engl. *Unified Modeling Language*, hier in der Version 2.0 [OMG04].

VDHL Hardwarebeschreibungssprache für digitale Systeme. Inzwischen existieren mit VHDL-AMS auch Erweiterungen für analoge Systeme.

VDM Methode zur formalen Spezifikation und Entwicklung von Computer-Programmen.

WCET engl. *Worst Case Execuion Time*, Zeitparameter für die längste (kürzeste) mögliche Ausführungszeit von Prozessen, bzw. für das Ausführungsintervall.

XFLTL engl. *Extended Finite Linear Temporal Logic*, Logik zur Beschreibung Event-basierter temporaler Eigenschaften.

1. Einleitung und Motivation

Die Entwicklung eingebetteter Systeme stellt besondere Anforderungen an den Entwicklungsprozess, die aus den Einsatzbedingungen des zu entwickelnden System resultieren. Die Systeme sollten wartungsfrei oder zumindest wartungsarm arbeiten, müssen folglich möglichst fehlerfrei und zuverlässig sein. Viele eingebettete Systeme stellen gleichzeitig Echtzeitsysteme dar, fordern also neben der funktionalen Korrektheit auch die Rechtzeitigkeit der Berechnungen bzw. Interaktionen. Sicherheitskritische Systeme repräsentieren oft eine Kombination aus Echtzeitsystemen mit gesteigerten Anforderungen an die Zuverlässigkeit, Verfügbarkeit, Wartbarkeit und Sicherheit. Der Entwicklungsprozess sicherheitskritischer Systeme muss also in systematischer Weise den Nachweis von Korrektheit und Rechtzeitigkeit der Berechnungen bzw. Interaktionen der zu entwickelnden Systeme unterstützen.

Weiterhin existieren Anforderungen beim Management der Unsicherheiten, die vor allem bei der Entwicklung komplexer sicherheitskritischer Systeme bestehen. Moderne Entwicklungsstrategien nutzen ausführbare Spezifikationen, um eine Kommunikationsplattform zwischen Designern und Stakeholdern zu etablieren. Weiterhin ermöglicht dies eine Validierung der Spezifikation, sowohl auf der Anwenderebene als auch auf der technischen Ebene. Die Validierung erlaubt die Analyse von Unvollständigkeiten und Wechselwirkungen auf der funktionalen Ebene. Bezieht man die Systemarchitektur in die ausführbare Spezifikation ein, so wird auch eine Validierung der Systemarchitektur ermöglicht.

Beim Entwurf eingebetteter Systeme werden in starkem Maße Protokolle auf der Applikationsebene eingesetzt. Zahlreiche Beispiele zeigen, dass selbst einfache Protokolle bzw. Zustandsmaschinen, etwa durch Erweiterung auf größere Systeme, eine hohe Komplexität erreichen können, die sich durch Beschreibung und Simulation nicht vollständig beherrschen lässt. Der Einsatz von Verifikationstechniken dient der Vermeidung von Fehlern, z.B. bei der Anwendungslogik und applikationsspezifischen Protokollen. Dadurch ist eine Ersparnis von Entwicklungsaufwand, Zeit und Geld erreichbar. Die Entwicklung in der Chipentwicklung zeigt die Vorteile der systematischen Anwendung von Verifikationstechniken. Zum sicheren Entwurf ist die Nutzung von Verifikationstechniken unerlässlich.

Schaut man sich die verfügbaren Techniken zur Modellierung, Simulation und Verifikation von Systemmodellen an, so stellt man fest, dass in jedem dieser Bereiche unzählige Ansätze existieren. Ausgehend von der Unified Modeling Language (UML) und deren Erweiterungen, über domänenspezifische Modellierungskonzepte, wie Specification and Description Language

(SDL) bis hin zu mathematischen Modellen, wie Petri-Netze, Spezifikationssprache Z und Prozessalgebra, existieren Ansätze zur Spezifikation von Systemen. Während für die UML keine verbindliche Semantik zur Ausführung komplexer Modelle besteht, ist die Semantik der Berechnung für domänenspezifische und mathematische Spezifikationssprachen definiert.

Betrachtet man die Anforderungen aus der Sicht des Systementwurfs, so zeigt sich, dass bei den Spezifikationssprachen, die eine Verifikation temporaler Eigenschaften ermöglichen, Diskrepanzen bestehen. Im Falle der rein zustandsbasierten Modelle, einschließlich Petri-Netzen, ist die Modellierung komplexer Datentypen, die eine synchronisierten Austausch komplexer Informationen erlauben, nicht möglich, oder aber für die Verifikation nicht verwendbar.

Generell besteht derzeit eine Diskrepanz zwischen der Ausdrucksmächtigkeit von Simulatoren auf der einen und Verifikationswerkzeugen auf der anderen Seite. Für die Modellierung von Systemmodellen hat sich die Nutzung heterogener Modelle, also Modellen, die sich verschiedener semantischer Konzepte bedienen, als besonders geeignet erwiesen [Sal04]. Geeignete Eigenschaftsbeschreibungssprachen für solche Modelle sind bisher nur unzureichend untersucht worden. Ein wesentliches Problem besteht dabei in der Vielfältigkeit der Sichtweisen und Modellierungsalternativen für (Echtzeit-)Eigenschaften, bei der Modellierung und Analyse von Systemen. So ist die Performance als Ausführungszeit für funktionale Prozesse interessant, sofern eine orthogonale Betrachtung von Zeit und Funktionalität möglich ist. Weiterhin sind vom Standpunkt der funktionalen Verifikation Fragestellungen der Erreichbarkeit kritischer Zustände oder Abläufe von Interesse.

Zwar existieren zahlreiche Methoden zur Analyse von Echtzeitsystemen, allerdings ist die Integration solcher Methoden in die Entwicklungsprozesse sowohl organisatorisch als auch technisch nur unzureichend gelöst. Ferner sind Fragestellungen der Konsistenz, im Hinblick auf die unterschiedliche Ausdrucksmächtigkeit der zur Validierung, Analyse und Verifikation verwendeten Modelle, noch weitestgehend ungelöst.

Insbesondere für komplexe Systeme kann es passieren, dass eine kompositionale Worst Case Analyse statisch betrachtet möglicherweise zu überapproximierten Ergebnissen führt, da die einzelnen Teile nicht zwangsweise unabhängig voneinander sind. Abbildung 1.1 stellt diesen Sachverhalt als Relation zwischen Erreichbarkeitsmengen schematisch dar. Informationstechnisch betrachtet sind die Erreichbarkeiten von Komponenten nicht orthogonal (Zustandsraum Z_{max}), sondern stellen, ausgehend vom Initialzustand Z_0, nur eine Teilmenge Z^*, Z_t^* dar ($Z_t^* \subseteq Z^* \subseteq Z_{max}$). Wechselwirkungen nebenläufiger Komponenten können dabei funktionaler Natur sein (Zustandsraum Z^*), oder zusätzlich durch Architektur bzw. Performance Constraints bedingt sein (Zustandsraum Z_t^*).

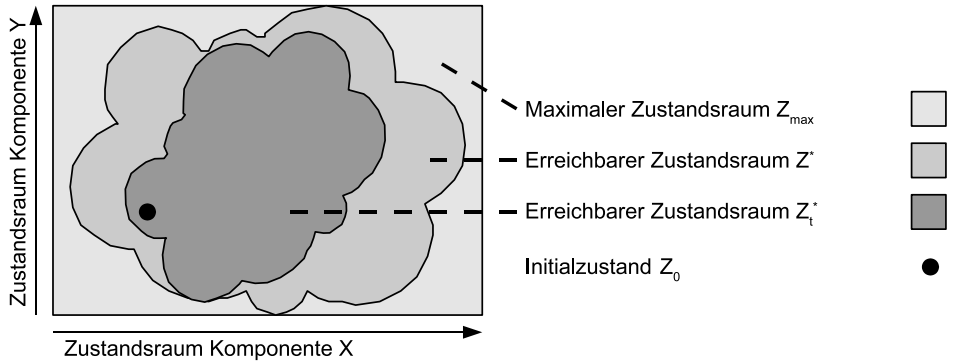


Abbildung 1.1.: Kombination von Zustandsräumen nebenläufiger Komponenten

1.1. Gegenstand der Arbeit

Gegenstand dieser Arbeit ist die Untersuchung von Methoden zur Beschreibung und Analyse von Echtzeiteigenschaften sowie die Integration mit Multi-Domänen-Modellen, wie sie zur Beschreibung innerhalb ausführbarer Spezifikation verwendet werden. Zielstellung ist dabei insbesondere die Beschreibung und Prüfung zeitbeschränkter temporaler Eigenschaften, also die Dynamik in heterogenen Systemen.

Für die Untersuchungen werden folgende Fragestellungen aufgestellt:

1. Welchen Charakter haben die als ausführbare Spezifikationen verwendeten Modelle?
2. Wie lassen sich Zeiteigenschaften spezifizieren?
3. Welche Anforderungen bestehen an eine Spezifikationssprache für Zeiteigenschaften, so dass diese leicht in Spezifikationsmodelle integriert werden kann?
4. Welche bestehenden Methoden sind für die Prüfung von Zeiteigenschaften geeignet?
5. Inwiefern lassen sich Ansätze zur kompositionalen Analyse von Zeiteigenschaften nutzen?
6. Wie lassen sich Verifikationstechniken für Zeiteigenschaften in den Entwicklungsprozess technisch und organisatorisch integrieren?

1.2. Gliederung der Arbeit

Diese Arbeit ist wie folgt gegliedert. Zunächst wird im Kapitel 2 der Stand der Technik zu ausgewählten Aspekten dargestellt. Dabei werden Anforderungen an die Verifikation in frühen Entwicklungsphasen aus dem Entwicklungsprozess abgeleitet, bestehende Methoden zur Performancemodellierung werden näher betrachtet und analysiert, Techniken zur Verifikation und zugrunde liegende Spezifikationssprachen werden detailliert dargestellt. Aus den Untersuchungen

zum Stand der Technik werden im Kapitel 3 detaillierte Problemstellungen zum Themengebiet der automatisierten Verifikation von Zeiteigenschaften in ausführbaren Systemspezifikationen abgeleitet. In Kapitel 4 erfolgt schließlich die Definition der verwendeten Spezifikationssprachen für dynamische Eigenschaften, die in den nachfolgenden Kapiteln angewandt wird. Im Kapitel 5 wird ein Ansatz der Verifikation von Systemen unter Zuhilfenahme einer Transformation in eine mathematische Darstellung (Timed Automata) beschrieben. Im Kapitel 6 wird ein ergänzender Ansatz der simulationsbegleitenden Verifikation vorgestellt, bei dem Monitore, ähnlich den Timed Automata zur Prüfung von Discrete-Event-Modellen verwendet werden. Eine Gegenüberstellung der Ansätze aus Kapitel 5 und 6 erfolgt in Kapitel 7. In Kapitel 8 werden die Ergebnisse und Erkenntnisse der Arbeit zusammengefasst und bewertet.

2. Wissenschaftlicher Stand

In diesem Kapitel werden die Grundlagen und der Stand der Technik zu den Themengebieten Modellierung und Verifikation dargestellt. Dazu gehören eine Erläuterung der wissenschaftlich-technischen Grundlagen sowie eine auszugsweise Darstellung vorangegangener Arbeiten aus dem jeweiligen Themengebiet.

Zunächst wird der Begriff des Verifikationsprozesses (Abschnitt 2.1) definiert und in Zusammenhang mit den organisationstechnischen Aspekten des Themengebietes der Entwicklungsprozesse (Abschnitt 2.2) betrachtet, da die Analyse der bereitstehenden Informationen im Verlauf der Entwicklung eines Systems ein wichtiger Aspekt der Analyse darstellt. Anschließend wird die Performancemodellierung (Abschnitt 2.3) betrachtet, da zahlreiche Techniken existieren, die die Zeitnutzung in Abhängigkeit von der Nutzung von Ressourcen statisch durch konservative Approximation beschreiben. Dem schließen sich in Abschnitt 2.4 Betrachtungen zu Modellierungssprachen zur Beschreibung des funktionalen und temporalen Systemverhaltens an, welche im Verlauf der Systementwicklung durch ihre Simulierbarkeit die Grundlage der möglichen Analysen bilden. Im Abschnitt 2.5 werden Sprachen zur formalen Eigenschaftsspezifikation betrachtet, die, im Vergleich zu den Modellierungssprachen eine nicht ausführbare Spezifikation darstellen. Abschließend wird in Abschnitt 2.6 die Verifikation als Kerngebiet betrachtet. Dabei werden zuerst Eigenschaftsbeschreibungssprachen, als Grundlage der Verifikationstechniken betrachtet und anschließend verschiedene Techniken zur Verifikation der Eigenschaften dargestellt. Abschließend werden die Erkenntnisse des Kapitels in Abschnitt 2.7 zusammengefasst.

2.1. Verifikationsprozess

Während der Systementwicklung sollten Aktivitäten zur Entwicklung und Aktivitäten zur Verifikation nahtlos ineinandergreifen [Ben05]. Dies setzt voraus, dass die Methoden und Techniken, die zur Entwicklung bzw. zur Verifikation genutzt werden aufeinander abgestimmt sind. Diese Abstimmung ist nur im Kontext einer Anwendungsdomäne möglich, so dass für verschiedene Anwendungsdomänen unterschiedliche Lösungen entwickelt wurden. Die auf den Entwicklungsprozess abgestimmten Aktivitäten der Verifikation, inklusive der verwendeten Methoden und Techniken werden im Folgenden als Verifikationsprozess bezeichnet. Dazu werden insbesondere die Wechselwirkungen zwischen Entwicklungsprozess und Verifikationsprozess definiert, die zwischen den genutzten Modellen aufgrund gemeinsam genutzter Informationen bestehen.

In der Literatur sind Überlegungen zum Verifikationsprozess, verglichen mit Betrachtungen zum Entwicklungsprozess selbst, nur vereinzelt zu finden. Besonders hervorzuheben ist *B-method* [Abr96, Lan96, Engb] von Jean-Raymond Abrial, ein werkzeuggestützter Ansatz zur Spezifikation, Entwicklung und Verifikation sicherheitskritischer Systeme basierend auf abstrakten Zustandsmaschinen. Durch Weiterentwicklungen, wie z.B. *EventB* und *UML-B* wurde der Ansatz für zahlreiche Anwendungsdomänen erweitert. Hervorzuheben ist hierbei die gute, teils kommerzielle Werkzeugunterstützung durch *AtelierB* [Enga], sowie die zunehmende Etablierung in Bereichen mit strengen sicherheitskritischen Anforderungen. Für die Sprache SDL [EHS97], die auf kommunizierenden Prozessen (Zustandsmaschinen) basiert, existieren ebenfalls Werkzeuge zur Verifikation, allerdings ist der Einsatzbereich dieser Sprache im wesentlichen auf den Telekommunikationssektor und die Modellierung von Protokollen beschränkt, da ein spezielles Kommunikationsschema genutzt wird, das auf Warteschlangen basiert und daher besonders für (paketvermittelnde) Kommunikationsnetze und -protokolle geeignet ist. Die Sprache *Z* [Spi89], basiert auf Mengenlehre und Prädikatenlogik und erlaubt die Spezifikation und Verifikation. Die mit dem Ansatz Vienna Development Method (VDM)++ [Fit05] steht ein mathematischer Ansatz mit operationaler Semantik, basierend auf typisierten Mengen zur Verfügung. Der Ansatz ist zwar gut für allgemeine, objektorientierte System- und Softwareentwicklung geeignet, bietet aber keine Unterstützung für Echtzeitsysteme.

Es existieren noch zahlreiche weitere Ansätze, bei denen die Spezifikation des Systems in der Regel durch die Komposition von speziellen Zustandsmaschinen erfolgt, während die zu prüfenden Eigenschaften in Form (temporaler) Logik notiert werden. In einer großangelegten Studie [ABL96] aus den Jahren 1994/1995 beschreiten nahezu alle Ansätze diesen Weg, auch wenn sich Notation, Semantik und Umsetzung deutlich unterscheiden.

Andere Ansätze beschränken die erlaubten Methoden und Techniken des Entwicklungsprozesses zum Zwecke der Verifikation, konnten sich aber in der Praxis u.a. aufgrund dieser Einschränkungen nicht durchsetzten. In [Nüt99] wird eine Methode definiert, die auf Komponenten basiert, die mittels Petri-Netzen beschrieben werden. Die Methode erlaubt zwar die Generierung und Verifikation von Modellen, allerdings entstehen dadurch auch Beschränkungen bezüglich der für die Implementierung verwendbaren Werkzeuge. In [Lic04] wird ein Verfahren vorgestellt, dass die zur Beschreibung von Automatisierungssystemen verwendeten Modelle durch Annotationen erweitert und für die Verifikation nutzt. Allerdings wurde die Verifikation nur manuell umgesetzt, ist also nur beschränkt wiederverwendbar.

Festzuhalten ist, dass sich in Bezug auf den Verifikationsprozess bisher keine geeignete formale Spezifikationssprache durchsetzen konnte. Die Ursachen sind in den Schwächen der existieren Ansätze zu finden [Lam00]:

- Die meisten formalen Spezifikationssprachen sind zu limitiert bezüglich der Anforderungen des Entwicklungsprozesses.

- Schlechtes Management der Bedeutung von Spezifikationen. Eine Unterscheidung zwischen gewünschten Eigenschaften des Systems, Annahmen über die Umwelt und Eigenschaften der Domäne sind meist nicht explizit möglich.
- Die Begrifflichkeiten der formalen Spezifikation sind auf niedriger Ebene angesiedelt und entsprechen nicht denen des Entwicklungsprozesses.
- Die meisten Sprachen sind technisch nicht ausreichend in den Entwicklungsprozess integriert. Insbesondere fehlt die Verknüpfung mit anderen Artefakten des Entwicklungsprozesses.
- Es fehlen methodische Anleitungen, wie formale Spezifikationen einzusetzen sind. Zudem sind die Sprachen in der Syntax sehr heterogen, oft auch kompliziert.
- Es ist ein hohes Maß an Expertise für die Erstellung formaler Spezifikationen erforderlich, die zu hohen Kosten führen kann.
- Analyseergebnisse sind nur schwer verwertbar, die Suche nach Ursachen von Fehlern muss weitestgehend manuell erfolgen.

Am ehesten ist die *B-method* geeignet, konnte sich bisher allerdings noch nicht vollständig durchsetzen.

Am weitesten Vorangeschritten sind Ansätze zum Assertion-basierten Design von digitaler Hardware [FKL04]. Dabei werden Empfehlungen definiert, wie Eigenschaften notiert und verwendet werden. Diese Eigenschaften können anschließen für die Verifikation verwendet werden, ohne dass Einschränkungen im Entwicklungsprozess bestehen.

Für die frühen Phasen eines Entwicklungsprozesses sind solche Methoden nur in Kombination mit spezialisierte formalen Beschreibungsmitteln, etwa Petri-Netzen, Timed Automata, SDL, und Prozessalgebra zu finden[1], die sich aus Anwendersicht nur eingeschränkt zum Zwecke der Validierung verwenden lassen, da die Ausdrucksmächtigkeit im Vergleich zu anderen simulierbaren Beschreibungstechniken beschränkt ist.

2.2. Entwicklungsprozess

Ziel dieses Kapitels ist die Darstellung des Systementwurfs bzw. des zugrunde liegenden Entwicklungsprozesses unter dem Aspekt der Verifikation. Dazu gilt es insbesondere die bereitgestellten Informationen und Eigenschaften zu betrachten.

Für die Entwicklung eingebetteter Systeme sind eine ganze Reihe von Entwicklungsprozessen (Vorgehensmodelle) definiert worden. Das Wasserfall-Modell und V-Modell sind nur einige

[1] siehe auch Abschnitt 2.4

davon. Es muss jeweils zwischen dem allgemeinen Entwicklungsprozess-Modell und dem konkreten Entwicklungsprozess unterschieden werden. Die allgemeinen Entwicklungsprozess-Modelle zählen *grundlegende Aktivitäten* auf und stellen diese in Bezug zueinander, beschreiben aber nicht, wie diese realisiert werden. Beispielsweise wird dargestellt, dass es einen Prozess der Anforderungsanalyse gibt und die aufgestellten Anforderungen in die Konzeption einfließen. Für einen konkreten Entwicklungsprozess müssen dagegen auch die technischen Realisierungen beschrieben werden. Zur Anwendung des allgemeinen Prozessmodells ist eine domänenspezifische (fachlichen) Anpassung notwendig, bei der z.B. festgelegt wird, welche Informationen bei der Anforderungsanalyse gesammelt werden und wie dies geschehen soll. Zur Umsetzung des Prozessmodells bedarf es in jedem Falle einer Akkreditierung. Bei der Akkreditierung wird festgelegt, wie die Umsetzung des Prozessmodells technisch realisiert wird. In welcher Form werden Anforderungen notiert, welche Werkzeuge kommen zum Einsatz, etc.

Es kann also festgestellt werden, dass allgemeine Entwicklungsprozess-Modelle nur die prinzipielle Verfahrensweise beschreiben und nicht konkret genug sind, um Aussagen über die Verfügbarkeit bestimmter domänenspezifischer Informationen zu machen. Solche Aussagen lassen sich erst nach einer fachlichen Anpassung des Entwicklungsprozesses treffen. Der Einsatz eines Entwicklungsprozesses wird dagegen erst mit einer Akkreditierung ermöglicht.

Für die weiteren Betrachtungen in dieser Arbeit soll, entsprechend der getroffenen Eingrenzung, siehe Abschnitt 1.1, die Domäne eingebetteter Hardware/Softwaresysteme vorausgesetzt werden. Ferner wird kein kompletter Entwicklungsprozess beschrieben, da nur frühe Phasen der Systementwicklung im Fokus dieser Arbeit liegen. Dementsprechend werden Teile eines abstrakten Entwicklungsprozess-Modells vorgeschlagen und eine mögliche fachliche Umsetzung, speziell die Methode der Validierung von Spezifikationen durch ausführbare heterogene Modelle, erläutert.

2.2.1. Allgemeines Entwicklungsprozess-Modell

In diesem Abschnitt werden allgemeine Zusammenhänge im technischen Entwicklungsprozess erläutert. Kaufmännische Fragestellungen, wie Markt- und Risikoanalyse sowie entsprechende Machbarkeitsanalysen auf der technischen Seite bleiben dabei außen vor. Es wird die eigentliche Umsetzung, beginnend mit Anforderungen von einem Auftraggeber betrachtet.

Aus der Sicht des Systems Engineering kann die Entwicklung des System Of Interrest (SOI), entsprechend Abbildung 2.1, schrittweise unter vier Aspekten betrachtet werden [Was05]:

1. Anforderungsphase: Was sind die Randbedingungen und Beschränkungen, bezüglich der Aufgaben, denen das System, Produkt oder der Service innerhalb einer vorgeschriebenen Einsatzumgebung unterliegt?

2. Spezifikationsphase: Bei gegebenen Anforderungen, wie soll das System, Produkt oder der

Service eingesetzt, genutzt und gewartet werden, um die Aufgaben innerhalb bestimmter Zeitschranken zu erreichen?

3. Verhaltens-/Designphase: Bei gegebenen Einsatz, Verwendung, Wartung, und Zeitbeschränkungen für das geplante System, Produkt oder Service, wie sieht funktional betrachtet die Menge der Verhalten und Antworten aus, die das geforderte System aufweisen muss, um seine Aufgaben zu erfüllen?

4. Implementierungsphase: Bei gegebener Menge an funktionalem Verhalten und Antworten, die das System zur Erfüllung der Aufgaben aufweisen muss, wie muss das fertige System, Produkt oder Service physisch implementiert sein, um die Aufgaben zu erfüllen?

Abbildung 2.1 stellt diesen Zusammenhang grafisch dar. Links sind die zu entwickelnden Artefakte dargestellt. Rechts sieht man den Zusammenhang zwischen Entwicklungsphasen, Detailgrad und Projektlaufzeit.

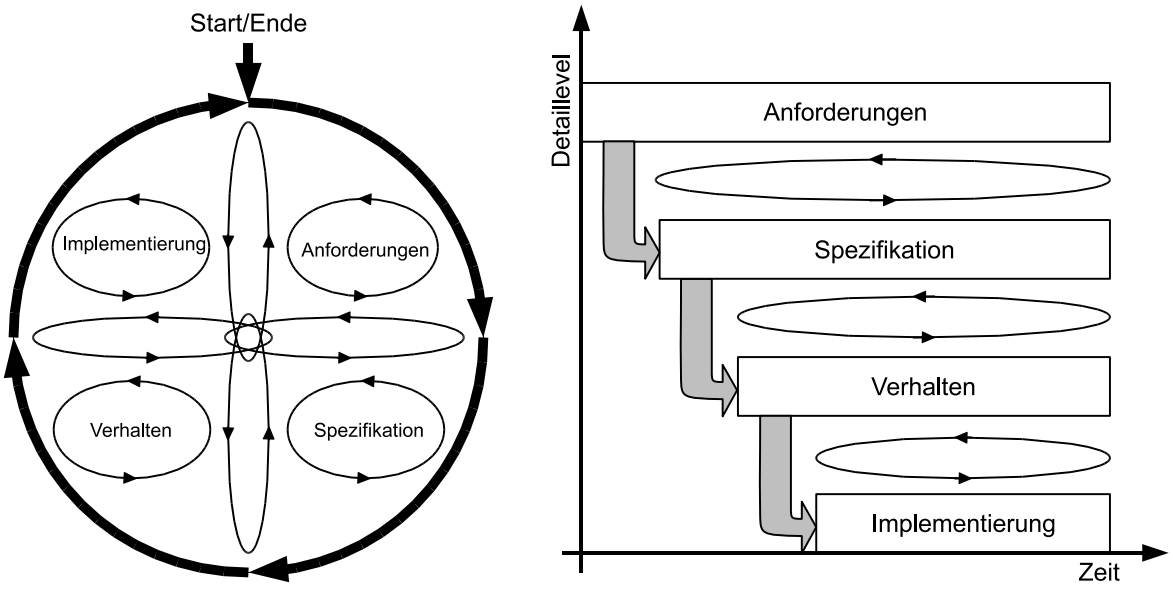


Abbildung 2.1.: Entwicklungsprozess im Systems Engineering nach [Was05]

Zunächst wird das SOI analysiert. Randbedingungen, Umgebung, Beschränkungen und Constraints werden definiert. Der mögliche Lösungsraum wird damit abgegrenzt. In der zweiten Phase werden die notwendigen Missionen und Szenarios (Use Cases) des SOI untersucht. Dies schließt eine Analyse operativer Funktionalitäten ein, die unterstützend zu den eigentlichen Missionen realisiert werden müssen. In der dritten Phase wird das erwartete Verhalten des SOI bezogen auf die Use Cases, sowie die Interaktion mit seiner Umgebung beschrieben. In der vierten Phase wird die physische Implementierung des SOI beschrieben. Dabei wird insbesondere die Architektur eines Systems betrachtet.

Eine solch strikte Trennung der Phasen ist in der Regel nur schwer möglich, da es starke Wechselwirkungen gibt. Bei eingebetteten Hardware/Softwaresystemen bestehen insbesondere Wechselwirkungen zwischen der Architektur, Technologie, Verhalten und Performance [Tei97, Ben05]. Die Realisierung (Architektur) von Verhalten durch verschiedene Technologien kann dabei die Performance wesentlich beeinflussen. Daher ist es sinnvoll, die grundlegende Architektur eines Systems vor der eigentlichen Implementierung zu analysieren (Tadeoff Analyse, Design Space Exploration), um eine ausgewogene Lösung sicherzustellen (Pareto-Optimierung). Anschließend kann das technologieabhängige Verhalten und die physikalische Umsetzung betrachtet werden.

Die beiden letzten Phasen werden dabei zweimal durchlaufen. Zunächst als Konzeption/Analyse und anschließend als Implementierung. In dem Konzeptionszyklus wird eine funktionale Lösung erarbeitet und anschließend die Wechselwirkungen zwischen Funktion, Architektur und Technologie modellbasiert untersucht. Auf der Grundlage dieser Untersuchungen können objektive Entscheidungen für oder gegen bestimmte Kombinationen aus Architektur und Technologie abgeleitet werden. Die grundlegende Architektur wird entsprechend der gegebenen Variabilität untersucht und festgelegt. In der Implementierungsphase erfolgt die technologieabhängige die Umsetzung des Systems [BHS07].

Es ist also erforderlich, sämtliche Randbedingung und Constraints bei dem Konzeptionszyklus zu berücksichtigen und die Szenarios (Use Cases) sowie die benötigten Funktionalitäten vollständig zu erfassen. Geschieht dies nicht, bleiben funktionale Wechselwirkungen unentdeckt oder Randbedingungen werden verletzt, so dass eine erfolgreiche Systementwicklung nicht gewährleistet werden kann.

Prinzipiell schwierig zu beantworten ist die Frage der Exaktheit der Modellierung. Einerseits muss darauf geachtet werden, dass der Abstraktionsgrad eingehalten wird, also beispielsweise Implementierungsdetails nicht in die Spezifikationsphase eingebracht werden. Andererseits müssen Entscheidungen der Designphase validiert werden. Dazu sind oftmals Detailuntersuchungen erforderlich, wenn eine technologieabhängige Performance einer Funktion berücksichtigt werden muss. Hierzu können Verfahren zum Rapid-Prototyping oder detaillierte Simulationen (z.B. Instruction Set Simulation) genutzt werden.

2.2.2. Domänenspezifische Anpassung

Wie die Phasen des Entwicklungsprozesses umgesetzt werden, also welche Modellierungstechniken und Beschreibungsmittel zum Einsatz kommen, ist von der konkreten Entwicklungsdomäne und den Randbedingungen des konkreten Entwicklungsprojektes abhängig. Es bedarf also einer domänenspezifischen Anpassung. Allgemein bestehen Bestrebungen, einen möglichst vollständig modellbasierten Ansatz zu verwenden, um die Konsistenz der Informationen und Modelle sicherzustellen und um eine Weiterverarbeitung erfasster Informationen zu ermöglichen. Dadurch wird ein Nachvollziehen und Überprüfen des Entwicklungsprozesses ermöglicht. Dem

stehen jedoch ein erhöhter Aufwand für die Organisation der Entwicklungsprozesse und ein erhöhter Aufwand für die Modellierung gegenüber, der unter anderem durch die Einarbeitung in die verwendeten Beschreibungsmittel bedingt ist.

Für die weiteren Betrachtungen in dieser Arbeit ist nicht der gesamte Entwicklungsprozess von Bedeutung, sondern nur der Konzeptionszyklus, bestehend aus der Anforderungsanalyse. Die Anforderungsanalyse stellt Missionen und Szenarien sowie die dazugehörigen Randbedingungen und Constraints zur Verfügung und modelliert Wechselwirkungen zwischen Technologie und Architektur. Mit der UML 2.0 [OMG04] und der daran angelehnten Beschreibungssprache SysML [Par05] stehen geeignete Notationen dafür bereit:

- Missionen bzw. Szenarien des SOI können als Use Cases abgelegt werden.
- Die grundlegenden Klassen und Objekte zur Realisierung bzw. aus der Umgebung des SOI können mittels des Klassendiagramms (SysML Blockdefinitionsdiagramm) modelliert werden.
- Für die Modellierung von Anforderungen steht in der SysML mit dem Anforderungsdiagramm ein eigenes Beschreibungsmittel bereit.
- Über Aktivitätsdiagramme lassen sich Abläufe beschreiben, Interaktionen über Sequenzdiagramme und Modi über Zustandsdiaramme.

Nachteil der Modellierung mit UML/SysML ist, dass sie im Allgemeinen nicht ausführbar sind, und somit eine Evaluierung erst spät möglich ist. Zwar ist nach entsprechender ausführlicher Modellierung eine (teilweise) Generierung von Quellcode möglich, die Verbindung zum eigentlichen UML/SysML Modell geht dann aber verloren[2].

Sinnvoll ist daher eine Validierung und Analyse auf der Basis eines Simulationsmodells. Auf der Basis ereignisbasierter/zustandsbasierter Modelle lässt sich das Verhalten von Funktionen auch unter dem Einfluss architektureller Randbedingungen simulieren, wodurch eine Analyse von Performanceparameter ermöglicht wird.

Kritisch ist dabei vor allem die Parametrisierung eines Modells [BCNN01] [WK04]. Die Gewinnung von Messwerten für Performanceparameter ist oft erst nach einer (prototypischen) Implementierung möglich. Alternativ können zwar Erfahrungswerte oder Abschätzungen genutzt werden, die Ergebnisse der darauf basierenden Analyse bleiben damit allerdings fraglich. Zunehmend werden Ansätze etabliert, die eine Codegenerierung erlauben, und somit, zumindest für Teilsysteme, eine effiziente Möglichkeit zur Gewinnung von Performanceparametern bieten.

[2] Einige Tools, wie Rhapsody, bieten einen Modus, der die grafische Animation des generiertes Quellcodes erlaubt. Allerdings wird das Modell dabei nicht symbolisch betrachtet und es wird eine proprietäre Schnittstelle verwendet.

Für die Modellierung technologieabhängiger Effekte in eingebetteten Systemen ist die Nutzung von Datenflussmodellen für Hardware sowie die Nutzung von kontinuierlichen (analogen) Teilmodellen für die Modellierung technischer Prozesse erforderlich. Basierend auf verschiedenen Parameterkonfigurationen und Architekturvarianten können so relativ leicht Designentscheidungen getroffen werden.

Die Kombination aus verschiedenen Simulationsstrategien (Model of Computation (MoC)) [JS05b, JS05a] hat sich dabei als vorteilhaft erwiesen. Eine Integration verschiedener MoC ist beispielsweise in den Simulationsframeworks MLDesigner sowie PtolemyII umgesetzt. Für ausgewählte MoC (CTDE, SDF, DE) ist eine kombinierte Etablierung auch in anderen kommerziellen Entwicklungswerkzeugen abzusehen (VHDL-AMS, SystemC, Matlab/Simulink, Labview).

2.2.3. Verifikation der Spezifikation

Das Ziel der Verifikation von Spezifikationen besteht darin, dass die Erfüllung geforderter Eigenschaften formal sichergestellt wird. Dies beinhaltet die syntaktische Korrektheit der beteiligten Modelle, also die Konsistenz und Vollständigkeit sämtlicher Datenobjekte (physikalische / informationstechnische Objekte) inklusive Definitionsbereich und Referenzen zu deren Herkunft (Anforderung/ Modelle/ Standards). Weiterhin muss die funktionale Korrektheit, sowie die Einhaltung von zeitlichen Randbedingungen sichergestellt werden.

Das Erreichen dieses Ziels kann durch ausführbare Spezifikationen realisiert werden. Zweck solcher ausführbaren Spezifikationen ist die vollständige Darstellung sämtlicher benötigten Funktionalitäten, deren Schnittstellen und Performanceanforderungen sowie Referenzen zu Quellen der Anforderungen. Durch Simulation kann die ausführbare Spezifikation von Funktion und Architektur bei der Interaktion mit der Systemumgebung validiert werden. Bei Verwendung formaler Modelle wird zusätzlich die Verifikation zeitlicher Randbedingungen sowie die Prüfung bezüglich logischer Inkonsistenzen und verbotener Zustände ermöglicht.

Plattformbasierter Entwurf

Aktuelle Entwicklungswerkzeuge, wie Matlab/Simulink und Labview unterstützen einen Plattformbasierten Entwurf. Dabei wird der Entwurf Technologie unabhängig nach funktionalen/datenorientierten Kriterien vorangetrieben und resultiert in einem zumeist datenflussorientierten, ausführbaren Modell. Aus diesem Modell kann anschließend die Implementierung für verschiedene Zielplattformen generiert werden.

In der Praxis bestehen zwei wesentliche Einschränkungen. Zum Einen muss bereits früh darauf geachtet werden, welche Zielplattformen in Frage kommen, da sich Einschränkungen in der Modellierung ergeben[3]. Zum Anderen können Performanceanforderungen vor der Implementie-

[3] Für die Implementierung in bestimmten Plattformen (z.B. programmierbare Hardware) ist nur eine einschränkte Menge von Modellelementen geeignet.

rung nur begrenzt berücksichtigt werden, da diese in der Regel von der Zielplattform abhängig sind.

Die Anpassung an verschiedene Plattformen ist im Vergleich zu einem konventionellen Entwurf sehr gering. Oftmals ist nur eine Plattform-spezifische Anpassung der Schnittstellen erforderlich. Andererseits ist die Verteilung eines Modells auf eine komplexe Architektur, also ggf. heterogene Plattformen noch sehr aufwendig, so dass Architekturänderungen zu Redesignzyklen führen, auch wenn diese stark verkürzt sind.

Ein wesentlicher Vorteil des Plattformbasierten Entwurfes ist die frühe Verfügbarkeit von detaillierten Messwerten für die Performance. Diese erhält man durch effiziente Techniken zur Codegenerierung, auch wenn diese jeweils nur Teilmodelle des zu entwickelnden Systems umfassen.

Schnittstellen-getriebene Spezifikation und Verifikation

Die Grundidee der Schnittstellen-getriebenen Spezifikation und Verifikation besteht darin, die Schnittstellen von Komponenten im Bezug auf Statik und Dynamik formal zu beschreiben. Dazu können Eigenschaftsbeschreibungssprachen, wie temporale Logik, Constraint-Beschreibungssprachen, wie Object Constraint Language (OCL), oder Spezifikationssprachen, wie z.B. Z dienen.

Die Spezifikation eines Systems über Schnittstellen bieten in frühen Entwicklungsphasen einige Vorteile. Es wird von der Implementierung einer Komponente abstrahiert, damit wird eine Entkopplung erreicht und die Komponente bleibt austauschbar. Die Interaktion von Komponenten kann anhand der Schnittstellen beschrieben und analysiert werden [FKL04, Bro07].

Für ereignisbasierte Systeme beschreibt eine Schnittstellenspezifikation die Interaktionsmöglichkeiten einer Komponente in Abhängigkeit von ihrem (sichtbaren) Zustand. Die Schnittstellenspezifikation ist Teil der Komponente, zu der die Schnittstelle gehört. Dabei werden unter anderem folgende Modellelemente einer Schnittstellenspezifikation zugeordnet:

- Eingehende und ausgehende Ereignisse,
  - Zeitrelationen
  - Wertebereich
  - (kausale Relationen)
  - (statistische Relationen)
- Genutzte oder bereitgestellte Zustandsvariablen.
- Zeitrelationen
- diskreter Wertebereich

Somit werden einerseits die Klasse von Modellen und andererseits die Klasse von Eigenschaften eingeschränkt, die im Fokus der Betrachtungen stehen.

Bevor die Semantik von Eigenschaften betrachtet wird, soll in diesem Kapitel auf die strukturellen Aspekte von Eigenschaften in der entsprechenden Ausprägung, als Anforderung oder Zusicherung, eingegangen werden [Hüs94, ZJ97]. Anforderungen bzw. Zusicherungen sind Eigenschaften und beziehen sich auf eine Komponente bestimmter Granularität. Strukturell betrachtet existiert ein Anforderer, und einen Zusicherer, wobei sich beides auf eine einzelne (hierarchische) Komponente oder eine Menge von Komponenten beziehen kann. Für die Verifikation wird dem Zusicherer eine Menge von geforderten Anforderungen sowie zugesicherten Zusicherungen zugeordnet. Es existieren folgende Ausprägungen von Schnittstellenspezifikationen, siehe auch Abbildung 2.2:

- Anforderungen der Komponente an die Umwelt
- Anforderungen der Umwelt an die Komponente
- Zusicherungen der Komponente an die Umwelt
- Zusicherungen der Umwelt an die Komponente
- Beschränkungen der Schnittstelle

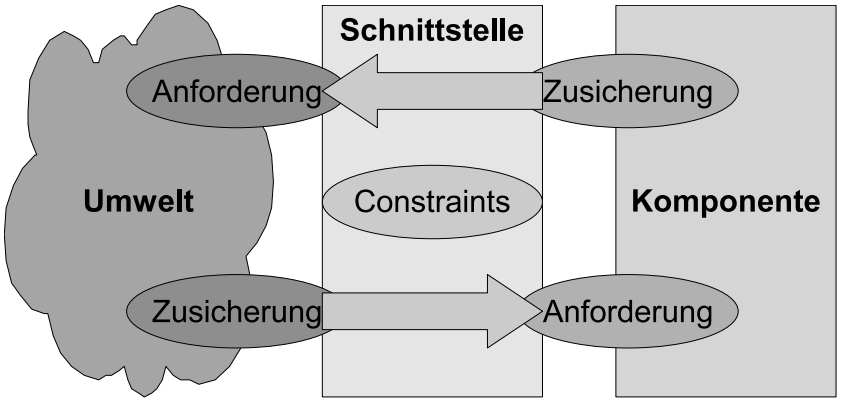


Abbildung 2.2.: Schnittstellen-getriebene Spezifikation von Komponenten

Anforderungen der Komponente beschreiben Voraussetzungen und Beschränkungen der Umwelt, die für den Einsatz einer Komponente erforderlich sind. Anforderungen der Umwelt beschreiben Voraussetzungen und Beschränkungen die für den Einsatz einer Komponente in einer Umwelt erforderlich sind. Zusicherungen der Komponente sind Eigenschaften und Beschränkungen, wie z.B. Reaktivität, die sich bei Einsatz einer Komponente ergeben. Zusicherungen der Umwelt sind Eigenschaften und Beschränkungen, wie z.B. Reaktivität, die für den Einsatz einer Komponente gelten. Beschränkungen einer Schnittstelle stellen Beschränkungen aus der

Sicht der Komponente dar, die sich durch Wechselwirkungen mit anderen Schnittstellen einer Komponente ergeben.

Dieses Modell der Ausprägungen von Schnittstellenspezifikationen behält auch bei hierarchisch verfeinerten - komponierten - Komponenten seine Gültigkeit. Dabei nimmt die Komponente selbst die Rolle einer Umwelt bezüglich seiner Subkomponenten ein. An ein System - als Komposition mehrerer Komponenten - können ebenfalls Anforderungen existieren. Dann nimmt das System die Rolle einer Komponente ein und die Anforderung besteht bezüglich einer gedachten Umwelt.

Die Schnittstellen-getriebene Spezifikation bezieht sich nur auf Zustände von Signalen, bzw. entsprechende Events. Abhängigkeiten zu internen Zuständen einer Komponente sind dadurch in der Regel nicht abgedeckt, da Interna einer Komponente durch die Schnittstelle verborgen werden sollen. Es besteht allerdings die Möglichkeit, extern sichtbare Zustände als Teil der Schnittstelle zu definieren. Dadurch können auch Abhängigkeiten zu internen Zuständen abgebildet werden.

Anforderungen und Zusicherungen der Umwelt ergeben sich erst aus dem Zusammenspiel anderer Komponenten, weshalb sich deren Herleitung als schwierig erweist. Ein vereinfachtes Modell beschreibt daher nur die Anforderungen und Zusicherungen von Komponenten mit je einer - möglicherweise komplexen - Schnittstelle und betrachtet das Zusammenspiel aller Komponenten als Konsistenzproblem. Damit wird die Notwendigkeit vermieden, die Anforderungen der Umwelt aus den die Umwelt bildenden Komponenten ableiten zu müssen.

In dem Kontext einer Komponente (Menge von Komponenten) wird somit behauptet, unter der Voraussetzung der Anforderungen (*claim*) die Zusicherungen (*promise*) erfüllen zu können:

$$claim \rightarrow promise \tag{2.1}$$

Diese Eigenschaft muss anschließend für jede Komponente bewiesen werden. Existieren keine reaktiven Anforderungen, so handelt es sich um invariante Zusicherungen $true \rightarrow promise$, bzw. $promise$. Für den Nachweis der Erfüllbarkeit dieser Eigenschaften lässt sich die Forderung nach der (logischen) Komponierbarkeit von Eigenschaftsspezifikationen ableiten.

2.2.4. Schlussfolgerungen

Die allgemeine Entwicklung eines Systems erfolgt stufenweise in mehreren Phasen, siehe Anschnitt 2.2.1. Gegenstand dieser Arbeit sind die frühen Entwicklungsphasen. Auf die Besonderheiten in Zusammenhang mit dem Entwicklungsprozess sowie auf die Charakterisierung ausführbarer Spezifikationen wird in Abschnitt 2.2.2 eingegangen. Zur Verifikation der Spezifikation ist es erforderlich einen bestimmten Umfang an Informationen über Struktur, Funktion und Performanceparameter bereitzustellen, etwa durch modellbasierten plattformbasierten Entwurf. Zudem werden erhöhte Forderungen an das Management der Zusicherungen und Anforderun-

gen gestellt, wobei eine formale Spezifikation von Komponentenschnittstellen erforderlich ist, siehe Abschnitt 2.2.3.

Im weiteren Verlauf ist es erforderlich mögliche Analysemethoden, Spezifikationssprachen und Verifikationstechniken bezüglich der Eignung zur automatisierten Verifikation von ausführbaren Systemspezifikationen zu analysieren.

2.3. Performancemodellierung

2.3.1. Definition Performance

Vor der Modellierung von Performance muss zunächst eine entsprechende Definition gefunden werden.

Definition Performance: *Eine quantitativ messbare Charakterisierung physikalischer oder funktionaler Attribute betreffend der Ausführung einer Operation oder Funktion* [Was05]

Diese Definition von Performance schließt also eine große Anzahl von Attributen ein, die einer Beschreibung der nichtfunktionalen Eigenschaften dienen. Dazu gehören unter anderem:

- Zeitverhalten; Das Zeitverhalten schließt die Verteilung elementarer Bearbeitungszeiten, Wartezeiten und Verzögerungen ebenso ein wie Durchlaufzeiten und Frequenzen (z.B. Bedienraten).
- Ressourcenauslastung; Die Ressourcenauslastung umfasst die Nutzung beliebiger Ressourcen, etwa zum Speichern von Informationen, zum Berechnen von Daten, etc. Ressourcenbelegungen können dabei in unterschiedlichster Weise betrachtet werden, entweder als exakte Belegung über die Zeit, als Durchschnittswert oder als minimale/maximale Auslastung für quantitative Ressourcen.
- Zuverlässigkeit/Verfügbarkeit; Zur Bewertung der Zuverlässigkeit und Verfügbarkeit werden vor allem statistische Kennzahlen, etwa für die Mean Time Between Failure (MTBF) und Mean Time To Repair (MTTR) herangezogen. Auch andere Kennzahlen für die Beschreibung von Verfügbarkeit bzw. Zuverlässigkeit sind hierbei eingeschlossen.

2.3.2. Performance im Entwicklungsprozess

Performance ist im Entwicklungsprozess in vielfältiger Weise als nichtfunktionale Anforderung zu finden. Die Berücksichtigung dieser Performanceanforderungen zieht sich durch sämtliche Entwicklungsphasen und muss durch geeignete Methoden modelliert und geprüft werden.

Die besondere Schwierigkeit besteht darin, dass die Bewertung der Performance, vor allem bezüglich der Echtzeitfähigkeit, oft nur durch eine gesamtheitliche Betrachtung des zu entwickelnden Systems, also im Zusammenspiel von Funktion und Architektur, bewertet und optimiert werden kann [Tei97].

Für einzelne Anwendungsdomänen ist durch geschickte Wahl der Entwurfsparadigmen eine orthogonale, also wechselwirkungsfreie Betrachtung von Funktion und Performance möglich. Ein Beispiel dafür ist die vollständig zeitgetriggerte Abarbeitung aller Funktionen, so dass die Bearbeitung, unabhängig von den konkret realisierten Funktionen, als globales Schedulingproblem betrachtet werden kann [Jer05]. Ähnlich kann die Ausfallwahrscheinlichkeit für ein System entweder abstrakt, allein in Abhängigkeit von der Architektur, oder detailliert als Folge wahrscheinlichkeitsbedingter Zustandsübergänge betrachtet werden.

Für die weiteren Betrachtungen in dieser Arbeit wird davon ausgegangen, dass eine solch abstrakte Betrachtung nicht möglich ist, also die Dynamik als zeit- und zustandsabhängige Bearbeitung von Funktionen berücksichtigt werden muss.

2.3.3. Aspekte der Modellierung

Die Modellierung eines Systems ist für eine Analyse unerlässlich. Ziel der Modellierung ist ein Modell, das eine sinnvolle Abstraktion der Realität darstellt. Bei der Erstellung des Modells sind einige grundsätzliche Sachverhalte zu berücksichtigen:

- Merkmalsausprägungen/Modellgrenzen: Es muss definiert werden, welche Merkmalsausprägungen im Modell abgebildet werden müssen, bzw. welche Merkmalsausprägungen abstrahiert werden können. Die Abstraktion kann dabei unabhängig vom Detailgrad eines Modells sein, wenn etwa eine Klasse von Umgebungseigenschaften (z.B. Temperaturabhängigkeit) nicht berücksichtigt wird (qualitative Abstraktion). Andererseits ist der Detailgrad eines Modells entscheidend für die Genauigkeit der Ergebnisse, wobei das weglassen von Details ebenfalls zur Abstraktion führt (quantitative Abstraktion).

- Systemgrenzen: Es muss definiert sein, welche Teile zum betrachteten System gehören und welche Teile der Umgebung zugeordnet werden müssen. Die Systemumgebung ist ebenfalls zu modellieren. Je nach Art der Analyse/des Modells besteht die Umgebung aus einfachen Generatoren für die Schnittstellen oder stellt ein komplexes Teilmodell des einbettenden Systems dar.

- Validierung/Verifikation: Modelle bedürfen einer Validierung bzw. Verifikation. Es muss sichergestellt sein, dass das Modell geeignet ist, die gewünschten Merkmalsausprägungen zu beobachten. Zusätzlich muss das Modell korrekt sein. Ergebnisse müssen sich durch Vergleiche mit der Realität belegen oder zumindest plausibilisieren lassen.

- Messbarkeit: Für technische Systeme reicht die Plausibilisierung von Ergebnissen in der Regel nicht aus. Stattdessen ist es erforderlich, einen Vergleich sowohl der Modellparameter als auch der zu analysierenden Eigenschaften des Modells mit dem realen System vorzunehmen. Dies setzt eine Messbarkeit der entsprechenden Größen voraus, wobei die

entsprechenden Regeln der Messtechnik und Stochastik ebenso zu berücksichtigen sind, wie die Arbeitspunkte des Systems.

2.3.4. Realzeitmodellierung

Die Modellierung von Zeiten in Modellen ist in unterschiedlicher Weise Möglich. Die einfachste Variante ist die qualitative Beschreibung als temporale Beziehung. Damit lassen sich jedoch noch keine quantifizierten Aussagen über Zeiten treffen. Eine weitere Möglichkeit besteht in der Verwendung eines ganzzahligen Zeitmodells. Dies erlaubt die Verwendung einer recht einfachen Modellierung. Durch konservative Abstraktion werden dabei effizientere Aussagen ermöglicht [AFH96].

Für die hier betrachteten Aussagen ist diese Vorgehensweise nur bedingt sinnvoll, da als Wertebereich unterschiedliche Zeitskalen möglich sind. Daher ist insbesondere das Modell der *Dense Time* von Bedeutung, das der realen Zeit[4] als kontinuierliches Zeitmodell recht nahe kommt.

Dense Time

Kontinuierliche Zeit kann als sogenannte *Dense Time* wie folgt mathematisch modelliert werden:

Zeit ist eine Menge M von diskreten, unterscheidbaren Zeitpunkten, über der die Ordnungsrelation „$<$“ („liegt zeitlich vor“) folgendermaßen definiert ist:

- Trichotomie:
  $\forall x, y \in M$ gilt genau eine der folgenden Bedingungen: $(x < y), (y < x), (x = y)$.
- Transitivität:
  $\forall x, y, z \in M : (x < y) \wedge (y < z) \Rightarrow (x < z)$
- Irreflexivität:
  $\forall x \in M : \neg(x < x)$
- Dichtheit:
  $\forall x, y \in M : (x < y) \Rightarrow \exists z : (x < z < y)$

Aus Trichotomie, Transitivität und Irreflexivität folgt, dass es sich bei „$<$“ um eine strenge Totalordnung handelt.

[4] Als mathematische Beschreibung des physikalischen (newtonschen) Zeitbegriffs.

2.3.5. Modelle zur algebraischen Analyse

Zur Analyse von Zeit- und Performanceeigenschaften sind einige algebraischen Ansätze entwickelt worden, also Ansätze, die darauf abzielen, quantitative Aussagen über Zeiten durch direkte Berechnungen treffen zu können. Die für diese Arbeit relevanten Ansätze werden in den nachfolgenden Abschnitten betrachtet.

Worst Case Execution Time

Für die Analyse der Worst Case Execution Time (WCET) in diskreten Systemen lassen sich verschiedene Ansätze finden, die sich grundlegend in zwei Kategorien einteilen lassen. Erstens existiert der Ansatz der kompositionalen Analyse, bei der die Berechnung direkt durch die Analyse der Struktur des Systems möglich ist [PZH96, FGM98].

Anders verhält es sich, wenn eine strukturelle Analyse nicht ausreichend ist, etwa weil die Dynamik explizit berücksichtigt werden muss. Dann ist die Analyse von Pfaden innerhalb des Systems erforderlich [CY92, PZH96]. Oder aber es besteht eine Fragestellung, etwa die Frage nach der Bemessung konkreter Zeitintervalle entsprechend gegebener Randbedingungen [Tei97, DHFF04]. In solchen Fällen ist die Beschreibung als Optimierungsproblem eines Ungleichungssystems möglich (*Ganzzahlige Lineare Optimierung*, ILP). Die *Ganzzahlige Lineare Optimierung* wird zur Komplexitätsklasse NP-vollständig gezählt.

Markov-Prozesse

Der Begriff Markov-Prozess[5] bzw. Markov-Kette umfasst eine Klasse von diskreten stochastischen Modellen, bei denen die Zukunft des Modells nur von seinem aktuellen Zustand und nicht von seiner gesamten Vorgeschichte abhängt [Wika].

Das stochastische Modell der zeitdiskreten Markov-Ketten (mit endlich vielen Zuständen) besteht aus Zustandsraum, Anfangsverteilung und Übergangsmatrix [Hey79, GL06, Sch03].

- Ausgangspunkt bildet dabei die (endliche) Menge $E = \{1, 2, \ldots, l\} | l \in \mathbb{N}$ aller möglichen Zustände der Markov-Kette, die den Zustandsraum der Markov-Kette darstellt.
- Für jedes $i \in E$ sei α_i die Wahrscheinlichkeit, dass sich das System zum „Zeitpunkt“[6] $n = 0$ im Zustand i befindet, wobei

$$\alpha_i \in [0, 1], \sum_{i=1}^{l} \alpha_i = 1$$

[5] Benannt nach Андрей Андреевич Марков, je nach Translitaration ergibt sich auch die Bezeichnung *Markow*-Prozess.

[6] Bei der zeitdiskreten Betrachtung wird die Abfolge der Zustände betrachtet, ohne dass die Zeit quantitativ bewertet wird.

vorausgesetzt wird. Der Vektor $\alpha = (\alpha_1, \ldots, \alpha_l)^T$ bildet die Anfangsverteilung der Markov-Kette.

- Für jedes Paar $i, j \in E$ wird die (bedingte) Wahrscheinlichkeit $p_{ij} \in [0, 1]$ des Übergangs von einem Zustand i in einen Zustand j zu einem Zeitschritt betrachtet. Die $l \times l$ Matrix $P = (p_{ij})_{i,j=1,\ldots,l}$ der *Übergangswahrscheinlichkeiten* p_{ij} mit

$$p_{ij} \geq 0, \sum_{j=1}^{l} p_{ij} = 1,$$

heißt (einstufige) *Übergangsmatrix* der Markov-Kette.

Sei $X_0, X_1, \cdots : \Omega \to E$ eine Folge von Zufallsvariablen, die über ein und dem selben Wahrscheinlichkeitsraum $(\Omega, \mathcal{F}, P)$ definiert sind, und ihre Werte in der Menge $E = \{1, 2, \ldots, l\}$ annehmen, dass heißt $X_0, X_1, \ldots$ (homogene) *Markov-Kette* mit der Anfangsverteilung $\alpha = (\alpha_1, \ldots, \alpha_l)^T$ und der Übergangsmatrix $P = (p_{ij})$, falls

$$P(X_0 = i_0, X_1 = i_i, \ldots, X_n = i_n) = \alpha_{i_0} p_{i_0 i_1} \ldots p_{i_{n-1} i_n}$$

für beliebige $n = 0, 1, ...$ und $i_0, i_1, \ldots, i_n \in E$ gilt.

Markov-Ketten ermöglichen die Ermittlung eines stabilen Zustandes für die statistische Verteilung von Prozesszuständen, ausgehend von einer Anfangsverteilung. Voraussetzung für die Anwendbarkeit solcher algebraischen Modelle ist das Vorhandensein einer geschlossenen Lösung. Weiterhin muss die Voraussetzung für die Abbildung des Systems als Markov-Prozess gegeben sein. Erweiterungen der Markov-Ketten sind in der Lage, Zeit quantitativ zu berücksichtigen, indem Wahrscheinlichkeiten für Übergangszeiten berücksichtigt werden. Für Markov-Prozesse wird beispielsweise eine Unabhängigkeit von Einzelentscheidungen (Zustandsunabhängigkeit) vorausgesetzt, also eine konstante Übergangsmatrix, die oftmals nicht gegeben ist.

Neben der algebraischen Analyse wird oft, vor allem für diskrete Modelle, eine numerische Simulation zur Analyse eingesetzt [Buc91]. Markov-Prozesse können dabei auch durch andere Modellierungssprachen, wie beispielsweise stochastische Petri Netze oder Automaten mit stochastischem Übergangsverhalten modelliert werden. Für die Analyse solcher Modelle stehen verschiedene Werkzeuge zur Verfügung, als Beispiel sei hier PRISM erwähnt [RKNP04, KNP07, Oxf].

Duration Calculus

Ein weiterer Ansatz zur Analyse dynamischer Aspekte in dynamischen Systemen besteht in der Verwendung von Logik höherer Ordnung (Higher Order Logic (HOL)). Dabei werden für eine Anwendungsdomäne Theoreme bzw. Inferenzregeln definiert. Liegen Aussagen für ein konkretes Problem als Axiome vor, so wird durch Anwendung der Theoreme (logisches Schließen, Infe-

renz) versucht Aussagen herzuleiten, und somit Aussagen zu beweisen oder zu widerlegen. Die automatisierte Anwendung des logischen Schließens wird allgemein der Klasse entscheidbarer, NP-vollständiger Probleme zugeordnet [Kro99, Wikb].

Daher wird meist eine semi-automatische Herangehensweise gewählt, bei der die Prüfung selbst durch einen Theoremprüfer (engl. theorem proover) erfolgt, die Abfolge der Prüfschritte als Anwendung von Axiomen jedoch durch einen Spezialisten vorgegeben wird [NPW02].

Ein Ansatz zur Verwendung der HOL zur Theoremprüfung von Zeitaspekten in Modellen ist der *Duration Calculus* [CH04]. Bei diesem Ansatz werden Axiome zur Beschreibung von Beziehungen zwischen Ereignissen definiert. Die Semantik der Events und Modellierung von Beziehungen zwischen Events ist dabei eindeutig definiert.

Die Anwendung des Duration Calculus verlangt Spezialwissen zur Modellierung, wobei die Nutzung der Theoremprüfer nicht trivial ist. Da die Nutzung von HOL im Allgemeinen nicht automatisierbar ist, werden Ansätze zur Theoremprüfung in dieser Arbeit nicht weiter verfolgt.

2.3.6. Modell zur Analyse durch Simulation

Eine weitere Möglichkeit der Performanceanalyse von Systemen ist durch Simulation gegeben. Bei der Simulation eines Systems wird das Verhalten eines Systems in Bezug auf eine Menge zuvor definierter Merkmalsausprägungen simuliert, siehe auch Abschnitt 2.3.3.

Die zur algebraischen Analyse verwendeten Modelle, siehe Abschnitt 2.3.5, lassen sich ebenfalls simulieren. Dies ist dann sinnvoll, wenn die Simulation zur Validierung dienen soll, oder aber eine detaillierte Analyse bestimmter Szenarien erforderlich ist (Fehlersuche, Gegenbeweis).

Die im Folgenden dargestellten Betrachtungen zur Simulation betreffen alle werte-diskreten Modelle mit globalem Zeitmodell und wird als Discrete-Event Modell bezeichnet.

Eine Diskussion zur Anwendbarkeit von Simulationen unter bestimmten Voraussetzungen ist unter [BCNN01] und [LK00] zu finden. In den einzelnen Simulationsdurchläufen ist dabei die Erzeugung von Zufallszahlen, z.B. für Wahrscheinlichkeitsverteilungen erforderlich. Die Wiederholbarkeit von Simulationsdurchläufen kann dabei durch eine gezielte Initialisierung der Zufallszahlengeneratoren durch so genannte *Randomseeds* erreicht werden.

Das Simulationsmodell schreibt in der Regel nur eine Halbordnung für die Abarbeitung von Simulationsschritten vor. Für die Abarbeitung der ungeordneten Simulationsschritte müssen geeignete Regeln gefunden werden.

Discrete-Event Simulation

Bei der Discrete-Event Simulation (DE Simulation) können folgende Bestandteile der Modelle unterschieden werden [BCNN01]:

- System: Umfasst die Gesamtheit aller Bestandteile des Modells, die dem zu entwickelnden oder zu analysierenden *System* zugeordnet werden können. Endogener Teil des Modells.

- *Environment* (Umgebung): Umfasst die Teile des Modells, die mit dem System interagieren aber nicht als dessen Bestandteil angesehen werden. Exogener Teil des Systems.
- *Entity* (Entität): Interessierendes Objekt innerhalb des Systems.
- *Activity* (Aktivität): Bezeichnet eine Zeitspanne, die für Veränderungen innerhalb des Systems benötigt wird.
- *Event* (Ereignis): Das Auftreten eines Ereignisses zu einem Zeitpunkt, durch das der Zustand eines Systems verändert werden kann.
- *Attribute* (lokale Zustandsvariable): Eigenschaft, die einer Entität zugeordnet ist.
- *State variable* (globale Zustandsvariable): Eine Zustandsvariable des Systems, die global sichtbar und manipulierbar ist.
- *System state* (Systemzustand): Der Zustand eines Systems umfasst alle Variablen und Attribute, die erforderlich sind, um das System zu einem bestimmten Zeitpunkt zu beschreiben.

Für die Simulation muss zusätzlich eine Laufzeitumgebung vorhanden sein. Dazu werden weitere Elemente benötigt[BCNN01]:

- *Event notice* (Ereignisvermerk): Datensatz eines Ereignisses, das jetzt oder später auftritt. Umfasst sämtliche Daten, die für die Abarbeitung des Ereignisses notwendig sind, mindestens Ereignistyp und Zeitpunkt des Auftretens.
- *Event list* (Ereignisliste): Liste aller Ereignisvermerke für die Zukunft in chronologisch geordneter Reihenfolge. Auch als *future event list* (FEL) bezeichnet.
- *Clock* (Uhr): Eine Variable, die die aktuelle Modellzeit repräsentiert.
- *Scheduler* (Ereignisplan): Der Scheduler bestimmt das bestimmt die Ordnung innerhalb der Ereignisliste und organisiert das Voranschreiten der Modellzeit.

Die Simulation kann in einzelne Simulationsschritte unterteilt werden, wobei die Abarbeitung eines einzelnen Simulationsschrittes als elementar angesehen wird. Die Abarbeitung der einzelnen Simulationsschritte wird durch die Laufzeitumgebung realisiert. Jeder Simulationsschritt wird in vier Stufen abgearbeitet:

1. Entfernen des unmittelbar bevorstehenden Ereignisses aus der Ereignisliste.
2. Voranschreiten der Modellzeit zum bevorstehenden Ereignis.

3. Ausführen des Ereignisses durch eine Aktivität. Aktualisieren des Systemzustandes (lokale und globale Zustandsvariablen) und ändern der Zugehörigkeiten von Entitäten. Generieren nachfolgender Ereignisse.

4. Einordnen der neuen Ereignisse in die Ereignisliste.

Während die Schritte 1, 2 und 4 in der Laufzeitumgebung realisiert werden, wird der Schritt 3 durch das Simulationsmodell realisiert. Dabei ist unerheblich, ob die Ausführung des Schrittes 3 symbolisch, also durch Interpretation eines mathematischen Modells erfolgt oder explizit durch die Implementierung der gewünschten Funktionalität, z.B. mittels C++ Quellcode. Die beiden ersten Schritte sind elementarer Natur und lassen sich relativ einfach realisieren. Schritt 3 wird, je nach verwendeter Simulationsumgebung unterschiedlich realisiert. Der letzte Schritt ist aus theoretischer Sicht besonders interessant, da das Ordnen der Ereignisse innerhalb der Ereignisliste nicht unbedingt eindeutig ist. So wird im Allgemeinen nur eine Halbordnung durch das Simulationsmodell vorgegeben. Diese Halbordnung ergibt sich zum Einen aus der zeitlichen Reihenfolge von Ereignissen, die zu verschiedenen Zeitpunkten auftreten. Zum Anderen können zwischen den Ereignissen, die zum selben Zeitpunkt stattfinden zusätzlich kausale Abhängigkeiten bestehen, beispielsweise wenn generierte Ereignisse zum Zeitpunkt ihrer Entstehung abgearbeitet werden. Für Ereignisse, die zum selben Zeitpunkt stattfinden, jedoch kausal unabhängig voneinander sind, ist die Ausführung in einer beliebigen Reihenfolge möglich. Die Reihenfolge während der Simulation wird durch den Scheduler oft als FCFS oder LCFS realisiert. Dieser Determinismus dient der Reproduzierbarkeit von Simulationen, allerdings können unter Umständen Probleme bei der Wiederverwendung von Teilmodellen auftreten. Ist die Zuweisung interner Zustände abhängig von der Verarbeitung potentiell gleichzeitig auftretender Ereignisse oder Ereignisfolgen, so ist das Modell nicht eindeutig. Von dieser Mehrdeutigkeit sind alle asynchronen Modelle betroffen, zu denen auch die Discrete-Event Modelle gehören.

Eigenschaften von DE-Simulationsmodellen

Discrete-Event (DE) Modelle gehören zur Klasse der asynchronen Modelle. Weiterhin gelten DE-Modelle als zustandsbehaftet. Dies bezieht sich auf die Belegung von lokalen Zustandsvariablen aus funktionaler Sicht (z.B. in Zustandsmaschinen) sowie die Belegung von Ressourcen (Performance Sicht). Die Abarbeitung von Ereignissen gilt als unteilbar. Das heißt, dass eine Bearbeitung beliebig komplex sein darf und alle Zugriffe auf Attribute und Ressourcen beinhaltet. Die für die Beschreibung der Abarbeitung lokal genutzten temporären Variablen gelten dabei nicht als Zustandsvariablen.

Die Komplexität eines Discrete-Event Modells wird durch die Variabilität eines Modells in Zuständen und Zeiten bestimmt. Treten zwischen funktionalen Zuständen und Ressourcen Zuständen keine Wechselwirkungen auf, so ist die Verwendung unabhängiger (orthogonaler)

Modelle möglich. Dies setzt jedoch eine strikte Trennung der betreffenden Modellbestandteile durch die Modellierungsmethodik voraus. Wegen der Wechselwirkungen ist die Nutzung unabhängiger Modelle jedoch oftmals nicht möglich.

2.3.7. Schlussfolgerungen

In diesem Abschnitt wurde der Begriff der Performance allgemein definiert und in Beziehung zum Entwicklungsprozess gesetzt. Dabei wurde festgestellt, dass Performance im Allgemeinen global, also im Zusammenspiel mit Funktion und Architektur eines Systems definiert, siehe Abschnitt 2.3.2. Die Einschränkung auf diskrete Systeme ergibt sich dabei bereits aus der gewählten Domäne der komplexen eingebetteten Echtzeitsysteme, gemäß Kapitel 1. Abschnitt 2.3.3 widmet sich allgemein den zu berücksichtigenden Aspekten der Modellierung. Die quantitative Modellierung von Zeit zur Bewertung der Performance von Echtzeitsystemen wurde in Abschnitt 2.3.4 beschrieben. Dies umfasst vor allem die Modellierung kontinuierlicher Zeit als *Dense Time*. Der Abschnitt 2.3.5 beschäftigt sich mit grundlegenden Methoden zur Analyse von Echtzeitsystemen basierend auf algebraischen Modellen. Dem wird in Abschnitt 2.3.6 mit dem *Discrete-Event Modell* ein allgemeiner Ansatz zur Simulation-basierten Analyse vorgestellt. Die dargestellten Methoden existieren allgemein und beziehen sich auf die Performance in Form quantitativer Echtzeiteigenschaften.

Offen ist demnach die Analyse konkrete Modellierungssprachen, die auch die Beschreibung funktionaler Zusammenhänge ermöglichen, sowie die Analyse konkreter Verifikationstechniken, die eine Analyse temporaler und funktionaler Eigenschaften erlauben.

2.4. Formale Modellierungssprachen

Neben verschiedenen Ansätzen einer rein auf Simulation basierenden Analyse (z.B. MLDesigner, Ptolemy), die eine hohe Ausdrucksmächtigkeit besitzen, existieren auch Ansätze zur statischen Analyse von Eigenschaften. Diese setzen dementsprechend eine syntaktisch und semantisch formale Modellierungssprache zur Beschreibung voraus, erlauben aber meist nur die Abbildung der zu analysierenden Aspekte. Ziel dieses Abschnittes ist die Vermittlung eines Überblicks über ausgewählte Sprachen zur Verhaltensmodellierung, die zur Beschreibung reaktiver Systeme mit Zeitaspekten verwendet werden.

Möchte man Modellierungssprachen vergleichen, so ist eine systematische Untersuchung bezüglich genutzter bzw. unterstützter Konzepte erforderlich. Abbildung 2.3 deutet einen Ansatz zur Systematisierung an. Grundlegend sind hier die *Basiskonzepte* zu nennen, die von Spezifikationssprachen und Verhaltensmodellen unterstützt werden. *Verhaltensmodelle* bieten eine mathematische Beschreibung des funktionalen Verhaltens und dienen der Analyse der beschriebenen Modelle. *Spezifikationssprachen* ermöglichen eine detaillierte Systembeschreibung und

setzen die Basiskonzepte um, gehen aber über die Möglichkeiten einzelner Verhaltensmodelle hinaus.

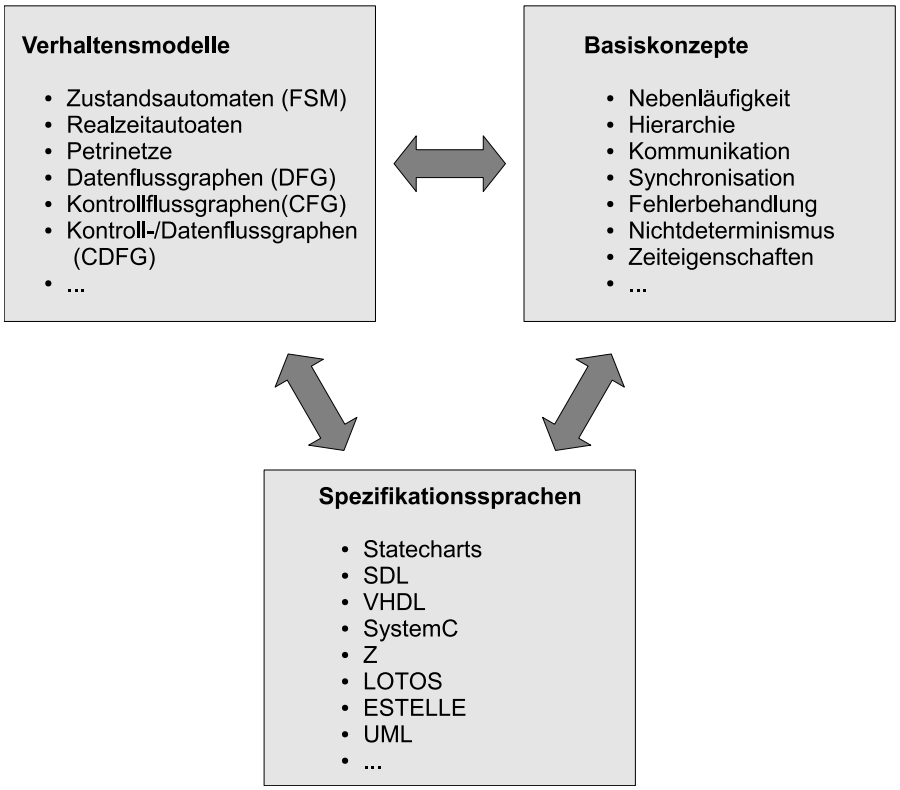


Abbildung 2.3.: Zusammenhang zwischen Verhaltensmodellen, Spezifikationssprachen und Basiskonzepten

Statt Bezüge zu einzelnen Spezifikationssprachen herzustellen, werden daher zunächst die Modellierungssprachen zur Verhaltensmodellierung gemäß der unterstützten Basiskonzepte untersucht und gegenübergestellt.

2.4.1. Synchrone Modelle

Synchrone Modelle müssen Eingaben rückwirkungsfrei zu diskreten Zeitpunkten bearbeiten. Rückwirkungsfrei bedeutet dabei, dass die Reihenfolge der Bearbeitung gleichzeitig auftretender Eingaben die Simulation nicht beeinflusst. Somit können sämtliche Folgezustände von Teilmodellen gleichzeitig berechnet werden. Wird ein streng monotones Zeitmodell verwendet, so dass die Zeit im nachfolgenden Berechnungsschritt echt größer der jeweils aktuellen Zeit sein muss $t_{i+1} > t_i$, so handelt es sich um ein rückwirkungsfreies System. Bei einem schwach monotonen Zeitmodell kann eine *Zero-Delay Rückkopplung* entstehen. Dabei ändert sich ein Prozesszustand ohne Voranschreiten der Zeit durch eine zeitlose Rückkopplung unendlich oft und terminiert nie. Dies bedeutet, dass entweder kein stabiler Zustand erreicht wird, oder aber

mehrere Lösungen existieren [JS05b]. Dazu setzen Hardwarebeschreibungssprachen Microsteps ein, wodurch transiente Zustände durch das so genannte *Delta-Delay* entstehen, denen eine eindeutige Reihenfolge zugeordnet werden kann. Begrenzt man die Anzahl der transienten Berechnungsschritte, so lässt sich ein Oszillieren und somit ein fehlerhaftes Modell, das nicht rückwirkungsfrei ist, erkennen.

Zu den synchronen Modellen gehören unter anderem:

- synchrone Datenflussmodelle (stark monoton)
- Hardwarebeschreibungsmodelle (schwach monoton)
- Automatenmodelle (schwach oder stark monoton)

Synchrone Modelle sind wegen der synchronen Verarbeitung von Eingaben nur beschränkt für die abstrakte Modellbeschreibung geeignet. Somit sind sie nur beschränkt für die Performanceanalyse abstrakter Modelle einsetzbar und werden daher hier nicht weiter betrachtet.

2.4.2. Asynchrone Modelle

Asynchrone Modelle nutzen eine sequentielle Verarbeitung von Eingaben. Bei gleichzeitig auftretenden Eingaben erfolgt die Bearbeitung also sequentiell, was zur Folge hat, dass ein Simulationsergebnis von der Reihenfolge der Verarbeitung von gleichzeitigen Eingaben abhängt. Das Ergebnis von Folgezuständen kann im selben Zeitpunkt, also ohne zeitliche Verzögerung wiederverwendet werden.

Petri-Netze

Petri-Netze [Sta90, Bau90] sind bipartite Graphen, bestehend aus Plätzen $p \in P$, Transitionen $t \in T$ sowie Vorkanten $\bullet A$ ($\bullet A \subseteq P \times T$) und Nachkanten $A\bullet$ ($A\bullet \subseteq T \times P$). Plätze besitzen Kapazitäten $K(p)$ ($Kin\mathbb{N}^+$), Kanten $a \in A = \bullet A \cup A\bullet$ besitzen Vielfachheiten $V(A)$ ($V \in \mathbb{N}^+$). Die Anzahl der Marken $m(p)$ in einem Platz P muss der Ungleichung $0 \leq m(p) \leq K(p)$ genügen. Die aktuelle Belegung der Plätze eines Petri-Netzes durch Marken (aktueller Zustand eines Petri-Netzes) wird als Markierung bezeichnet. Durch die Struktur eines Petri-Netzes (Verknüpfung von Plätzen und Transitionen durch Kanten und deren Vielfachheiten) sowie der Initialmarkierung eines Petri-Netzes werden die erreichbaren Markierungen (Zustände) eines Petri-Netzes bestimmt. Dabei führt das so genannte *Schalten* einer Transition zu einer neuen Markierung. Eine Transition T_i ist schaltfähig, wenn die Vorbedingung $(\forall p_i \in P : m(p_i) \geq \Sigma V((p_i, T_i)))$ und Nachbedingung $(\forall p_i \in P : K(p_i) \geq \Sigma V((T_i, p_i)) - \Sigma V((p_i, T_i)))$ erfüllt ist. Der Zustandsraum eines Petri-Netzes wird durch seinen Erreichbarkeitsgraphen bezüglich der Markierungen beschrieben.

Zeitbewertete Petri-Netze Es existieren verschiedene Erweiterungen für Petri-Netze. Zeitintervall-bewertete Petri-Netze nach Popova-Zeugmann [PZ89, Bau90] besitzen eine zusätzliche Schaltbedingung. Dabei wird jeder Transition ein ganzzahliges Zeitintervall $I, I \in (n_1 \times n_2), n_1 \in \mathcal{N}^+, n_2 \in \mathbb{N}^+ \cup \omega | n_1 \leq n_2$ sowie eine Uhrenvariable $\delta, \delta \in \mathbb{N}^+ \cup \sharp$ zugeordnet. Ist eine Transition nicht schaltfähig, so wird die Uhr gestoppt und mit $\sharp$ markiert. Anderenfalls wird die Uhr beim Eintreten der Schaltfähigkeit mit 0 gestartet und entsprechend dem globalen Zeitmodell inkrementiert. Falls eine Transition schaltfähig ist, so darf sie entsprechend ihres Intervalls I frühestens bei n_1 schalten, spätestens aber zum Zeitpunkt n_2, falls sie nicht vorher die Schaltfähigkeit verliert. Im weiteren Verlauf der Arbeit werden die Zeitintervall-bewerteten Petri-Netze als zeitbewertete Petri-Netze bezeichnet.

Zeitbewertete Petri-Netze erreichen durch das Platz- / Transitionskonzept eine verteilte Beschreibung von Zuständen. Die Zuordnung von Zeitvariablen zu Transitionen erfolgt lokal und beschreibt Abhängigkeiten zu den Vorplätzen. Der Zustand eines Teilnetzes ergibt sich dabei aus der Markierung aller beteiligten Plätze. Zeitvariablen können nur lokal, innerhalb von Transitionen gemessen und bewertet werden. Eine globale Messung und Bewertung von Zeiten wird erst durch zusätzliche Teilnetze ermöglicht. Der Zustandsraum eines zeitbewerteten Petri-Netzes kann durch die Verwendung eines Erreichbarkeitsgraphen beschrieben werden, dessen Markierungen aus Erreichbarkeitsklassen[7] bestehen. Der Erreichbarkeitsgraph eines zeitbewerteten Petri-Netzes ist äquivalent zu einem Regionengraphen.

(Uppaal) Timed Automata

Timed Automata wurden in [AD94] ausführlich definiert und in leicht abgeänderter Form als *Uppaal Timed Automata* [BDL04, BY03] in das Modellierungs- und Analysewerkzeug *Uppaal* [Dep] übernommen.

Timed Automata bestehen aus Zuständen $z \in Z, |Z| \geq 1$ und Transitionen $T \in Z \times Z$. Weiterhin können einem Automaten normale Variablen und Zeitvariablen zugeordnet werden. Normale Variablen können algebraisch manipuliert werden. Zeitvariablen werden mit 0 initialisiert und können beim Schalten einer Transition auf einen bestimmten Wert gesetzt werden, ansonsten steigt ihr Wert monoton. Jeder Automat besitzt einen Initialzustand $z_0 \in Z$. Jeder Zustand kann Beschränkungen bezüglich (Zeit-)Variablen enthalten. Ist die Beschränkung, etwa durch das Voranschreiten von Zeit, nicht erfüllt, so muss entweder eine ausgehende Transition schalten oder der Automat befindet sich in einem verbotenen Zustand (Deadlock).

Automaten können sich synchronisieren, indem sie Events E empfangen oder versenden. Dazu existieren verschiedene Kommunikationssemantiken ($\{$*unicast*, *multicast*$\} \times \{$*zwingend*, *optional*$\}$). Eine Transition besitzt zweierlei Voraussetzungen für das Schalten: Synchronisati-

[7] Erreichbarkeitsklassen sind Tupel, bestehend aus Markierungsvektor der Plätze und Vektor der Intervalle von Uhrenvariablen der Transitionen.

onsbedingung und Zustandsbedingung. Ist eine Synchronisationsbedingung notiert, so erfolgt das Schalten der beteiligten Transitionen synchron, also beim Senden oder Empfangen eines Events. Ist eine Zustandsbedingung in Form von Beschränkungen von Variablen notiert, so müssen diese Beschränkungen für das Schalten erfüllt sein. Zusätzlich kann eine Transition Zuweisungen enthalten, bei denen (Zeit-) Variablen konstante oder berechnete Werte zugeordnet werden. Durch die Beschränkungen von Variablen kann ggf. ein endlicher Zustandsraum in Form eines Regionengraphen verwendet werden.

Prozessalgebra Timed CSP

Prozessalgebren umfassen eine Klasse von Ansätzen zur formalen Modellierung von nebenläufigen Systemen. Sie beschreiben abstrakte Prozesse, deren Interaktions- und Kommunikationsmöglichkeiten vom internen Zustand abhängen. Prozessalgebren besitzen spezielle Operatoren zur Beschreibung der strukturellen Komposition von Prozessen.

Communicating Sequential Processes (CSP) [Hoa78] unterstützt eine Vielzahl von Interaktionsmustern, wie deterministische und nichtdeterministische Auswahl, Nebenläufigkeit, Parallelkomposition, nebenläufige Komposition, Bedingungen, Timeout und Interrupt. Mit *Timed CSP* [OS06] steht eine Erweiterung von CSP zur Verfügung, die zusätzlich eine Spezifikation von quantitativen Zeiten erlaubt.

Die Prüfung von CSP kann entweder regelbasiert erfolgen, oder aber durch Transformation in Automaten bzw. Transitionssystemen. Die Verifikation von Timed CSP kann durch Transformation in Realzeitautomaten realisiert werden, da beide Formalismen die gleiche Ausdrucksmächtigkeit besitzen [Oua02]. Somit scheinen Realzeitautomaten geeigneter für die Verifikation zu sein.

2.4.3. Zeitbewertete Prozessmodelle

Ziel zeitbehafteter Prozesse ist die Beschreibung kausaler Abhängigkeiten. Diese Beschreibung wird oftmals genutzt, um eine Zuordnung von Prozessen zu Ressourcen vorzunehmen, zu analysieren und ggf. zu optimieren. Dementsprechend werden zeitbehaftete Prozessmodelle u.a. für die Schedulinganalyse verwendet. Zeitbehaftete Prozessmodelle bieten somit in gewisser Weise einen orthogonalen Ansatz zur zustandsbasierten Verhaltensmodellierung. Hier steht nicht die Modellierung der Zustandsabhängigkeiten, sondern die Beschreibung der kausalen Abhängigkeiten über Prozesse und Ressourcen hinweg im Vordergrund, während das zustandsabhängige Verhalten, im Vergleich zu Prozessalgebren wie Timed CSP, nicht modellierbar ist.

Der Unterschied lässt sich wie folgt beschreiben. Während kausale Abhängigkeiten die Abfolge von Prozessaktivierungen beschreiben, beschreibt eine Zustandsabhängigkeit die Art und Weise einer Bearbeitung oder die konkreten Datenabhängigkeiten. Ein rein kausales Modell beschreibt die Struktur einer Prozesskette mit Entscheidungspunkten und optionalen Prozessen

in Abhängigkeit von einem externen Auslöser (Ereignis). Die Verbindung zwischen Prozessen wird durch Ereignisse realisiert.

Ein Echtzeitsystem muss innerhalb einer bestimmten Zeit auf Ereignisse reagieren. Diese Zeitspanne kann, wie in Abbildung 2.4 dargestellt, als Intervall für die Ausführung eines Prozesses verallgemeinert werden, wobei der Prozess für die Reaktion auf das Ereignis zuständig ist.

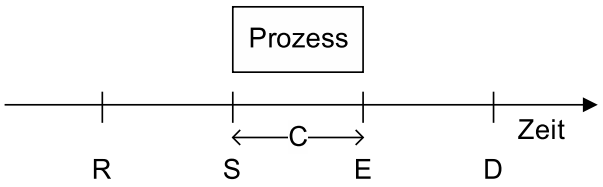


Abbildung 2.4.: Prozesszeiten nach [Hüs94]

Die Zeiten haben folgende Bedeutung:

- R: frühester Startzeitpunkt (release time)
- D: spätester Endzeitpunkt (deadline)
- S: tatsächlicher Startzeitpunkt (start time)
- E: tatsächlicher Endzeitpunkt (completetion time)
- C: Laufzeit (computation time)

Dabei gilt $R \leq S < E < D$ und somit wegen $S < E$ auch $C > 0$. Das Systemverhalten wird dabei, entsprechend der Sicht der Systemspezifikation, von außen betrachtet. Für die Spezifikation sind also die Parameter R und D ausschlaggebend, wobei sich konkrete Werte (Intervalle) für S und E aus dem Spezifikationsmodell bzw. dem Entwurfsmodell ergeben können, so dass die Bedingung $R \leq S < E < D$ automatisch prüfbar wird. Das Modell hat immer einen genauen Zeitpunkt, wann die Simulation (oder der Betrieb) beginnt. Deswegen hat die Zeitskala einen festen Anfang, der mit dem Symbol 0 bezeichnet ist. Das Systemverhalten ist im Allgemeinen von den aktuellen Eingangsdaten und Parametern abhängig, weshalb die genaue Systemfunktion zum Zeitpunkt des Simulationsstarts nicht eindeutig bekannt ist. Daher wird ein Zeitmodell benötigt, das mit Klassen von Zeitpunkten umgehen kann, also intervallbasiert ist. Um für einige elementare Modellelemente (Primitivelemente) die Ausführungszeit exakt definieren zu können, muss das Zeitmodell Zeitintervalle der Breite Null unterstützen.[8] Für Zeitintervalle der Breite Null muss daher für die Zeitpunkte R, S, E und D des Prozessmodells aus Abbildung 2.4 $R \leq S \leq E \leq D$ gelten.

Ein zustandsbasiertes Modell beschreibt die Wechselwirkungen zwischen Zuständen, Ereignissen und Zeitbedingungen. Dadurch ist nicht mehr entscheidbar, ob eine Zeitbedingung Auslöser

[8] In Zeitmodellen, beispielsweise in Realzeitlogiken werden Zeitschritte der Breite Null definiert, so das Modellhaft mehrere Systemzustände ohne Zeitverlust durchschritten werden können. Bedeutung hat dies u.a. für die Synchronisation oder die Verifikation mittels Fehlerzuständen.

einer selbständigen Ereigniskette ist, oder die Verbindung zu einem bestimmten vorangegangenen Ereignis darstellt. Das Problem kann ggf. dadurch gelöst werden, dass Ressourcen als besonderes Modellelement betrachtet werden. Über die durch Ressourcenbelegungen kann eine Verbindung zwischen belegendem Ereignis, Zeit und freigebendem Ereignis hergestellt werden, die als kausale Abhängigkeit interpretiert wird.

2.4.4. Verhaltensmodellierung mit der UML

Die UMLstellt eine breite Palette von grafischen Notationsmöglichkeiten in Form verschiedener Diagrammtypen bereit. Die verschiedenen Diagrammtypen sind jeweils für die Darstellung bestimmter struktureller oder funktionaler Sachverhalte angepasst. Wesentlich ist dabei, dass alle Diagrammtypen auf dem selben Metamodell basieren, wodurch die Wechselwirkungen der einzelnen Diagramme formalisiert werden können. Weiterhin besteht die Möglichkeit eigene Anpassungen (Profile) unter Verwendung des UML Metamodells (Meta Object Facility, MOF) zu definieren.

Abbildung 2.5 zeigt eine Übersicht der Diagrammtypen der UML 2.0, unterschieden in Strukturdiagramme, Verhaltensdiagramme und Interaktionsdiagramme als Teil der Verhaltensdiagramme. Die Verwendung der Diagramme innerhalb eines Modells ist bewusst offen gehalten. Leider ist dadurch das systematische Ineinandergreifen verschiedener Verhaltensdiagramme nicht durchgängig realisiert, wodurch die Ausführbarkeit der Modelle nicht gewährleistet werden kann. Nur wenige Werkzeuge bieten hierzu eine geeignete Methodik.

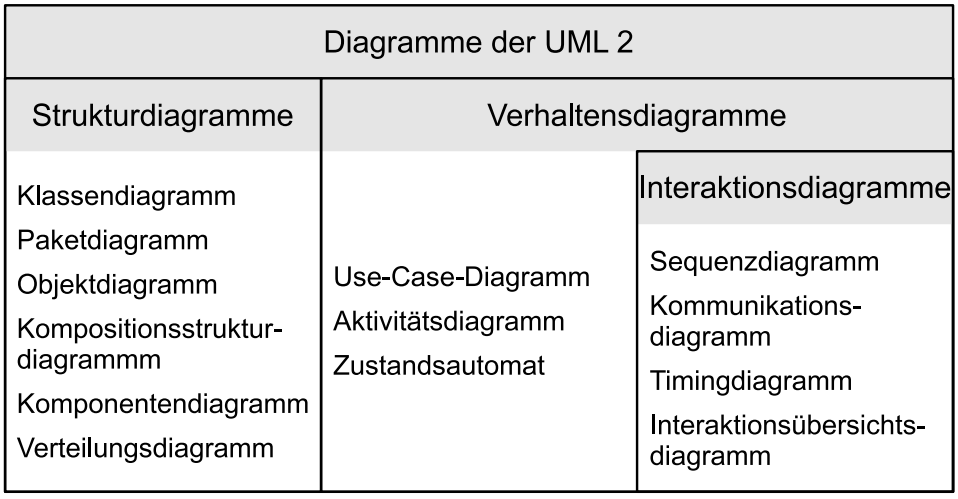


Abbildung 2.5.: Diagramme der UML 2.0 nach [RHQZ05]

Wesentliche Kritikpunkte des Systems Engineering an die UML bestehen in der geringen Unterstützung des Anforderungsmanagements, in der begrenzten Wiederverwendbarkeit in späteren Entwicklungsphasen (Codegenerierung) und der fehlenden Eineindeutigkeit einiger Ver-

haltensdiagramme [Wei06]. Ein wesentlicher Aspekt wird durch Methoden der Model Driven Architecture (MDA)adressiert, deren Ziel die automatische Codegenerierung von Systemmodellen für verschiedene Plattformen darstellt. Maßgeblich dabei ist die Trennung fachlicher und technischer Aspekte. Das dazu verwendete Systemmodell muss domänenspezifisch sein, also die fachliche Aspekte der betreffenden Domäne unterstützen, und andererseits formal eineindeutig und vollständig sein, um die Codegenerierung zu ermöglichen[RHQZ05]

2.5. Formale Eigenschaftsspezifikation

Techniken der funktionalen Verifikation erlauben eine Überprüfung, ob ein System eine Funktion, entsprechend einer Spezifikation, korrekt ausgeführt wird [Kro99, CGP99]. Dementsprechend sind zwei Voraussetzungen für eine Verifikation zu erfüllen. Es muss eine formale Spezifikation der zu verifizierenden Eigenschaft gegeben sein. Zusätzlich muss das zu überprüfende System in geeigneter Form vorliegen. Mathematisch gesehen wird die Implikationsbeziehung zwischen Implikation (IMPL) und Spezifikation (SPEC),

$$IMPL \rightarrow SPEC \tag{2.2}$$

nachgewiesen.

Formale Eigenschaftsspezifikationen beschreiben Eigenschaften in einer Weise, die mathematisch eineindeutig interpretiert werden kann. Somit kann entschieden werden, ob eine Eigenschaft gilt, oder nicht[9]. Je nach Verifikationsverfahren kann die formale Eigenschaftsspezifikation in unterschiedlicher Form vorliegen. Für Verfahren der Äquivalenzprüfung können können Eigenschaften und Modell als formales funktionales Modell (Automaten) vorliegen [Kro99]. Andere Verifikationsverfahren, etwa Model Checking, nutzen abstrakte Spezifikationen, basierend auf temporaler Logik, während das zu prüfende Modell allgemein in Form eines Automaten vorliegt.

2.5.1. Temporale Logiken

Die Verwendung weniger Bit-basierter Datentypen in einfacher oder vektorisierter Form begünstigte diese Entwicklung. Da die Beschreibung von digitaler Hardware ausschließlich im Zeitbereich stattfindet und Zusammenhänge kombinatorisch bzw. zustandsbasiert modelliert werden, wurden temporale Spezifikationssprachen entwickelt. Diese so genannten Temporalen Logiken beschreiben Pfade (Traces), also zeitlich-logische Abfolgen von Systemzuständen und-/oder Signalzuständen, ausgehend von einem aktuellen Systemzustand. Dabei werden zu den booleschen Operatoren so genannte temporale Operatoren definiert, die die Erfüllung einer

[9] Sofern aufgrund der Komplexität der Nachweis praktisch realisierbar ist.

Eigenschaft in Abhängigkeit von der Zeit beschreibt. Die Relation $\vDash$ in der Formel $s \vDash \varphi$ sagt aus, dass im betrachteten Zustand s die Eigenschaft φ erfüllt wird. Eine Unterscheidung zwischen Zustands- und Pfadformeln erfolgt hier nicht. Die grundlegenden temporalen Operatoren X (ne$X$t), F (Finally/eventually), G (Generally) und U (Until) werden als wie folgt definiert [Kro99]:

$$
\begin{aligned}
s \vDash X\varphi \quad &\leftrightarrow \quad \text{im nächsten Zeitschritt ist } \varphi \text{ erfüllt, } s^1 \vDash \varphi \\
s \vDash F\varphi \quad &\leftrightarrow \quad \text{es existiert ein } k \geq 0, \text{ so dass } s^k \vDash \varphi \\
s \vDash G\varphi \quad &\leftrightarrow \quad \text{für alle } k \geq 0 \text{ gilt } s^k \vDash \varphi \\
s \vDash \varphi_1 U \varphi_2 \quad &\leftrightarrow \quad \text{es existiert ein } k \geq 0, \text{ so dass } s^k \vDash \varphi_2 \text{ und} \\
&\qquad\quad \text{für alle } 0 \leq j < k \text{ gilt } s^l \vDash \varphi_1
\end{aligned}
$$

Zudem gibt es weitere Operatoren, die sich jedoch aus den bestehenden ableiten lassen, etwa wU (weak Until), W (When), B (Before) und V (release). Ersetzungen der Operatoren sind in [Kro99] ausführlich beschrieben. Die temporalen Operatoren bilden die Grundlage für die temporalen Logiken mit linearer Zeit, insbesondere Linear Temporal Logic (LTL). Für die Unterscheidung verschiedener Verzweigungen existieren bei verzweigenden temporalen Logiken, wie z.B. bei der Computation Tree Logic (CTL), so genannte Pfadquantoren, A (alle Pfade) und E (Pfad existiert), die jedem temporalen Operator vorangestellt werden müssen.

Es existiert eine Vielzahl temporaler Logiken, von denen nur die im Kontext der Arbeit relevanten betrachtet werden. Temporale Logiken lassen sich nach verschiedenen Kriterien unterscheiden [Kro99], siehe Tabelle 2.1. Auf lineare und verzweigende Zeit (LTL und CTL) wurde

Verzweigung	Kontinuität	Zeitmessung	Zeitrichtung
lineare Zeit	Zeitinstanzen	diskrete Zeit	inklusive Vergangenheit
verzweigende Zeit	Zeitintervalle	kontinuierliche Zeit	nur Zukunft

Tabelle 2.1.: Klassifikationsmöglichkeiten für Temporale Logiken nach [Kro99]

in diesem Abschnitt bereits hingewiesen. Bei linearer Zeit werden unterschiedliche alternative Ausführungspfade identisch behandelt, während bei verzweigender Zeit spezifiziert wird, ob Eigenschaften in einem oder allen Ausführungspfaden erfüllt sein müssen. Zustände können für Zeitpunkte oder für Zeitintervalle beschrieben werden. Zeiten können weiterhin entweder rein diskret erfasst werden, so dass natürliche Zahlen $\mathbb{N}$ zur Markierung der Zeit benutzt werden, oder es wird ein kontinuierliches Zeitmodell mit gebrochen rationalen Zahlen verwendet. Die Begriffe *diskrete Zeit* und *kontinuierliche Zeit* führen unter Umständen zu Missverständnissen. Daher werden in dieser Arbeit die Begriffe *abgezählte Zeit* und *diskretisierte kontinuierliche Zeit* verwendet. Damit wird dem Umstand Rechnung getragen, dass kontinuierliche Zeitverläufe aus Gründen der Berechenbarkeit nicht komplett beschrieben werden können, und daher diskretisierte Zeitpunkte bzw. Intervalle in quasikontinuierlicher Zeit referenziert werden, da der Mechanismus zur Quantifizierung (quasi-)kontinuierlich arbeitet. Des Weiteren wird unterschie-

den, ob temporale Operatoren in die Vergangenheit oder in die Zukunft referenzieren. Dafür existieren temporale Operatoren, deren Bedeutung am Zeitpunkt 0 gespiegelt ist. Dies stellt keine direkte Einschränkung dar, da eine äquivalente Ausdrucksmächtigkeit besteht [MP91].

Für die Prüfung der Implementierung digitaler Hardware wird meist ein lineares oder verzweigendes Zeitmodell mit Zeitinstanzen mit einem zukunftsorientierten abgezähltem Zeitmodell verwendet[Kro99].

Eine besondere Bedeutung fällt daher den temporalen Spezifikationssprachen LTL und CTL sowie CTL* (CTL*) zu [Kro99]. LTL ist in der Lage lineare Bearbeitungspfade eines Systems durch temporale Operatoren zu beschreiben. CTLbenötigt zusätzlich einen Pfadquantor und ist in der Lage zu unterschieden, ob es einen Bearbeitungspfad für eine temporale Eigenschaft gibt, oder eine Eigenschaft für alle möglichen Bearbeitungspfade gelten muss. Eine formale Definition temporaler Strukturen sowie der Spezifikationssprachen LTL, CTL und CTL* ist in [Kro99] zu finden.

Für Bounded Model Checking und Assertion-Checking werden die temporalen Operatoren um Zeitintervalle erweitert, um die Wirkung der Operatoren und somit die Länge der zu betrachtenden Pfade zeitlich zu beschränken. Ein entsprechender Spezifikations- und Verifikationsansatz wurde u.a. in [RHKR00] vorgestellt. Für digitale Hardware werden dabei ganzzahlige Intervalle genutzt, welche die Zeit als Abzählung von Taktzyklen beschreiben.

Beschreibungssprachen, die diskretisierte kontinuierliche Zeit als Beschränkungen für temporalen Operationen zulassen, werden als Realzeitlogiken bezeichnet [JH08].

Für Beschreibungen, die sich auf Signale beziehen, besteht im Allgemeinen eine eineindeutige Abbildung zwischen Zeitpunkt und Signalzustand, egal ob Zeit abgezählt oder diskretisiert kontinuierlich modelliert wird. Für abstrakte Modelle ist dagegen oft ein zweidimensionales Zeitmodell mit so genannten Deltazyklen vorzuziehen. Dabei werden neben den stabilen Zuständen zusätzlich transiente Zustände modelliert, die jedoch nicht real sichtbar sind [MNP08].

Realzeitlogiken

Als Realzeitlogiken werden temporale Logiken bezeichnet, die quantitative Aussagen über Zeitverläufe innerhalb der betrachteten Pfade erlauben. Allgemein werden dazu die temporalen Operatoren mit Zeitangaben versehen (z.B. $F_{[1,5]}\varphi$), wie etwa bei der *finite linear temporal logic* (FLTL) [RHKR00] , der *metric interval temporal logik* (MITL) [AFH96] und der *event clock temporal logic* (ECL) [RS96]. Zudem besteht die Möglichkeit, Beschränkungen von Zeitvariablen des zu prüfenden Modells zu spezifizieren (z.B. $AG(x < 3)$).

Zu den verzweigenden Realzeitlogiken (Klasse CTL) gehören unter anderem *clocked computation tree logic* (CCTL) [RK00] und *timed computation tree logic* (TCTL) [ACD93]. CCTL nutzt ein ganzahliges Zeitmodell. Bei TCTL können hingegen auch reellwertige Zeitbeschränkungen genutzt werden.

2.5.2. Eigenschaftstemplates mit temporaler Logik

Es gibt verschiedene Ansätze Eigenschaften in temporaler Logik durch Templates zu spezifizieren. Dabei wird die Komplexität einer Eigenschaft beschränkt, um die Ausdrucksmächtigkeit auf eine sinnvolle, vom Anwender fassbare Auswahl zu beschränken [MP91, Hol03].

Tabelle 2.2 beschreibt die wichtigsten Templates für temporale Logiken. In der ersten Spalte ist die (verallgemeinerte) LTL-Formel für die Beschreibung der Eigenschaft notiert, G stellt den $Generally-$Operator und F den $Finally-$Operator dar. In der zweiten Spalte ist die englische Bezeichnung notiert. Die dritte Spalte notiert die Klasse der Eigenschaft, wobei *Progress* (Verlauf) und *Safety* (Sicherheit) bezüglich des Verhaltens unterschieden werden. In der letzten Spalte sind Erläuterungen zur Lesart der Eigenschaft notiert.

Formel	Bezeichnung	Klasse	Erläuterung
$\bigwedge_{i=1}^{n} [GFp_i \vee FGq_i]$	Reactivity	Progress	Verallgemeinert, Antwort oder Persistenz.
GFp	Response	Progress	Antwort, immer irgendwann p.
FGp	Persistence	Progress	Stabilität, Irgendwann gilt immer p.
$\bigwedge_{i=1}^{n} [Gp_i \vee Fq_i]$	Obligation	Progress	Bindung, immer p oder irgendwann q
Fp	Guarantee	Progress	p gilt irgendwann.
Gp	Safety	Safety	p gilt immer.

Tabelle 2.2.: Templates für Temporale Logiken nach [Hol03]

2.5.3. Spezifikationssprache PSL

Die Sprache Property Specification Language (PSL)[Acc04, FKL04] zur Eigenschaftsspezifikation ist für synchrone digitale Systeme entworfen worden. Sie unterstützt lineare temporale Eigenschaften, sowie Erweiterungen der CTL, die so genannten *Optional Branching Extensions*. Hinzu kommen abkürzende Schreibweisen, wie z.B. die Sequential Extended Regular Expressions (SERE), die eine vereinfachte Spezifikation sequentieller Abläufe erlauben. Die Sprache ist in verschiedene semantische Ebenen, aufgeteilt:

- Modeling Layer
- Boolean Layer
- Temporal Layer
- Verification Layer

Der *Modeling Layer* dient zur Interpretation von Signalen, die aus der Verifikationsumgebung, also beispielsweise einem HDL-Simulator, stammen. Dabei kann es sich um Signale, Signalgruppen oder Events handeln, die im Sinne der Eigenschaftsspezifikation als ein endliches oder unendliches Wort über boolesche Werte $\mathbb{B}^n, n \in \mathbb{N}$ interpretiert werden. Für die Verifikation

selbst werden nur boolesche Werte bzw. endliche/unendliche Worte über boolesche Werte verwendet, die im Folgenden als boolesche Signale bezeichnet werden. Mit dem *Modeling Layer* wird die Verbindung zum Modell hergestellt, während Boolean Layer und Temporal Layer zur eigentlichen Eigenschaftsspezifikation verwendet werden.

Der *Boolean Layer* dient zur Bildung boolescher Ausdrücke basierend auf booleschen Signalen. Das Ergebnis eines Ausdrucks ist ein boolesches Signal. Rein boolesche Ausdrücke besitzen keinen Zeitbezug.

Der *Temporal Layer* dient dazu, den Verlauf boolescher Signale über die Zeit zu beschreiben, also z.B. wann ein boolescher Ausdruck gelten muss. Das Ergebnis ist wiederum ein boolesches Signal, wobei die Bewertung eines boolesches Signals ggf. verzögert vorliegt.

Im *Verification Layer* wird die Verwendung der Eigenschaftsspezifikationen gesteuert. Während einer Simulation können Eigenschaften auf verschiedene Weise verwendet werden. Neben der eigentlichen Prüfung der Eigenschaft, kann während einer Simulation die Erzeugung von Signalen zufällig so gesteuert werden, dass die resultierende Abfolge konform zu den Eigenschaften sind, die z.B. ein Kommunikationsprotokoll oder Datenpakete beschreiben. Damit wird sichergestellt, dass zwar zufällige, aber dennoch gültige Daten verwendet werden.

Für die eigentliche Verifikation werden Eigenschaften als *Assertions* genutzt. Sie müssen also erfüllt werden, ansonsten liegt ein Fehler vor, was für Log Nachrichten, zur Erstellung von Fehler Traces, oder zur Steuerung der Simulation verwendet werden kann.

Für die Simulation unvollständiger Spezifikationen können *Constraints* benutzt werden. Dabei wird die Belegung unbeschalteter Eingänge zufällig mit Werten belegt, so dass die Constraints erhalten bleiben. Dadurch können beispielsweise Teilnehmer eines Kommunikationsprotokolls simuliert werden, indem das Kommunikationsprotokoll mittels Constraints beschrieben wird.

PSL erlaubt die Spezifikation formaler temporaler Eigenschaften für getaktete digitale Systeme und enthält Erweiterungen, die einerseits die Anbindung an verschiedene Modelle/ Simulatoren erlauben und andererseits die Verwendung von Eigenschaften steuern. Es existieren erste Erweiterungen zur Beschreibung von Analogen Systemen. Derzeit wenig verbreitet ist die Unterstützung von Multi-Clock Systemen.

2.5.4. Spezifikationssprache STL

Signal Temporal Logic (STL) ist für die Beschreibung von Eigenschaften analoger Signale mit einem Zeit- und Werte-kontinuierlichem Signalmodell als Erweiterung von PSL entwickelt worden [MNPB06, MNP08, NMP+06]. Dazu wurden im EU-geförderten Projekt PROSYD [PRO] verschiedene Verfahren zur Eigenschaftsbeschreibung und -Generierung analysiert, bewertet und erweitert. Dieser Ansatz besteht aus zwei Teilen. Zum einem wird ein Ansatz zur Beschreibung von Analogsignalen als Menge von booleschen Interpretationen über eine geschlossene

Menge von Intervallen. Die Interpretation wird dabei über Abstraktionsfunktionen realisiert, die eine Abbildung $\mathbb{R}^n \rightarrow \mathbb{B}$ darstellen. Die geschlossene Menge von Intervallen wird als *Interval-Covering* bezeichnet und enthält ausschließlich links geschlossene rechts offene Intervalle. Die enthaltenen Intervalle überlappen sich nicht und lassen sich paarweise aneinander fügen, so dass eine geschlossene Definitionsmenge $\mathbb{T} = [0, t); t > 0 \in \mathbb{R}$ abgebildet wird.

Die Prüfung erfolgt in zwei Schritten. Zum einen werden Signale zur Prüfung in eine Menge endlicher, für die zu prüfende Eigenschaft optimales Intervall-Covering zerlegt (Filter). Anschließend werden für jedes ermittelte Covering die möglichen, ggf. zeitlosen, Abfolgen als Monitore realisiert (Prüfer). Die Modellierung von Filter und Prüfer können dabei mittels angepasster Timed Automata erfolgen.

2.6. Verifikationstechniken

Als *Formale Verifikationsansätze* werden die Ansätze betrachtet, die eine mathematisch vollständige Prüfung von Modellen erlauben. Aus Betrachtungen der theoretischen Informatik, speziell der Automatentheorie, folgt, dass eine Formale Verifikation nicht für beliebige Kombinationen aus Modellen und Eigenschaften praktisch realisiert werden kann, da im Allgemeinen nicht genügend Zeit und/oder Speicher zur Verfügung steht. Trotz dieser Beschränkungen hat sich gezeigt, dass der Einsatz von Formalen Verifikationsansätzen auch für komplexe Systeme praktikabel sein kann.

In den folgenden Absätzen werden verschiedene Ansätze zur formalen Eigenschaftsprüfung erläutert bzw. gegenübergestellt.

2.6.1. Zustandsbeschreibung als Regionengraph

Die Techniken des *Model Checking* nutzen Transitionsmodelle, um die temporalen Abhängigkeiten erreichbarer Zustände zu Modellieren. Ein Transitionsmodell kann dabei als gerichteter Graph repräsentiert werden, dessen Ecken Zustände darstellen, während die gerichteten Kanten die Transitionsrelation repräsentieren. Bei der Verifikation durch Model Checking wird dann, anhand der möglichen Abfolgen von Zuständen, die sich aus der Struktur des Graphen ergeben, die Erfüllbarkeit von Eigenschaften entschieden.

Die Zusammensetzung eines Zustandes hängt dabei von dem zugrunde liegenden Modell ab, wobei die konkrete Darstellung zusätzlich von der Verifikationstechnik beeinflusst wird.

Für zeitbehaftete Systeme werden lokale *Zeitvariablen* C (Clocks) benutzt, die ein relatives Zeitmodell beschreiben[10]. Dabei ergeben sich die möglichen Zustände S des Transitionssystems aus dem Produkt der diskreten Zustände Z des Systems und der Belegungen der Zeitvariablen

[10] Bei einem absoluten Zeitmodell wäre für Systeme mit unendlicher Laufzeit keine geschlossene Darstellung möglich.

C, so dass je nach verwendetem Zeitmodell $S \subseteq Z \times \mathbb{N}^C$ oder $S \subseteq Z \times \mathbb{R}^C$ gilt. Um eine kompaktere Darstellung zu erreichen, werden die Zeiten nicht einzeln betrachtet, sondern je Zeitvariable so zu Intervallen zusammengefasst, dass resultierende Menge S paarweise disjunkt ist. Dabei wird ausgenutzt, dass die Zeit für alle Zeitvariablen gleichermaßen voranschreitet. Die Darstellung der Zerlegung des $|C|$-dimensionalen Raumes der Zeitvariablen in Intervalle der Zeitvariablen wird auch als *Regionengraph* (Clock Zones) bezeichnet. Ist die Menge der (aktiven) Zeitvariablen konstant, so können Difference-Bound Matrices (DBM) für die Darstellung der Regionen verwendet werden [Dil90].

2.6.2. Explizites Model Checking

Beim expliziten Model Checking wird der Erreichbarkeitsgraph eines Modells explizit erzeugt. Für Systeme die aus Automaten bestehen ist der Erreichbarkeitsgraph zwar endlich, aber mitunter zu groß für eine explizite Darstellung. Viele Optimierungen bestehen daher bezüglich der Codierung von Zuständen oder Zustandsmengen. Oftmals wird vor der Generierung des Erreichbarkeitsgraphen die Eigenschaft mit dem Modell verschränkt, also ein gemeinsamer Erreichbarkeitsgraph erzeugt. Dessen Generierung kann dann vorzeitig abgebrochen werden, wenn ein Zustand erreicht wird, der die zu prüfende Eigenschaft verletzt. Für LTL-Eigenschaften wird ein ω-Automat erzeugt, der die Negation einer Eigenschaft repräsentiert. Ist die Eigenschaft verletzt, so existiert eine Cosimulation mit dem *omega*-Automaten, anderenfalls ist der Erreichbarkeitsgraph leer. Ist der Erreichbarkeitsgraph nicht leer, so stellt eine Spur (Trace) im Erreichbarkeitsgraphen einen Gegenbeweis (Zeuge/Witness) dar.

Wegen dem oft erheblichen Verbrauch an Speicherplatz wird explizites Model Checking vorwiegend für zeitbehaftete Modelle (Region Graph) oder für asynchrone Modelle, z.B. Software, verwendet [Hol03].

2.6.3. Symbolische Verifikation

Bei der symbolischen Verifikation werden Zustände bzw. Zustandsübergänge nicht explizit sondern symbolisch repräsentiert. Dadurch ist es möglich, nicht nur einzelne Zustände/ Zustandsübergänge, sondern gleich Mengen von Zuständen/ Zustandsübergängen zu beschreiben. Durchschnittlich wird eine kompaktere Darstellung erreicht, wodurch die Prüfung effizienter wird, so dass sich auch komplexere Systeme prüfen lassen.

Für die symbolische Repräsentation werden u.a. Binary Decision Diagram (BDD) verwendet, bzw. noch weiter optimierte Varianten, wie *zero suppressed BDD* oder *reduced ordered BDD*. BDD basierte Darstellungen sind immer dann gut geeignet, wenn einfache boolesche Zusammenhänge in der Codierung von Zuständen oder Zustandsübergängen existieren, so dass ggf. ganze Variablen aus den BDDs entfernt werden können. Daher sind BDD basierte Darstellungen für digitale Hardware besonders geeignet.

Das im Folgenden betrachteten Verfahren für SAT Model Checking stellt ebenfalls ein Verfahren zur symbolischen Modellprüfung dar.

2.6.4. SAT basiertes Model Checking

SAT basierte Verfahren versuchen Model Checking auf ein Erfüllbarkeitsproblem (SATisfiability) abzubilden. Bei einem Erfüllbarkeitsproblem wird eine Variablenbelegung für eine kombinatorische boolesche Gleichung gesucht, die zur Erfüllung der Gleichung führt.

Bei SAT-basierten Verfahren wird das Model Checking Problem in ein SAT-Problem transformiert und von einem SAT-Solver analysiert.

2.6.5. Bounded Model Checking

Beim *Bounded Model Checking* werden nur endlich viele Schritte untersucht. Somit lassen sich Eigenschaften einer endlichen Länge analysieren. Entweder wird von einer Menge von Initialmarkierungen aus n Schritte weit gesucht, ob eine bestimmte Markierungsmenge erreicht werden kann. Diese Suche kann vorwärts und rückwärts erfolgen. Es lässt sich entscheiden, ob die gewählten Schritte ausreichen und wenn ja, ob die Eigenschaft erfüllt ist.

Oftmals wird das *Bounded Model Checking* in ein SAT-Problem überführt. Generell gehören die betrachteten symbolischen Verfahren zur gleichen Komplexitätsklasse. Während SAT-basierte Verfahren meist effizienter Arbeiten als die BDD-basierten, gibt es Fälle, in denen dies nicht zutrifft.

2.6.6. Semiformale Verifikation

Semiformale Ansätze werden vor allem dann angewandt, wenn die Systembeschreibung nicht in formaler Form vorliegt, oder aber eine Formale Verifikation, etwa aus Gründen der Komplexität, nicht durchführbar ist.

Zu den semiformalen Verifikationsansätzen werden allgemein Simulationen und Tests gezählt, wenn diese die automatisierte Überprüfung formal definierter Eigenschaften erlauben. In beiden Fällen werden entweder vordefinierte Testläufe durchgeführt und die Ergebnisse der Ausführung bzw. Simulation werden mit dem erwarteten Ergebnis verglichen, oder aber die Erzeugung der Testfälle erfolgt zufallsgesteuert, ggf. durch Parameter beschränkt.

Zu den Ausprägungen semiformaler Verifikationsmethoden gehört das *Test Driven Development* [Wikd, Wikc], bei dem die Testfälle während der Spezifikationsphase definiert werden. Bei Methoden zur Test Pattern Generierung wird versucht, regelbasiert eine möglichst große Pfadüberdeckung mit möglichst wenig Testfällen zu erreichen.

Unabhängig von der entsprechenden Ausprägung soll die Art und Weise betrachtet werden, wie eine simulationsbegleitende semiformale Prüfung temporallogischer Eigenschaften realisiert

werden kann. Obwohl sich die Techniken teilweise sehr stark unterscheiden, liegt ihnen das gleiche Prinzip zu Grunde. Eine temporallogische Formel wird als Akzeptanzautomat (Monitor) instantiiert, der, in Abhängigkeit von den Signalen des zu beobachtenden Systems, seinen Zustand verändert. Dabei werden einige Zustände des Automaten final als *true* (Accept) oder *false* (Reject) markiert und lassen einen Rückschluss auf die Erfüllung der Eigenschaft zu. Da jeweils nur ein einzelner Ablauf geprüft wird, lassen sich nur Eigenschaften linearer Zeit (LTL) prüfen, wobei oftmals auf durch Intervalle beschränkte temporale Operatoren zurückgegriffen wird [JLP+08, RP03, WRKR05, RHKR00].

2.7. Zusammenfassung zum Stand der Technik

Entwicklungsprozesse definieren die systematische Vorgehensweise zur Entwicklung eines Produktes. Aus Sicht der Systemtechnik werden Wechselwirkungen des zu entwickelnden Systems mit seiner Umwelt betrachtet. Ein Entwicklungsprozess durchläuft verschiedene Entwicklungsstufen, wie Anforderungsebene, Spezifikationsebene, Verhaltensebene und Implementierungsebene, in denen eine Betrachtung unter jeweils unterschiedlichen Zielstellungen erfolgt. Um die Wechselwirkungen der einzelnen Phasen dokumentieren und analysieren zu können, wird die Entwicklung durch (ausführbare) Modelle vorangetrieben.

Ansatzpunkt dieser Arbeit ist, dass auf der Basis eines der Implementierung vorgelagerten, ausführbaren Simulationsmodells neben der Validierung auch eine Verifikation ermöglicht wird, um die Qualität des Entwurfs zu verbessern.

Die grobe Eingrenzung dieser Arbeit liegt in der Analyse von Zeiteigenschaften. In diesem Zusammenhang muss eine Abgrenzung zum Begriff *Performance* erfolgen, der wesentlich weiter gefasst ist. Unter dem Begriff Performance verbergen sich eine ganze Reihe nichtfunktionaler Eigenschaften, die in starker Wechselwirkung mit dem Verhalten, also der Dynamik eines Systems stehen. Dazu können statische oder dynamische Analyseverfahren eingesetzt werden. Statische Verfahren nutzen eine abstrahierte Sicht auf das Modell, um aggregierte Werte zu berechnen. Dynamische Verfahren hingegen nutzen die Verhaltensbeschreibung, um durch Simulation aussagekräftige Werte zu erhalten. *Zu bemerken ist dabei, dass es eine Diskrepanz zwischen den zur Validierung genutzten Modellierungssprachen und den zur Analyse/Verifikation genutzten formalen Modellierungssprachen gibt.*

Für die Beschreibung und Analyse des dynamischen Verhaltens von Systemen existieren zahlreiche Ansätze, die unterschiedliche Konzepte verfolgen bzw. unterstützen. *Bei einem Vergleich der Konzepte zeigt sich, dass asynchrone Modelle eine ähnliche Semantik wie Performancemodelle besitzen und daher am ehesten für die formale Beschreibung zum Zwecke der Analyse geeignet sind.*

Für die Bewertung zeitlich-dynamischer (temporaler) Eigenschaften in dynamischen Modellen stehen zahlreiche Verifikationstechniken zur Verfügung. Die Beschreibung der Eigenschaf-

ten erfolgt durch Spezifikationssprachen, die auf temporaler Logik beruhen. Ein Großteil der Verifikationstechniken bezieht sich entweder auf Hardwarebeschreibungen oder setzt die Verfügbarkeit getakteter Automaten voraus. *Für die Prüfung von Zeiteigenschaften im betrachteten Kontext müssen asynchrone Modelle mit quasikontinuierlichen Zeiten unterstützt werden.*

Weiterhin existieren prinzipielle Grenzen formaler symbolischer Verifikation durch den modellabhängigen Zeit- und Speicherbedarf. Symbolische Darstellungen basieren auf der diskreten Darstellung von Zuständen und Zustandsmengen. Diese Darstellung ist prinzipiell nur für zeit- und wertdiskrete Modelle geeignet. Für reellwertige bzw. kontinuierliche Modelle existieren jedoch Abstraktionen auf konvexe Mengen, wobei sich durch Interferenzen dieser konvexen Mengen in der Regel die Komplexität des Zustandsraumes erhöht.

Die Verifikation dient dabei der Prüfung eines Modells auf Eigenschaften, ohne mögliche spätere Weiterentwicklungen durch den Entwicklungsprozess explizit zu berücksichtigen. Die spätere Gültigkeit des Modells bzw. der verifizierten Eigenschaften wird implizit vorausgesetzt. Oftmals wird angenommen, dass es sich um ein finales Modell handelt, oder die zeitlich-funktionalen Eigenschaften unabhängig von der konkreten Realisierung erhalten bleiben. *Die zeitlich funktionalen Eigenschaften stellen bezüglich des später implementierten Modells Approximationen bzw. Erwartungswerte dar, aus denen mittels Methoden der Verifikation Aussagen über voraussichtliche Erreichbarkeit/Erfüllbarkeit von Anforderungen getroffen werden sollen.*

3. Ausarbeitung der Problemstellung

Der Entwurf immer komplexer werdender informationstechnischer Systeme stellt Entwickler vor enorme Herausforderungen. Ausgehend von einer oft sehr wagen Vorstellung eines Produktes wird die Produktentwicklung vorangetrieben. Um Risiken zu minimieren, kommt dazu ein strukturierter Entwicklungsprozess zum Einsatz. Der verwendete Entwicklungsprozess dient in erster Linie der Qualitätssicherung auf organisatorischer und technischer Ebene. Durch die Qualitätssicherung soll erreicht werden, dass Entwicklungsaktivitäten in korrekter Reihenfolge durchgeführt und dass Fehler in Spezifikation, Design und Implementierung möglichst früh erkannt und beseitigt werden.

3.1. Grundsätzliche Problemstellungen

Als besonders problematisch hat sich die Verwendung informaler Beschreibungsmittel während der frühen Phasen des Entwicklungsprozesses erwiesen. Eine mögliche Lösung besteht in der durchgängigen Benutzung von formalen Modellen.

Das fertige Produkt soll im Wesentlichen den bestmöglichen Kompromiss aus verschiedenen Entwurfszielen bilden. Dazu sind verschiedene Freiheitsgrade im Entwurf zu berücksichtigen, wobei für eingebettete Systeme insbesondere Wechselwirkungen zwischen Architektur und Performance zu beachten sind. In diesem Zusammenhang fand eine Verschiebung von Entwurfsaktivitäten hin zu abstrakteren Beschreibungen statt. So weisen Entwurfswerkzeuge, wie Matlab/Simulink und LabView, Techniken zur Codegenerierung auf, die für einen praktischen Einsatz geeignet sind [Cro07].

Durch die Automatisierung der Implementierung wird eine Verkürzung von Entwicklungsaktivitäten erreicht, so dass eine späte Bindung an die verwendete IT-Architektur möglich wird.

Der Steigerung der Produktivität während einzelner Entwicklungsaktivitäten steht der Bedarf an Techniken zur Validierung und Verifikation, insbesondere für frühe Phasen der Systementwicklung gegenüber. Für die Validierung komplexer Systeme sind Methoden des *Virtual Prototypings* besonders geeignet. Dabei wird ein ausführbares Modell erstellt, das die Kernfunktionen des späteren Systems simulieren kann, wobei Abstraktionen in Bezug auf informationstechnische und/oder physikalische Eigenschaften genutzt werden.

Um den Entwurfsprozess zu verbessern, ist es dabei sinnvoll, bereits in solchen Modellen eine Verifizierung vorzunehmen, um die Korrektheit des virtuellen Prototyps nachzuweisen, um ihn

als Referenzmodell heranziehen zu können. Als besonders geeignet haben sich komplexe Simulationssysteme, wie z.B. MLDesigner[11], erwiesen, die eine Modellierung heterogener Systeme erlauben. Neben rein syntaktischen und strukturellen Prüfungen ist eine funktionale Verifikation meist nur durch Simulation möglich. Es existieren zwar eine Reihe von Ansätzen, die auf symbolischen Techniken zur funktionalen Verifikation beruhen. Die Verifikation heterogener und/oder nativer[12] Teilmodelle wird durch solche Ansätze jedoch nur unzureichend abgedeckt.

Wie im Kapitel 2 dargelegt wurde, sind für dynamische Eigenschaften insbesondere Spezifikationssprachen basierend auf temporaler Logik geeignet. Dabei können zwei grundlegende Probleme herausgestellt werden:

- **Problem 1** Verfügbarkeit einer Spezifikationssprache für zeitbeschränkte temporale Eigenschaften in heterogenen Modellen.

- **Problem 2** Verfügbarkeit leistungsfähiger Analysealgorithmen.

3.2. Abgrenzung

Die technische Umsetzung des Entwicklungsprozesses selbst, also die technische Umsetzung der Entwicklungsschritte sowie der dafür benötigten Daten, bietet Raum für Optimierungen. Die Modellierung und Organisation der vielfältigen Informationen muss mit verschiedenen Modellsichten, Abstraktionsniveaus und Anwendungsdomänen-spezifischen Modellen koordiniert werden.

Diese Betrachtungen übersteigen jedoch den Umfang dieser Arbeit und bleiben daher im Wesentlichen außen vor. Es werden ausführbare Modelle zur Analyse oder Implementierung betrachtet, so dass entsprechend dem Einsatzzweck des Modells sämtliche Informationen in ausreichendem Detailgrad[13] verfügbar sein müssen. Aufbauend auf diesen Informationen sollen Eigenschaften modelliert, analysiert und verifiziert werden.

Nicht weiter untersucht werden spezialisierte Ansätze der Modellierung, die domänenspezifische Modellierungsparadigmen nutzen, da ihnen meist Einschränkungen der betreffenden Domäne zugrunde liegen.

[11] Die Unterstützung verschiedener Modellierungssemantiken durch ein einheitliches Framework setzt sich zunehmend durch. Zu Erwähnen ist hier insbesondere SystemC, aber auch kommerzielle Entwicklungswerkzeuge (Matlab/Simulink, LabView) werden in dieser Richtung weiterentwickelt.

[12] Nativ bedeutet hierbei die Verwendung von Teilmodellen, die für eine Verifikation nicht formal nutzbar sind, etwa Anweisungen in Hochsprachen.

[13] Gerade im Bezug auf Zeiteigenschaften ist die Erstellung von generiertem Code oder von Prototypen erforderlich, um zuverlässige Informationen zu erhalten.

3.2.1. Aspekte von Modellen

Aus den Abschnitt 2.3 lassen sich Aspekte zusammenfassen, die für die Modellierung von Systemen in frühen Entwicklungsphasen typisch sind:

- *Ressourcen:* bilden quantitativ Anforderungen an die Ausführbarkeit von Funktionen zu Datenpartikeln ab.
- *Protokolle:* bilden strukturell Wechselwirkungen zwischen Kontrollpartikeln und Zeiten ab.
- *Funktionen:* bilden strukturell Abhängigkeiten zwischen Datenpartikeln, Zeitaspekten (beschränkt Kontrollpartikel) ab.
- *Algorithmen:* bilden die Umsetzung von Funktionen ab.
- *Datenfluss:* durch Datenpartikel werden Informationen modelliert. Die Informationen aus Datenpartikeln werden von Funktionen genutzt und durch Algorithmen interpretiert. Ausgehende Datenpartikel sind von eingehenden Datenpartikeln einer Funktion abhängig. (kausaler Zusammenhang)
- *Kontrollfluss:* durch Kontrollpartikel werden Zusammenhänge modelliert. Kontrollpartikel können als Typisierung auftreten.

Kontrollfluss und Datenfluss stellen die Informationsflüsse dar, die zwischen Teilmodellen ausgetauscht werden, und sind daher nicht eigenständig. Zudem existieren Modellelemente, die der Modellbewertung und Auswertung dienen (Instrumentierung) aber nicht Teil des zu modellierenden Systems sind. Diese Modellaspekte können prinzipiell für die Validierung und Verifikation genutzt werden. In Tabelle 3.1 sind mögliche Realisierungen der Modellaspekte mit den grundlegenden Modellelementen *Zustandsautomat, Prozess und Memory*[14] dargestellt, siehe dazu auch Abschnitt 2.4. In den letzten beiden Spalten wird jeweils die Eignung für die Validierung bzw. Verifikation bewertet.

3.3. Zielstellung

Aus den Problemstellungen gemäß Abschnitt 3.1 lassen sich einige Zielstellungen für die Arbeit ableiten.

- *Ziel 1* Definition einer formalen Syntax und Semantik für die Eigenschaften heterogener Systeme, basierend auf temporaler Logik.

[14] Speicher bzw. globale Zustandsvariable.

Modellaspekt	Modellelement	Validierung	Verifikation
Ressource	Zustandsautomat + Memory	+	++
Ressource	Prozess + Memory	++	+
Protokoll	Zustandsautomat	+	++
Protokoll	Prozess	-	-
Funktion	Prozess	+	+
Funktion	Zustandsautomat + Prozess	+	+
Funktion	Zustandsautomat	-	+
Algorithmus	Prozess	+	-
Algorithmus	Zustandsautomat	+	+

Tabelle 3.1.: Umsetzung von Performancemodellen

- *Ziel 2* Entwicklung von Algorithmen für die (semi-)formale Prüfung:
  - formal für pure DE Modelle,
  - semiformal für heterogene Systeme.

3.3.1. Use Case Formale Verifikation

Bei der Systementwicklung möchte man die Einhaltung von Eigenschaften möglichst formal und vollständig sicherstellen. Hier bieten sich also insbesondere Methoden der formalen Verifikation an. Dabei wird nach Veränderungen des Systemmodells zu dedizierten Zeitpunkten überprüft, ob Eigenschaften erfüllt werden oder nicht.

Da die Prüfung zu dedizierten Zeitpunkten stattfindet, kann die Prüfung vergleichsweise aufwändig sein, also auch mehrere Stunden (Tage) in Anspruch nehmen. Die Verifikation selbst erfolgt ohne bzw. mit wenig Nutzerinteraktion, etwa wie ein Kompiliervorgang. Detaillierte technische Kenntnisse über die Verifikation an sich sollten nicht erforderlich sein. Die zu verifizierenden Eigenschaften sollen dem Modell bzw. dem Teilen des Modells zugeordnet werden können.

Nach Möglichkeit werden die Ergebnisse dem Nutzer in Übersichtlicher Form, möglichst mit Hinweisen zur Behebung von Problemen präsentiert.

3.3.2. Use Case Semiformale Verifikation

Während der Entwicklung und Simulation eines Modells gestaltet sich die Fehlersuche insbesondere dann als schwierig, wenn mehrere Entwickler zusammenarbeiten oder existierende Teilmodelle verwendet werden. Ein häufig auftretendes Problem ist dabei die Fehlerhafte Benutzung von Komponenten, so dass entweder Vorbedingungen nicht eingehalten werden, oder unerwartete Ergebnisse auftreten. Insbesondere bei komplexen Modellen fällt es schwierig, die eigentliche Fehlerursache zu bestimmen.

Assertions beschreiben Annahmen an das Systemverhalten durch Vor- und Nachbedingungen. Dabei werden vor allem Verhaltensweisen an den Schnittstellen von Komponenten beschrieben. Assertions beschreiben dabei oft Invarianten oder temporal oder strukturell lokale Zusammenhänge.

Assertions können für eine semiformale Verifikation eingesetzt werden, indem sie als Eigenschaften mit dem Simulationsmodell cosimuliert werden. Durch die Cosimulation wird entscheidbar, ob Eigenschaften verletzt werden oder nicht. Es ist dabei nicht erforderlich, dass die über die Eigenschaft vollständig entschieden wird, stattdessen werden bestimmte Aktionen ausgelöst, wie das Unterbrechen einer Simulation oder das Ändern von Parametern.

Durch die lokale Zuordnung der Assertions Erkennung von Fehlern auch dann möglich, wenn sie sich (noch) nicht auswirken. Zudem ist, durch die Lokalität, eine Zuordnung auftretender Fehler zu Komponenten vergleichsweise leicht möglich. Nach Möglichkeit sollten Fehler in kompakter bzw. in detaillierter Form im Kontext des Simulationsmodells präsentiert werden.

3.4. Ansatz Formale Verifikation

Möchte man solche Modelle bezüglich dynamischer Eigenschaften formal verifizieren, so ist eine für die Verifikation geeignete, mathematische Beschreibung erforderlich, die sowohl die Struktur als auch die Semantik formal interpretiert.

3.4.1. Direkte Verwendung des Systemmodells

Eine Möglichkeit der Realisierung einer formalen Verifikation bestünde in der direkten Verwendung des Simulationsmodells bzw. Simulators. Die Simulatoren erlauben in der Regel jedoch nur eine eingeschränkte Nutzung zum Zwecke der Verifikation dynamischer Eigenschaften. Die den Systemzustand verkörpernden Datenstrukturen liegen nicht in expliziter Form vor, einige Informationen sind zum Zwecke der beschleunigten Simulation nur intern verfügbar. Die Simulatoren nutzen ein absolutes Zeitmodell, das sich auf die Simulationsdauer bezieht, wohingegen für die Verifikation zur Verkleinerung[15] des Zustandsraumes ein relatives Zeitmodell eingesetzt wird, das die Zeit relativ zum aktuellen Zustand des Systems repräsentiert.

Prinzipiell besteht natürlich die Möglichkeit, das Simulationstarget/ Scheduler[16] zur Erzeugung des Zustandsraumes zu nutzen. Dies setzt voraus, dass der Modellzustand beliebig serialisiert/ deserialisiert werden kann. Dabei würde der Scheduler (bzw. das Target) die Simulation derart beeinflussen, dass der gesamte Zustandsraum des Modells unter gegebenem Random-

[15] Auch für Systeme mit potentiell unendlicher Laufzeit kann so eine endliche zeitbehaftete Zustandsrepräsentation als Regionengraph gewonnen werden.

[16] Durch ein Target besteht die Möglichkeit strukturell Einfluss auf das Simulationsmodell zu nehmen. Ein Scheduler ermöglicht den Ablauf der Abarbeitung zu beeinflussen.

seed[17] durch laufen würde. Folgende Probleme können dabei identifiziert werden:

- (1) Bei stochastischen Simulatoren tritt das Problem auf, dass die Variabilität der Zeit zur Generierung zeitabhängig erreichbarer Zustände nicht einfach realisierbar ist, da diese ggf. vom Randomseed abhängt. Hier ist also eine spezielle Behandlung der die Zeit beeinflussenden funktionalen Blöcke erforderlich.
- (2) Vielfach werden dynamische Datenstrukturen genutzt, die von Verifikationswerkzeugen derzeit noch unzureichend unterstützt werden.

Ein weiteres Hindernis für die Nutzung der Simulationsumgebung für die formale Verifikation bestehen in der Kapselung durch den Hersteller. Selbst wenn theoretisch eine Nutzung des Simulators besteht, ist eine Beeinflussung zur Zustandsgenerierung nicht oder nur eingeschränkt möglich.

3.4.2. Verwendung von Annotationen zur Modellgenerierung

Die Voraussetzungen für die formale Verifikation sind im Simulationsmodell meist nicht direkt gegeben, es ist also eine Transformation in ein geeignetes Modell erforderlich. Ein wesentliches Problem dabei ist der hohe Detailgrad der betrachteten Modelle. Möchte man trotzdem eine Formale Verifikation realisieren, so müssen Teile des Modells abstrahiert werden, wofür zusätzliche Informationen erforderlich sind. Ideal wäre die Verwendung funktionaler Eigenschaftsspezifikationen, die bereits von konkreten Realisierungen abstrahiert und zusätzliche Informationen, etwa zum Zeitbedarf enthält.

Wegen der Überführung in eine andere Darstellungsform wird die Vorgehensweise im Folgenden als *Transformation-basierte Verifikation* bezeichnet und ist in Abbildung 3.1 dargestellt. Zunächst wird das im Modellierungswerkzeug erstellte und durch Eigenschaftsspezifikationen ergänzte Modell interpretiert. Diese Spezifikationen enthalten Anforderungen an das Modell sowie Ergänzungen zur funktionalen Systembeschreibung. Anschließend wird das Ausgangsmodell in ein geeignetes formales Modell zum Zwecke der Verifikation überführt. Dieses formale Modell enthält sowohl die Systembeschreibung als auch die Beschreibung der Anforderungen in geeigneter Form. Diese Anforderungen werden im formalen Modell durch ein Werkzeug zur Verifikation geprüft. Die Ergebnisse können durch Rücktransformation ins Modellierungswerkzeug interpretiert werden.

Für die weiteren Betrachtungen werden zwei Hypothesen aufgestellt:

1. Durch Erhaltung der Struktur des Ausgangsmodells ist eine schrittweise Interpretation elementarer Eigenschaften und Anforderungsdefinitionen zu einem für die Verifikation geeignetem Modell möglich.

[17] Für die Eineindeutigkeit (Wiederholbarkeit) eines Simulationslaufes ist die Initialisierung des Zufallsgenerators zu berücksichtigen. Der Zufallsgenerator wird für die Realisierung nichtdeterministischer Effekte benötigt.

2. Durch die Nutzung der strukturellen bzw. funktionalen Hierarchie kann eine Abstraktion/Effizienzsteigerung erreicht werden.

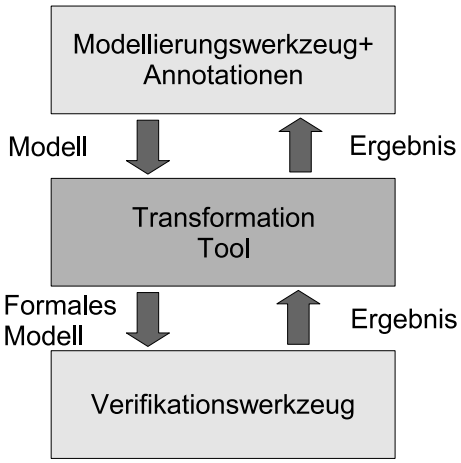


Abbildung 3.1.: Transformation-basierte Verifikation

3.4.3. Fragestellungen zur Transformation-basierten Verifikation

Für die Umsetzung dieses Prinzips ergeben sich folgende Anforderungen und Fragestellungen:

1. A1: Ergänzung des Designmodells um Anforderungen und Zusicherungen.
   - F1.1: Welche Form ist für die Anforderungsmodellierung geeignet?
   - F1.2: In welcher Art und Weise können dem Designmodell Anforderungen hinzugefügt werden?
2. A2: Anreicherung des Designmodells zum Zwecke der Verifikation.
   - F2.1: Welche Informationen des Designmodells werden für die formale Verifikation benötigt? (Menge formale Verhaltensbeschreibung)
   - F2.2: Welche Aspekte des Designmodells sind Formal interpretierbar? (Schnittmenge von Designmodell und formaler Verhaltensbeschreibung)
   - F2.3: Welche zusätzlichen Informationen sind für die formale Verifikation erforderlich? (Differenz: formale Verhaltensbeschreibung minus Designmodell)
3. A3: Formale Interpretation des Designmodells.
   - F3.1: Wie kann das angereicherte Designmodell formal interpretiert werden?
   - F3.2: Welche formale Interpretation ist für konkrete Verifikationswerkzeuge am besten geeignet?

- F3.3: Wie kann die Transformation des Designmodells in die mathematische Darstellung realisiert werden?
- F3.4: Ist eine vollständige Automatisierung möglich?

3.5. Ansatz Semiformale Verifikation

Eine alternative Vorgehensweise zur formalen Verifikation stellt die semiformale Verifikation dar. Dabei wird zwar eine formale Prüfung von Eigenschaften realisiert, allerdings kann die Vollständigkeit der Prüfung nicht sichergestellt werden. Angewendet werden solche Verfahren in der Regel dann, wenn entweder keine mathematisch formale Beschreibung des Modells vorhanden ist oder aber das Modell zu komplex ist, um eine vollständige Prüfung gewährleisten zu können. Dann stellt die Generierung geeigneter Pfade im Verhalten des Modells eine manuelle Tätigkeit dar.

Im Falle der betrachteten Modelle sind beide Voraussetzungen gegeben. Die Modelle enthalten oft Legacy-Code, stellen also keine rein mathematisch interpretierbare Beschreibung dar. Weiterhin sind die Modelle oft so komplex, dass eine Formale Verifikation nicht in Frage kommt, unter anderem auch durch die Verwendung mehrerer Berechnungsdomänen innerhalb eines Systemmodells.

Während die formale Verifikation eine vollständige (symbolische) Simulation in der Regel auf mathematischer Ebene erfordert, werden bei der semiformalen Prüfung die Signale und Zustände des simulierten Modells im Kontext Eigenschaften interpretiert. Praktisch bedeutet dies, dass die Eigenschaften ins Simulationsmodell eingebunden werden können und eine Prüfung während der normalen Simulation erfolgt.

Für die Assertion-basierte Verifikation werden mehrere Konzepte vereint. Es werden Eigenschaften vornehmlich bezogen auf Schnittstellen von Komponenten definiert, um das von außen sichtbare Verhalten der Schnittstellen auf Konformität zu prüfen, und implizite Annahmen über die Verwendung einer Komponente explizit darzustellen. Gleichzeitig werden Eigenschaften so definiert, dass sowohl eine Cosimulation als auch eine formale Verifikation prinzipiell möglich ist.

Dazu wird allgemein eine einheitliche Abstraktion von Signalen des Ausgangsmodells vorgenommen, und eine formale Repräsentation der Eigenschaften in Form von Automaten implementiert, um eine automatische Prüfung definierter Eigenschaften zu realisieren. Allerdings wurde eine solche Vorgehensweise bislang nicht für eine heterogene Modellumgebung realisiert. Das im Folgenden dargestellte Konzept berücksichtigt die sich aus der Heterogenität ergebenden Anforderungen für die Assertion-basierte Verifikation.

Um die Anwendbarkeit für Systemmodelle zu gewährleisten, müssen mehrere Anforderungen erfüllt werden, die insbesondere aus der Unterstützung heterogener Modelle bezüglich verwen-

deter Berechnungsdomänen resultieren:

- Etablierung einer gemeinsamen Modellbasis zur Integration der Eigenschaften heterogener Teilmodelle. (heterogene Eigenschaften)
- Definition einer geeigneten Abstraktion von Signalen des Ausgangsmodells. (Signalabstraktion)
- Berücksichtigung der Interaktion der Domänen verschiedener Teilmodelle. (Vorverarbeitung)
- Erhaltung der Semantik bei der Interpretation von Eigenschaften verschiedener Domänen durch Formale Repräsentation in Form von Automaten. (Eigenschaftsinterpretation)
- Automatische Bewertung der Assertion während der Simulation. (Assertion-Cosimulator)

Aus diesen Anforderungen ergibt sich eine Struktur, siehe Abbildung 3.2, die im Folgenden erläutert wird. Die Definition der *Eigenschaften* stellt auf Modellierungsebene die Basis der zur Verifikation verwendeten Assertions dar. Für ein Signal muss eine geeignete Abstraktion für die formale Prüfung realisiert werden. Im Hinblick auf die Synchronisation der Informationen verschiedener Domänen ist es naheliegend, die Mechanismen des Modellierungswerkzeugs für diese *Signalabstraktion* zu nutzen und diese Abstraktion auch dort zu realisieren. Zur Berücksichtigung der Interaktion verschiedener Domänen und die dynamische Instantiierung von Assertions ist eine *Vorverarbeitung* der abstrahierten Signale erforderlich. Durch die *Interpretation von Eigenschaften* wird eine Formale Darstellung der Eigenschaften erreicht, die zur eigentlichen Prüfung verwendet werden kann. Der *Assertion-Cosimulator* kann anschließend diese Darstellung nutzen, um eine automatisierte Bewertung der Assertions während der Simulation zu realisieren.

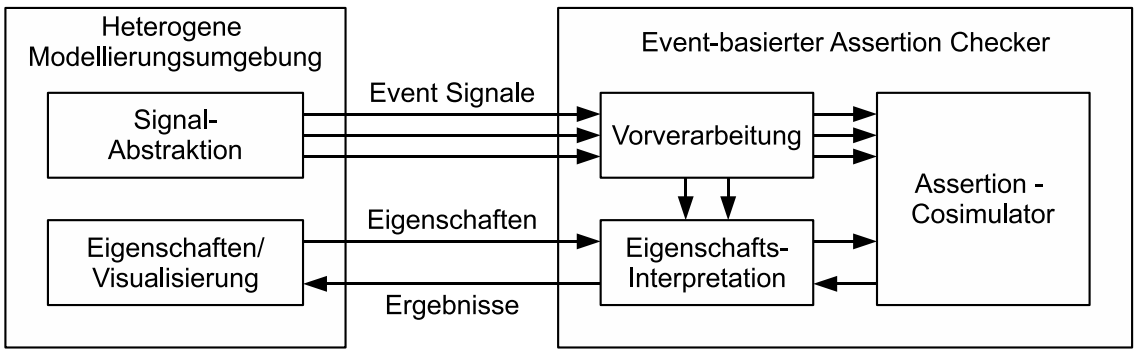


Abbildung 3.2.: Konzept zur Assertion-basierten Prüfung in heterogenen Modellen

3.5.1. Fragestellungen zur Assertion-basierten Verifikation

Im Kontext der Systemmodellierung ergeben sich folgende Anforderungen und Fragestellungen für eine semiformale Prüfung.

1. A1: Definition von Eigenschaften für heterogene Modelle.
   - F1.1: Welche Art der Eigenschaftsbeschreibung ist für System-Level Assertions geeignet?
   - F1.2: Welche Aspekte müssen aufgrund der Heterogenität berücksichtigt werden?
   - F1.3: Welche gemeinsame Basis ist für die Semantik der Eigenschaften heterogener Modelle geeignet?
   - F1.4: Wie können bestehende Eigenschaften in die gemeinsame Basis überführt werden?
   - F1.5: Wie können Eigenschaften von Teilen unterschiedlicher Berechnungsdomänen in geeigneter Weise verknüpft werden?
2. A2: Entwicklung einer formalen Methode zur Prüfung der definierten Eigenschaften.
   - F2.1: Welche Repräsentation ist für die semiformale Prüfung der Eigenschaften geeignet?
   - F2.2: Wie können Eigenschaften in die formale Repräsentation überführt werden?
3. A3: Entwicklung einer Methode zur Integration von Simulation und Eigenschaftsprüfung.
   - F3.1: Welche Anpassungen sind für die Integration der Eigenschaftsprüfung in die Simulation erforderlich?
   - F3.2: Wie kann die Kopplung von Simulation und Eigenschaftsprüfung realisiert werden?
   - F3.3: Welche Möglichkeiten der Auswertung sind realisierbar?

3.6. Vorgehensweise

Ausgehend der Problem- und Fragestellungen, die sich aus der Analyse des Standes der Technik ergeben wurden Eingrenzungen zum Einsatzbereich und zum funktionalen Umfang der Verifikationsmethoden vorgenommen und Zielstellungen definiert. Dazu wurden zwei Anwendungsszenarien definiert, die unterschiedliche Anforderungen an die eingesetzte Verifikationsmethode stellen. Für jeden der beiden Anwendungsfälle (Use Cases) wurde ein entsprechender Ansatz entwickelt, der für die Umsetzung der spezifischen Anforderungen prinzipiell geeignet ist. Für die Anwendung der formalen Verifikation wird ein Transformation-basierter Ansatz

vorgeschlagen, während für die semiformale Verifikation der Assertion-basierte Ansatz gewählt wird.

Für den Einsatz bzw. die Realisierung einer Verifikationsmethode kann eine prinzipielle Vorgehensweise beschrieben werden. Zunächst müssen die zu analysierenden Eigenschaften sowie die Gegebenheiten des Ausgangsmodells analysiert werden. Aufbauend auf diese Analyse kann eine geeignete Spezifikationssprache definiert werden und die Festlegung von der einzusetzenden Verifikationstechnik erfolgen.

Entsprechend ist der Aufbau der Arbeit gewählt. Zunächst wird in Kapitel 4 spezifiziert, welche Eigenschaften beim jeweiligen Verifikationsansatz unter Berücksichtigung des jeweils gegebenen Modells infrage kommen. Für den jeweiligen Ansatz wird eine geeignete Spezifikationssprache definiert. Die Betrachtungen erfolgen zunächst gemeinsam, da sich eine Unterscheidung erst durch die detaillierte Betrachtung der Anforderungen und Gegebenheiten ergibt.

Für die betrachteten Ansätze kommen unterschiedliche technische Lösungen zur Verifikation zum Einsatz. Daher werden die konzeptionellen Verfeinerungen der Ansätze im weiteren Verlauf separat betrachtet und realisiert. In Kapitel 5 wird der Transformation-basierten Ansatz betrachtet, während in Kapitel 6 der Assertion-basierte Ansatz realisiert wird. In Kapitel 7 erfolgt die vergleichende Gegenüberstellung der realisierten Ansätze. Gleichzeitig wird untersucht, welches Potential die Ansätze voraussichtlich bieten. In Kapitel 8 wird die Arbeit zusammengefasst, die Ergebnisse kritisch diskutiert und ein Ausblick auf mögliche Nachfolgearbeiten geboten.

4. Eigenschaftsspezifikation

Ziel dieses Kapitels ist die Definition einer Beschreibungssprache für Eigenschaften in reaktiven Systemen als Ergänzung zu ausführbaren Modellen, die zur Verifikation genutzt werden kann. Dazu wird zunächst in Abschnitt 4.1 ein Konzept zur Eigenschaftsspezifikation aufgestellt, wobei Anforderungen an Spezifikationssprachen herausgearbeitet und analysiert werden.

Daraufhin werden im Abschnitt 4.2 Klassen von Eigenschaften analysiert bewertet und eine formale Darstellung vorgeschlagen. Anschließend wird in Abschnitt 4.3 untersucht, welche Eigenschaften durch die Semantik der verwendeten Modellierungsmittel aufgrund der verwendeten dynamischen Berechnungsmodelle (MoC) jeweils sinnvoll sind. Dem schließt sich in Abschnitt 4.4 die Untersuchung von Wechselwirkungen der verschiedenen Berechnungsmodelle in heterogenen Modellen an und es werden Auswirkungen bezüglich heterogener Eigenschaftsdefinitionen diskutiert. Danach wird in Abschnitt 4.5 eine Spezifikationssprache für Eigenschaften für die nachfolgenden Kapitel vorgeschlagen. Abschließend werden die Ergebnisse zusammengefasst und bewertet.

4.1. Grundkonzept zur Eigenschaftsspezifikation

Die Beschreibung von Echtzeitsystemen folgt in der Regel als Transformation von Steuerfluss und Datenfluss [WM91]. Dabei wird oftmals eine Unterteilung in Vor- und Nachbedingungen vorgenommen.

Die Interpretation von Vor- und Nachbedingung hängt dabei von der Semantik des betrachteten Modells ab. Handelt es sich um Ein- und Ausgehende Signale bzw. um Vor- und Nachzustände, so können Vor- und Nachbedingung invariant gelten oder paarweise einander bedingen. Neben der gleichzeitigen Gültigkeit mehrerer Paare von Vor- und Nachbedingungen können insbesondere auch unvollständige Spezifikationen vorliegen.

Methoden der Assertion-basierten Verifikation nutzen paarweise Vor- und Nachbedingungen, um eine Prüfung als Assume/ Guarantee zu realisieren [WK04].

Insbesondere Methoden des *Theorem Proovings* [Kro99, CH04] bedienen sich solcher Beschreibungen, um, ausgehend von der transformativen Beschreibung der Funktion als Relation zwischen Signalen und Zuständen (Obligation), über die Struktur, Komposition und Vernetzung von Komponenten auf Eigenschaften zu schließen (Proof). Voraussetzung solcher Methoden ist insbesondere die Vollständigkeit der funktionalen Beschreibung, die hier - im Gegensatz zu der

Assertion-basierten Verifikation - nicht seiteneffektfrei bezüglich der referenzierten Signale ist.

4.1.1. Anforderungen an Eigenschaftsspezifikationen

Seiteneffektfrei Grundsätzlich sollen Spezifikationssprachen für Eigenschaften frei von Seiteneffekten sein. Dies bedeutet, dass eine Sprache nicht beschreibt, *wie* eine Eigenschaft geprüft wird, sondern *was* geprüft wird. Ein Ausdruck, der zur Prüfung verwendet wird, besitzt somit keine Rückwirkung zum analysierenden System.

Verwendungsmöglichkeiten Die Verwendungsmöglichkeiten (Beschränkungen) einer Spezifikationssprache müssen bekannt sein. Neben dem Aspekt der Spezifikation betrifft dies insbesondere auch den Aspekt der Verifikationsmethodik.

Problemraum Der Problemraum der Eigenschaftsspezifikation muss bekannt sein. Im Kontext der Verifikation bedeutet dies, dass Sichtweise und Abstraktion zum verwendeten Modell passen muss. Wird dies nicht berücksichtigt, so besteht die Gefahr, dass Eigenschaften spezifizierbar sind, die nicht zum Modell passen, weil die Eigenschaften nicht beobachtbar sind, oder im Kontext des Modells eine andere Semantik bekommen.

Modellkontext Die Eigenschaften müssen im Kontext des Modells interpretiert werden. Die Eigenschaften müssen also syntaktisch und semantisch einen eineindeutigen Bezug zu Modellelementen haben. Gegebenenfalls ist auch eine direkte Zuordnung zu Modellelementen erforderlich.

4.1.2. Konzept

Seiteneffektfrei Die Abwesenheit von Seiteneffekten für die zu konstruierende Sprache zur Eigenschaftsspezifikation wird vorausgesetzt. Prinzipiell ist jedoch eine Modellgenerierung aus einer vollständigen Menge von Eigenschaften nicht ausgeschlossen.

Verwendungsmöglichkeiten Zur Verifikation einer Eigenschaft existieren verschiedene Ansätze, siehe Abschnitt 2.6. Relevant ist einerseits die Prüfung der Eigenschaften selbst im Rahmen des vorhandenen Systemmodells. Hierbei ist die Komplexität der Modelle zu berücksichtigen, wodurch eine direkte Verifikation nicht möglich scheint. Es ist also eine geeignete Abstraktion erforderlich, die durch die Spezifikationssprache realisiert werden muss.

Hier bietet es sich an, das Paradigma der *hierarchischen Verifikation* zu unterstützen, bei dem eine konforme Komposition auf der Ebene der Eigenschaften geprüft wird, also ob die Eigenschaften eines Teilsystems durch die Eigenschaften der Subsysteme realisiert werden. Ist

dies nicht erfüllt, so liegt entweder eine fehlerhafte/ unvollständige Spezifikation vor, oder eine detailliertere Spezifikation ist im Kontext des Problemraumes nicht möglich oder sinnvoll.

Ein weiterer geeigneter Ansatz unter Berücksichtigung der Komplexität besteht in der *semiformalen Verifikation* von Eigenschaften, also in der Cosimulation von Eigenschaften (Assertions) während der Simulation. Zwar ist dadurch keine Vollständige Verifikation möglich, allerdings kann die Fehlersuche im Entwurfsmodell bei Verwendung geeigneter Szenarien automatisiert und somit Entwicklungszeit reduziert werden.

Problemraum Die Abgrenzung des Problemraumes ergibt sich aus der Klasse der zu verifizierenden Modelle. Im Falle der funktionalen Modelle in frühen Phasen des Entwicklungsprozesses, gemäß der Abschnitte 2.2, 2.3 und 2.4 ist eine Eingrenzung auf heterogene, bzw. Event-basierte Systemmodelle vorzunehmen. In Bezug auf Zeiteigenschaften lassen sich zwei wesentliche Abstraktionsebenen finden. Zum einen erfolgt eine abstrakte Modellierung mittels zeitbehafteter technischer Prozesse. Zum anderen bekommen diese Prozesse in der verfeinerten Darstellung zusätzliche Informationen bezüglich des funktionalen Verhaltens, wobei vom Zeitverhalten abstrahiert wird, so dass die Abstraktion der idealisierten Prozesse nur mit Einschränkungen oder gar nicht möglich ist. Die dadurch entstehenden Modelle berücksichtigen physikalische Eigenschaften nur so weit, wie dies für die Belange des Modells erforderlich ist.

Diese Abstraktionsebenen sind oft in verschachtelter Form zu finden. Das bedeutet, dass einzelne Teile des Systems unterschiedlich exakt bezüglich Zeit und Funktion spezifiziert sind.

Modellkontext Die in Betracht kommenden Modelle sind kompositional hierarchisch in Blöcke strukturiert. Blöcke besitzen fest definierte Schnittstellen und kommunizieren über Signale miteinander. Zudem existieren Parameter und bezüglich der Sichtbarkeit beschränkte Modelle. Eigenschaften müssen einem Block zugeordnet werden, der strukturell die referenzierten Modellelemente vollständig enthält.

4.2. Klassifizierung von Eigenschaften

Es lassen sich verschieden Sichtweisen auf Eigenschaften reaktiver Systeme finden. Unter Sichtweise wird die Modellierung der Eigenschaften unter konzeptionelle Aspekten betrachtet. Im Folgenden werden konzeptionelle Aspekte der Prozess-basierten Sicht (vergleiche Abschnitt 2.4.3) und der Zustands-basierten Sicht (vergleiche Abschnitt 2.5) näher erläutert. Dabei werden insbesondere die Aspekte Zeitbezug/Modellierung und Hierarchie betrachtet.

4.2.1. Prozessbasierte Eigenschaften

Sichtweise

Prozessbasierte Eigenschaften betrachten die Abarbeitung eines Systems als Folge der Abarbeitung zeitbehafteter technischer Prozesse. Die Aktivierung von Prozessen kann zeitlich gesteuert sein, getrieben von Daten oder Ereignissen, oder als Mischform bestehen. Innerhalb eines Systems können gleichzeitig mehrere Prozesse aktiv sein. Eine Zuordnung von Prozessen zu Ressourcen ist optional möglich, aber bei rein funktionaler Betrachtung nicht zwingend erforderlich. Abbildung 4.1 zeigt im linken Bereich die Struktur eines einfachen Prozessmodells mit *Inputs*, *Outputs*, und einer Zeitbewertung.

Charakteristisch für einen Prozess sind somit die benötigten Informationen (Abhängigkeiten) als Ereignisse und der Zeitbedarf für die Abarbeitung des Prozesses. Die Abarbeitung solcher Prozesse wird dabei als zeitinvariant betrachtet, was bedeutet, dass ein Prozess nicht von Informationen seiner vorherigen Abarbeitung abhängt. Vorteil dieser Zeitinvarianz ist die Entkopplung von funktionalem und zeitlichem Verhalten. Im Spezialfall können solche Informationen als ein- und ausgehende Informationsflüsse modelliert werden.

Semantisch bedeutsam ist die Zuordnung eines Ausgangssignals als Reaktion zu einer Menge von Eingangssignalen.

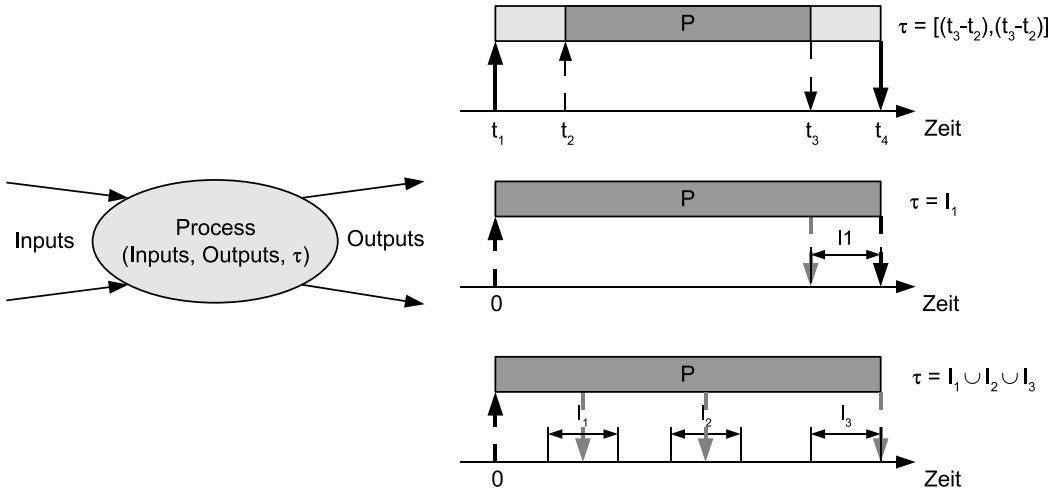


Abbildung 4.1.: Einfaches Prozessmodell

Zeitbezug

Im rechten Teil der Abbildung 4.1 sind verschiedene Varianten der Zeitbewertung dargestellt. Oben sieht man die allgemeine Abarbeitung, wobei zunächst durch Eingabewerte (Zeitpunkt t_1) der Prozess zur Aktivierung vorbereitet wird. Anschließend erfolgen der Start des Prozesses (t_2),

die Beendigung des Prozesses (t_3) und das Auslesen der Ausgabewerte (t_4). Die Unterscheidung zwischen Bereitschaft und Aktivierung ist dem Umstand geschuldet, das eine Synchronisation aller Eingangswerten, vergleichbar mit dem Datenflussparadigma vorausgesetzt wird, während ein Prozess erst nach dem Auslesen der Ausgabewerte wieder aktiviert werden darf. Oftmals ist diese Unterscheidung nicht erforderlich, so dass durch die Synchronisation aller erforderlichen Eingabewerte der Start des Prozesses erzwungen wird und am Ende der Abarbeitung des Prozesses zwangsweise die Ausgabe erfolgt (Abbildung 4.1, rechts Mitte). Die Dauer der Prozessbearbeitung kann als Intervall notiert werden. Je nach Detailgrad kann der Zeitbedarf bei starker Schwankung als Vereinigung disjunkter Intervalle dargestellt werden, siehe Abbildung 4.1 rechts unten. Sofern strukturell bedingt, ist auch eine von der Eingabe abhängige Abarbeitungsdauer möglich. Charakteristisch ist jedoch, dass die Dauer der Abarbeitung nicht von der vorherigen Aktivierung abhängig ist.

Wenn unterschiedlich häufig aktivierte Prozesse existieren bzw. sich unterschiedliche Instanzen einer Prozesskette, z.B. durch Informationsaustausch oder Ressourcenbelegung Wechselwirken, ist die Betrachtung der Nebenläufigkeit erforderlich. Alternativ wird eine Makroperiode der Prozesskette explizit mit mehreren Instantiierungen untersucht. Ein weiterer Aspekt betrifft die strukturelle Dekomposition hierarchischer Prozesse. Ist die Anzahl gleichzeitig aktiver hierarchischer Prozesse bzw. die Aktivierungsrate nicht bekannt, so kann nicht entschieden werden, ob es sich um eine gültige Dekomposition handelt.

Prinzipiell ist also eine gleichzeitige Abarbeitung des gleichen Prozesstyps in mehreren Instanzen denkbar, was im Folgenden als innere Nebenläufigkeit bezeichnet wird. Falls Ressourcen betrachtet werden, kann entweder eine parallele Abarbeitung vorliegen, also die Abarbeitung durch mehrere Ressourcen, oder es erfolgt eine serielle Abarbeitung im Sinne einer Pipeline, wobei Aktivierungen und Beendigungen dem FIFO-Prinzip entsprechen. Die Art der Nebenläufigkeit wirkt sich auf die (De-)Komposition von Aktivierungsraten aus. Besteht eine innere Nebenläufigkeit, so ist die Charakterisierung der maximalen Anzahl nebenläufiger Aktivierungen direkt (Anzahl) oder indirekt, etwa durch die Aktivierungsrate, möglich.

Hierarchie

Eine hierarchische Modellierung ist in verschiedener Weise möglich. Rein strukturell betrachtet, können mehrere Prozesse in einem Kontext beschrieben werden. Die Aktivierung unterschiedlicher Teilprozesse stellen dann alternative Aktivierungen eines abstrakteren Prozesses dar. Somit werden Details bezüglich der Existenz mehrerer Teilprozesse durch verschiedene Varianten eines einzelnen abstrakten Prozesses vereinfacht dargestellt. Abbildung 4.2 verdeutlicht diesen Zusammenhang. Hierbei sind zwar mehrere Prozesse abstrahiert, allerdings findet keine Wechselwirkung statt.

Funktional hierarchisch kann die Abarbeitung eines komplexen Prozesses als Komposition

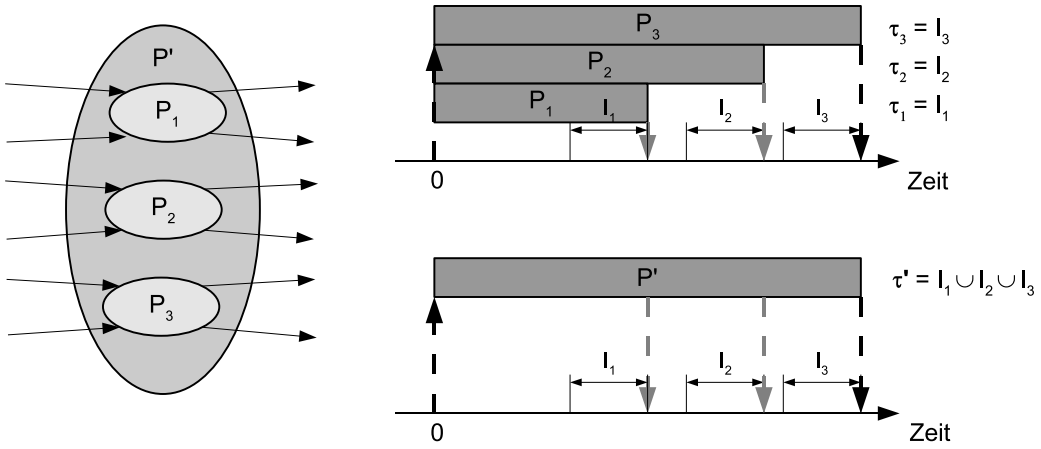


Abbildung 4.2.: Strukturelle Komposition

einfacherer Prozesse modelliert werden. Dabei werden Details über den Zeitbedarf einzelner Teilprozesse und über die innere Struktur verborgen. Abbildung 4.3 zeigt diesen Zusammenhang anhand der Komposition der Prozesse P_1, P_2 und $P3$, die zu einem hierarchischen Prozess P' komponiert sind.

Ein Beispiel für eine solche Dekomposition ist die Zerlegung eines komplexen Regelungsalgorithmus als Prozess in seine Teilalgorithmen, wobei jeder Algorithmus für sich ebenfalls einen Prozess repräsentiert.

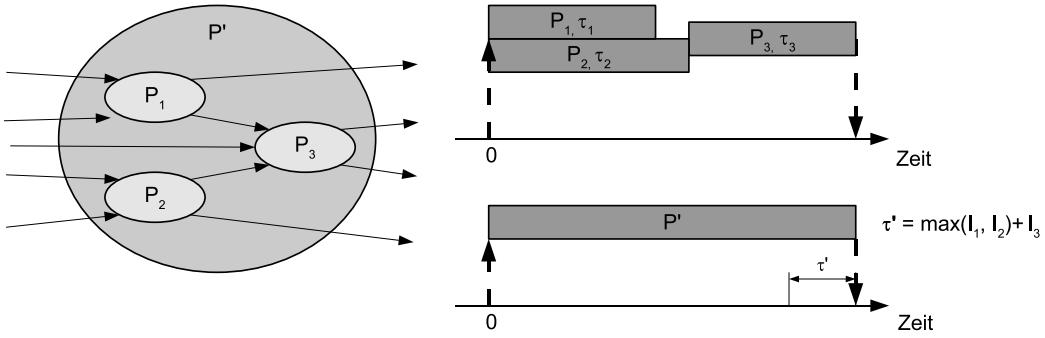


Abbildung 4.3.: Hierarchische Komposition

Modellierung

Für eine Implementierung eines Prozessmodells ist, zumindest zu Verwaltungszwecken, die Modellierung eines Prozesszustandes erforderlich. Für die alleinige Modellierung von Performance sind dagegen nur Ereignisse, wie Kommunikation, Aktivierung, Ende und Unterbrechung erforderlich. Die Unterbrechung eines Prozesses ist optional, und nur bei einer Verdrängung auf einer Ressource interessant. Die Kommunikation wird meist nicht explizit modelliert, sondern

als Abhängigkeit dargestellt. In diesem Falle sind rein Aktivierung und Beendigung eines Prozesses interessant, die durch zur Verfügung gestellte Eingabemengen (Eingabeevents) und Ausgabemengen (Ausgabeevents) dargestellt werden können. Dabei bietet sich eine normalisierte Darstellung an.

Die Synthese der prozessbasierten Eigenschaften kann durch ein Discrete-Event Modell erfolgen. Dabei werden Ausgabeevents als Folgerung von Eingabeevents beschrieben. Die Prüfung von prozessbasierten Eigenschaft kann mittels Monitoren erfolgen, die prüfen, ob die beschriebenen Relationen zwischen Eingabeevents und Ausgabeevents beobachtbar sind. Solche Monitore können u.a. mittels dynamisch instantiierten Automaten realisiert werden. Dabei wird die zu prüfende Eigenschaft als abstrakter Prozess betrachtet, der durch Prozesse des zu prüfenden Modells realisiert wird.

In komplexen Prozessen kann, bedingt durch nebenläufige Aktivierungen von Prozessketten, das Problem auftreten, dass Ereignisse den Instanzen der einzelnen Prozesse nicht eineindeutig zuzuordnen sind, ohne zusätzliche Annahmen zu treffen. Dieses Problem kann durch die explizite Modellierung der Zuordnung von Ereignissen zu Prozessinstanzen innerhalb des zu analysierenden Modells gelöst werden [PF07]. Verfahren der Schedulinganalyse nutzen zwar ebenfalls eine solche Zuordnung, analysieren jedoch einzelne Pfade, anstatt die korrekte Komposition/ Dekomposition zu prüfen.

4.2.2. Zustandsbasierte Eigenschaften

Sichtweise

Zustandsbasierte Eigenschaften beschreiben partiell den Verlauf elementarer Aussagen über die Zeit. Elementare Aussagen beziehen sich auf den Zustand eines Systems, Systemteils oder Signals (allgemein Komponente). Zustandsänderungen werden durch Zeit, Signale oder Signaländerungen (Ereignisse) ausgelöst. Zustandsübergänge selbst können ebenfalls zur Änderung von Signaländerungen (Emittierung von Ereignissen) führen. Als Signale werden die strukturell koppelnden Elemente bezeichnet, wohingegen Ereignisse Instanzen von Änderungen dieser Signale repräsentieren.

Beschreibt man jede Zustandsänderung als Ereignis, so lassen sich zustandsbasierte Eigenschaften in temporale Eigenschaften überführen, die sich auf Startzustände und Änderungsereignisse beziehen. Diese Sichtweise wird für alle nachfolgenden Überlegungen genutzt.

Charakteristisch sind die Beständigkeit der strukturellen Zuordnung eines Zustandes zu einer Komponente sowie die zeitlich variante Reaktion auf Ereignisse.

Semantisch kann keine Zuordnung eines Ausgangssignals zu einer bloßen Menge von Eingangssignalen getroffen werden, da neben dem Auftreten von Ereignissen auch der Zustand und die zeitlichen Relationen von Bedeutung sind. Im Spezialfall kann die gesamte Historie der Eingangsereignisse für ein einzelnes Ausgangsereignis relevant sein.

Die Beobachtung von zustandsbasierten Eigenschaften kann auf unterschiedliche Art beschrieben werden, je nachdem, ob interne Zustände beobachtbar sind oder nicht. Ist der interne Zustand beobachtbar kann eine zustandsabhängige Reaktion (Ausgangsereignisse, Folgezustand) als Reaktion auf Zustand, Zeit und Eingangsereignisse beschrieben werden. Diese Sichtweise setzt Wissen über die innere Struktur der Komponente voraus. Andererseits kann die Beschreibung schnittstellenbezogen sein, d.h. es werden zeitlich quantitative Beziehungen zwischen Folgen von Eingangsereignissen und Ausgangsereignissen hergestellt, ohne die interne Struktur der Komponente zu kennen.

Zeitbezug

Zeitliche Relationen bestehen zwischen Eingangsereignissen, internen Zuständen und Ausgangsereignissen. Es können dabei temporale Relationen, Mengenbezogen-quantitative Relationen oder zeitlich-quantitative Relationen bestehen.

Temporale Relationen beschreiben die Reihenfolge des Auftretens von Ereignissen oder Zuständen, ausgedrückt durch temporale Operatoren. Mengenbezogen-quantitative Relationen beschreiben absolute Häufigkeiten von Ereignissen. Zeitlich-quantitative Relationen beschreiben temporale Relationen mit zeitlichen Beschränkungen, ausgedrückt durch zeitlich beschränkte temporale Operatoren.

Hierarchie

Hierarchische Beziehungen können in Bezug auf die Komposition von Komponenten erfolgen, das heißt, die Teilkomponenten und koppelnde Signale sind nicht im Detail sichtbar. Eine solche Abstraktion ist besonders dann sinnvoll, wenn die interne Struktur von Komponenten (Zustände) nicht bekannt ist. Sind Eigenschaften der detaillierten Strukturen bekannt, können Eigenschaften der abstrakten Struktur ggf. aus der Kompositionsbeziehung durch Inferenz abgeleitet werden.

Ist die interne Struktur bekannt, können hierarchische Zustandsbeziehungen genutzt werden, um eine Abstraktion zu erreichen. Sind Unterzustände[18] eines hierarchischen Zustandes nicht für die betrachtete Eigenschaft erforderlich, so können die betreffenden Details dieser Unterzustände vernachlässigt werden.

Modellierung

Für die Modellierung zustandsbasierter Eigenschaften können finite temporale Eigenschaften verwendet werden. Das Alphabet, also die Grundmenge der Basisaussagen der beschreibenden formalen Sprache, besteht aus Ereignissen (Eingangs- und Ausgangsereignissen), Internen

[18] die hierarchischen Verfeinerungen eines Zustandes

Zuständen, sowie Strukturinformationen des Systems. Als Operatoren werden boolesche Operatoren, (kontinuierlich zeitbeschränkte) temporale Operatoren, sowie Zähloperatoren verwendet.

Die Synthese einer zustandsbasierten Eigenschaft kann in Form eines Automaten erfolgen. Für die Prüfung kann ein markierter Automat genutzt werden, der gültige und ungültige Eigenschaften unterscheidet. Im Falle infiniter zustandsbasierter Eigenschaften ist ggf. das Entscheidungsproblem der zustandsbasierten Eigenschaft praktisch nicht entscheidbar.

4.2.3. Gegenüberstellung Zustandsbasierter und Prozessbasierter Eigenschaften

Prozessbasierte Eigenschaften und zustandsbasierte Eigenschaften besitzen eine Schnittmenge bezüglich ihrer Ausdrucksmächtigkeit. Zustandsbasierte Eigenschaften, die zeitinvariante Eingabe-Ausgabebeziehungen beschreiben, können auch als Prozessbeschreibungen dargestellt werden. Für andere, zeitvariante (zustandsabhängige) Eigenschaften ist eine Darstellung als prozessbasierte Eigenschaft nicht möglich. Prozessbasierte Eigenschaften können als zustandsbasierte Eigenschaften abgebildet werden, falls der Grad der Nebenläufigkeit bekannt ist. Anderenfalls ist eine Abbildung als zustandsbasierte Eigenschaft nicht möglich, da keine endliche Zustandsmenge angegeben werden kann.

Die Synthese der Eigenschaften ist im Falle der prozessbasierten Eigenschaften wegen der einfachen Struktur vergleichsweise einfach. Für zustandsbasierte Eigenschaften ist die Synthese kompliziert, da aufwendige temporale Strukturen entstehen können. In beiden Fällen können die Eigenschaften mittels Automaten zum Monitoring geprüft werden.

4.3. Domänenspezifische Semantik vom Eigenschaften

Nachdem im vorherigen Abschnitt verschiedene Sichtweisen auf Eigenschaften beschrieben wurden, wird in diesem Abschnitt die Nutzung solcher Eigenschaften in verschiedenen Berechnungsdomänen (MoC) erläutert.

4.3.1. Synchronous Data Flow (SDF)

Die *SDF*-Domäne ist besonders geeignet, um Berechnungsvorschriften durch kausale Abhängigkeiten zwischen Daten und Datenoperationen als Datenfluss zu beschreiben. Die Berechnung wird durch die Verfügbarkeit benötigter Daten beschrieben, was als Synchronisationsereignis angesehen werden kann. Dadurch ergibt sich eine virtuelle Taktung des Datenflussmodells. Können die Datenabhängigkeiten als Directed Acyclic Graph (DAG)beschrieben werden, so handelt es sich um ein gültiges Modell. Die Berechnung des Modells erfolgt rein deterministisch, die Abarbeitung kann durch ein statisches Scheduling beschrieben werden [MLD07]. Eine Abwandlung dieses Konzepts zur Beschreibung und Untersuchung von Performanceaspekten in einem

feiner aufgelösten Zeitschema ist unter anderem als Fractional Dataflow [OH04] beschrieben worden.

Als Eigenschaftsbeschreibungssprachen kommen synchrone LTL bzw. CTL in Frage.

4.3.2. Discrete-Event (DE)

Die *Discrete-Event* Domäne nutzt das Paradigma der diskreten Ereignisse, wie es bei der ereignisbasierten, simulativen Performanceanalyse genutzt wird, siehe Abschnitt 21.

Grundidee dabei ist, dass für die globale Betrachtung eines Systems von bestimmten lokalen Entscheidungsfindungsprozessen abstrahiert wird und nur das statistische Verhalten des zu modellierenden Prozesses in die Simulation eingeht. Dabei wird ein kontinuierliches Zeitmodell genutzt, bei dem jedoch nur eine endliche Menge von Zeitpunkten tatsächlich referenziert wird.

In einer globalen Warteschlange werden die zukünftig zu bearbeitenden Ereignisse notiert, die entsprechend der Zeitstempel sortiert sind. Ereignisse, die den gleichen Zeitstempel besitzen, werden entweder zufällig, oder nach einem bestimmten Schema abgelegt (z.B. FIFO, LIFO) [MLD07, BCNN01]. Je nach Modell kann die Reihenfolge der Abarbeitung die Ergebnisse beeinflussen, falls sich dadurch die Reihenfolge von Zugriffen auf lokale Zustandsvariablen ändert [JS05b]. Diese Eigenschaft ist auf die asynchrone Modellsemantik zurückzuführen. Je nach Realisierung des Modells gibt es auch spezielle Optionen, um eine bestimmte Reihenfolge bei der Abarbeitung lokal gleichzeitiger stattfindender Ereignisse zu gewährleisten.

Für die Eigenschaftsbeschreibung sind zeitlich-logische Relationen zwischen Ereignissen von Bedeutung. Die Referenzierung von Ereignissen erfolgt dabei durch den Typ und partielle Daten. Bei den Relationen ist zum einen die zeitliche Relation, teilweise auch der kausale Zusammenhang von Bedeutung, wobei sich die kausalen Relationen nicht allein aus dem zeitlichen Verlauf rekonstruieren lassen. Ferner sind auch Zustände als Nebenbedingung zu berücksichtigen, wobei die Zustandsbetrachtung jeweils vor und nach einem Ereignis durchgeführt werden kann. Im Unterschied zur synchronen Modellsemantik ist der Zugriff auf globale Zustände während der Verarbeitung eines Ereignisses exklusiv. Das Oszillationsparadoxon [JS05b], welches in Hardwarebeschreibungssprachen auftreten kann, wird dabei vermieden.

4.3.3. Continuous Time Discrete Event (CTDE)

Die CTDE Domäne wird dazu verwendet, kontinuierliches (nichtlineares) Verhalten numerisch abzubilden und mit diskreten Verhaltensmodellen zu koordinieren, weshalb es sich um eine hybride Domäne handelt. Dazu wird die Struktur eines Systems als Differentialgleichungssystem, Ordinary Differential Equation (ODE) System modelliert. Dieses Modell kann anschließend numerisch simuliert werden und erlaubt die Triggerung herausgehobener Zeitpunkte, wie z.B. dem Nulldurchgang von Signalen, als Ereignisse. Die Besonderheit liegt in der Koordinierung von kontinuierlichen und diskreten Ereignissen. Für iterative Integrationsverfahren mit adapti-

ver Schrittweite ist eine bidirektionale Variierbarkeit von Zeitpunkten erforderlich. Für diskrete Ereignisse wird dagegen ein (schwach) monotones voranschreiten von Zeit benötigt.

Für die Eigenschaftsbeschreibung wird meist auf zeitliche Relationen zwischen hervorgehobenen Signalzuständen zurückgegriffen. Dies kann entweder als Signalzustand im Sinne einer (möglicherweise nichtlinearen) Diskretisierung [NMP+06] erfolgen oder aber durch Ereignisse beschrieben werden. Die Relationen beschreiben dabei in der Regel die Zeitdauer als Intervall und erlauben eine Verknüpfung mittels temporaler Operatoren.

4.3.4. Finite State Machine (FSM)

Die FSM Domäne wird dazu verwendet, zustandsabhängiges Verhalten formal zu definieren. Prinzipiell lassen sich die synchrone und die asynchrone Semantik unterscheiden. Bei synchronen Finite State Machine (FSM) erfolgen sämtliche Zustandsübergänge paralleler Automaten gleichzeitig, inklusive der Eigentransitionen. Dabei erfolgt je Automat ein Zustandsübergang je Takt. Bei asynchronen FSM erfolgt der Zustandsübergang ereignisgesteuert für jede FSM separat, wobei je Ereignis einer FSM ein stabiler Folgezustand durch eine Abfolge endlich vieler Zustandsübergänge ermittelt wird.

In der FSM Domäne wird auf bekannte Formalismen aus dem Bereich der ereignisorientierten Modellierung zurückgegriffen. Dies sind (bei der asynchronen Semantik) Harel-Charts [Har87] bzw. UML-Statecharts [OMG04]), wobei *Ereignis* (Event), *Bedingung* (Condition) und *Aktion* (Action) an den Zustandsübergängen (Transitionen) notiert werden und Zustände *entry-, do-* sowie *exit*-Aktionen besitzen. Bei der datenflussorientierten Modellierung (synchrone Semantik) die Semantik mit den Automaten der Hardwarebeschreibungssprachen [RS03] vergleichbar, wobei Ereignis und Aktion an den Kannten dort nicht sinnvoll sind. Stattdessen wird nur die *do-Aktion* unterstützt, eine Taktung kann durch Abfrage eines Events erfolgen.

Die Semantik der FSM-Modelle ist also je nach einbettender Modellierungsdomäne unterschiedlich. Als Besonderheit kann ein FSM-Modell nicht selbständig existieren, auch die direkte Einbettung in die CTDE-Domäne ist nicht erlaubt.

4.3.5. Heterogene Modelle mit mehreren Modellierungsdomänen

Die verschiedenen Eigenschaften der Modellierungsdomänen bezüglich Zeitmodell und Zustandsmodellierung sind in Tabelle 4.1 dargestellt.

Man erkennt dabei deutlich die Unterscheidung in synchrone und asynchrone Modellierungsdomänen. Prinzipiell lassen sich die Domänen fast beliebig in einander einbetten, mit der Ausnahme dass die CTDE Domäne nur als Top-Level Domäne korrekt funktioniert und nur die DE-Domäne direkt einbetten kann. Bei einer Einbettung ist weiterhin zu berücksichtigen, dass die eingebettete Domäne die Einschränkungen des Zeitmodells der einbettenden Domäne erbt. Bettet man also ein DE-Modell in ein SDF-Modell ein, so arbeitet dieses anschließend synchron.

Domäne	Zeitmodell	Semantik	Zustandsbegriff
SDF	abgezählt	synchron	Datenmenge
DE	diskretisiert kontinuierlich	asynchron	Ereignismenge + Daten
CTDE	diskretisiert kontinuierlich	asynchron	Signalzustand
FSM (SDF)	abgezählt	synchron	Zustandsmenge + Daten
FSM (DE)	diskretisiert kontinuierlich	asynchron	Zustandsmenge + Ereignis + Daten

Tabelle 4.1.: Übersicht über die Domänen

4.3.6. Interpretation von Systemmodellen

Die Interpretation von Modellen hängt im Wesentlichen von deren Modellsemantik ab. Tabelle 4.2 zeigt eine Übersicht von Modellierungswerkzeugen und der benutzten Semantik als Berechnungsdomäne bzw. MoC. Im Gegensatz zu MLDesigner erfolgt die Unterscheidung in Berechnungsdomänen meist nicht explizit, sondern verbirgt sich in der Arbeitsweise der jeweils verwendeten Modellierungs- und Simulationsumgebung. Die in Klammern notierten Berechnungsdomänen werden entweder nicht direkt unterstützt, lassen sich aber leicht nachbilden (SDF, FSM), oder sie finden nur selten Verwendung, weil sie mit Einschränkungen verbunden sind (DE). Die Domäne BDF stellt eine Verallgemeinerung/Erweiterung der Datenflussdomäne SDF dar, die die Modellierung dynamischer Abhängigkeiten erlaubt, aber dennoch eine statische Gültigkeitsprüfung ermöglicht, siehe auch Abbildung 4.4.

Modellierungswerkzeug	Modell-Domänen
MLDesigner	DE, SDF, BDF, FSM, CTDE
SystemC	DE, (SDF), CTDE, (FSM)
UML	DE, FSM
Matlab/Simulink	BDF, CTDE, FSM, (DE)
Labview	BDF, (CTDE), FSM, (DE)

Tabelle 4.2.: Semantik von Modellierungswerkzeugen

MLDesigner wird hier als Referenzsystem betrachtet. es werden unter andrem die Domänen DE, SDF, BDF, FSM und CTDE direkt unterstützt [MLD07]. Abbildung 4.4 zeigt einen Teil der vom MLDesigner unterstützten Berechnungsdomänen. Wichtig ist, dass die Nutzung der Domänen vom Simulationskernel gesteuert wird, wodurch prinzipiell eine beliebige Kombination von Domänen innerhalb eines Modells möglich ist[19]. Die Datenflussdomänen (SDF, DDF und BDF) stehen in einer Teilmengenbeziehung. Eine Sonderstellung nehmen die Codegenerierungsdomänen ein, die hier aber nicht weiter betrachtet werden.

SystemC unterstützt eine eventorientierte Modellierung, bei der Prozesse aus Signale(Events) warten können oder durch Signale(Events) getriggert werden (DE). Erfolgt die Modellierung

[19] Es gibt einige Einschränkungen, etwa dass die CTDE-Domäne eine System-Level Domäne ist. Zudem ist nicht jede Kombination sinnvoll.

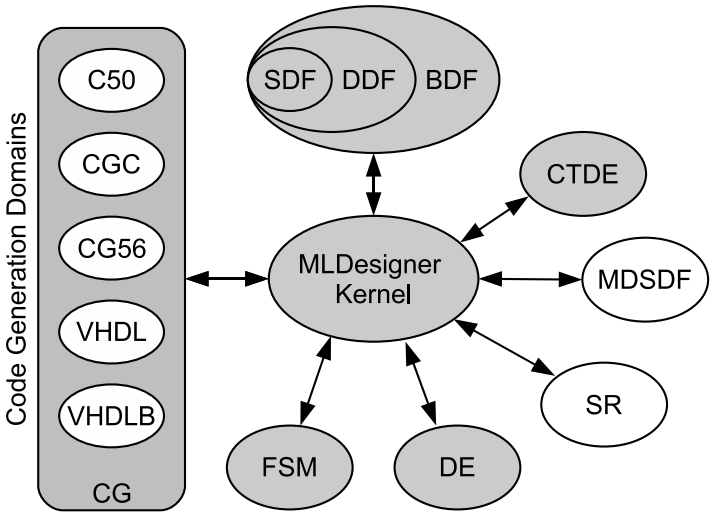


Abbildung 4.4.: MLDesigner Domänen nach [MLD07]

komplett signalsensitiv, so kann das Modell auch als Datenfluss aufgefasst werden (SDF). Endliche Automaten werden zwar nicht als eigenes Modellkonstrukt realisiert, lassen sich aber dennoch abbilden (FSM). Durch Erweiterungen für analoge und Mixed-Signal Simulationen ist die Modellierung von Differentialgleichungssystemen möglich [VGE05, OSC08].

Die Verhaltensdiagramme der UML erlauben vor allem die Beschreibung eines Ablaufen auf anhand diskreter Ereignisse (DE) oder endlicher Zustandsautomaten (FSM). Aktivitätsdiagramme können auch kontinuierliche Informationsflüsse beschreiben (Datenfluss), gelten im Allgemeinen aber als nicht vollständig simulierbar [Stö04].

Matlab/Simulink wird in der Regel dazu verwendet, eine Berechnung bzw. Simulation als Datenfluss zu beschreiben. Die Berechnungen können dabei bedingt sein, entsprechen also der BDF-Domäne. Erfolgt die Modellierung nicht zeitdiskret, sondern durch Differentialgleichungen und numerische Solver, so entspricht sie der CTDE-Domäne. Mit der Erweiterung *Stateflow* kann die Modellierung auch per endlicher Zustandsmaschine (FSM) erfolgen. Durch die Verwendung spezieller Signale ist auch eine eventbasierte Modellierung (DE) realisierbar [The07].

Labview nutzt zwar im Vergleich zu Matlab/Simulink ein anderes Konzept, ist jedoch prinzipiell vergleichbar. Grundlage ist auch hier die Verwendung eines bedingten Datenflusskonzeptes (BDF). Die endlichen Zustandsautomaten (FSM), sind dabei als spezielle Konstrukte realisierbar. Durch spezielle Bibliotheken, sogenannten Toolboxen, wird die Modellierung und Analyse von Differentialgleichungssystemen ermöglicht (CTDE). Die Modellierung mit eventartigen Signalen (DE) ist ebenfalls möglich [Nat04].

4.4. Konzept zur Organisation von heterogenen Eigenschaften

Die Modellierungskonzepte der verschiedenen verwendeten Modellierungsdomänen besitzen Teilmengenbeziehungen bezüglich der verwendeten Semantik. Ist eine Domäne A in einer Domäne B enthalten, bedeutet dies, dass ein Modell der Domäne A in ein Modell einer Domäne B eingebettet werden darf. Da sich die Eigenschaftsspezifikation für Modelle einer Domäne an der zugehörigen Semantik orientiert, sollte auch eine Einbettung von Eigenschaften der Domäne A in die Domäne B möglich sein. Folgende Einbettungsbeziehungen zwischen den Domänen SDF, DE, CTDE, FSM sind allgemein anerkannt[20] [MLD07, OSC07] :

- $DE \subset CTDE$
- $SDF \subset DE$
- $FSM \subset DE$
- $FSM \subset SDF$

Die hier verwendete Ordnung, siehe auch Abbildung 4.5, der Domänen berücksichtigt ebenfalls der Mächtigkeit der Zeitmodellierung und wird daher auch bei anderen heterogenen Modellierungsansätzen verwendet [VGE05, OSC08]. Man erkennt, dass die DE-Domäne zur Synchronisation verwendet wird, und somit eine Sonderrolle einnimmt. Dies ist u.a. dem Umstand geschuldet, dass die CTDE-Domäne der Forderung nach Kausalität[21] nicht gerecht wird, und somit statt der Einbettung in der Regel eine Synchronisation realisiert wird. Die hier betrachtete FSM-Domäne besitzt andererseits kein eigenes Zeitmodell (wohl aber die Forderung nach Kausalität), so dass die Semantik der einbettenden Domäne geerbt wird.

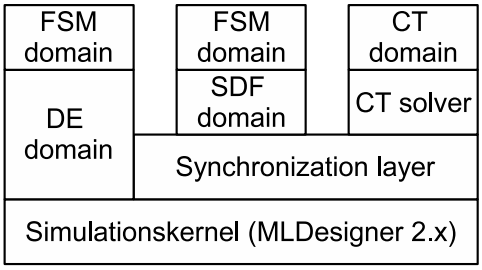


Abbildung 4.5.: Hierarchische Ordnung von Modellierungsdomänen nach [VGE05]

Die verschiedenen Domänen repräsentieren verschiedene Sichten auf die Funktionalität und Semantik von Modellen. Dementsprechend sind unterschiedliche Eigenschaftsbeschreibungen sinnvoll. Um eine Integration von Eigenschaften in heterogenen Modellen beschreiben zu können, ist die Transformation von Eigenschaftsmodellen sinnvoll. Soll beispielsweise eine Eigenschaft der Domäne CTDE als Vorbedingung und eine Eigenschaft der Domäne DE als Nachbe-

[20] Zwar bestehen auch weitere Möglichkeiten der Einbettung, z.B. $DE \subset SDF$, allerdings sind diese hier nicht sinnvoll.

[21] Bedingt durch die notwendige Vorwärts- und Rückwärsintegration bei iterativen numerischen Solvern

dingung in einer heterogenen Eigenschaftsbeschreibung verwendet werden, so ist eine Transformation in eine gemeinsame Beschreibung sinnvoll. Dazu ist es erforderlich, dass zur Integration ein Modell genutzt wird, das als Zeitmodell und Semantik die Obermenge der Domänen unterstützt. Dies ist in der DE-Domäne erfüllt, die ein diskretisiert-kontinuierliches Zeitmodell[22] in einer asynchronen Semantik unterstützt, und daher auch zur Synchronisation, siehe Abbildung 4.5 genutzt wird.

Grundidee ist es, Eigenschaftsbeschreibungen für heterogene Systeme zu ermöglichen und für die Verifikation nutzbar zu machen. Dabei ist es erforderlich, die Eigenheiten der einzelnen Simulationsdomänen zu berücksichtigen.

4.5. Sprachdefinition heterogener temporaler Eigenschaften

Für die Assertion basierter Verifikation eignen sich vor allem temporale Eigenschaften, da diese keine strukturellen Informationen des zu prüfenden Modells benötigen. Weiterhin existieren bereits vielfältige Ansätze zur Beschreibung von Eigenschaften unterschiedlicher Domänen. Allerdings existiert noch kein Ansatz, der wirklich für die Beschreibung von Eigenschaften in komplexen heterogenen Systemen geeignet ist. Ziel dieses Abschnittes ist daher die Entwicklung einer Beschreibungssprache, die auf temporaler Logik basiert und die Besonderheiten heterogener Modelle berücksichtigt.

Das Zeitmodell sowie dessen Semantik müssen die Eigenschaften der jeweiligen Modellierungsdomäne berücksichtigen. Es werden für die Sprachdefinition ein Modell für abzählbare (enumerierte) Zeit (nächster Zustand) und ein Modell für quasikontinuierliche Zeit benötigt. Weiterhin müssen Wechselwirkungen zwischen beiden Zeitmodellen besonders beachtet werden. Die Sprachdefinition unterteilt sich nach dem Vorbild PSL in mehrere Ebenen.

- Der *Verification Layer* spielt im betrachteten Fall nur eine untergeordnete Rolle, da der Zweck der Eigenschaftsdefinition, die Assertion-basierte Prüfung feststeht.
- Im *Modeling Layer* wird die Abstraktion von Signalen zu booleschen Signalen beschrieben, die als elementare Aussagen der weiteren Ebenen dienen. Signale und Wertedefinitionen können je nach Modellierungsdomäne unterschiedlich realisiert werden.
- Im *Boolean Layer* werden boolesche Verknüpfungen zwischen Aussagen realisiert.
- Im *Temporal Layer* werden zeitbeschränkte temporale Operatoren definiert. Hierbei müssen Wechselwirkungen der Zeitmodelle berücksichtigt werden.

Die nachfolgenden Definitionen und Erläuterungen in diesem Abschnitt bauen auf den Arbeiten [RHKR00, NMP+06, AFH96] auf. Die im Abschnitt 4.5.2 dargestellte Definition erstreckt

[22] Das Zeitmodell an sich ist kontinuierlich (Dense Time) allerdings werden nur einzelne (diskrete) Zeitpunkte tatsächlich für die Simulation genutzt.

sich auf die abstrakte Syntax und die Semantik der Sprache. Die konkrete Syntax der Realisierung gemäß Kapitel 6 ist im Anhang A.2 zu finden.

4.5.1. Eventbasiertes Zeitmodell

Die Semantik von Realzeitlogiken ist für Signalzustände definiert, vgl. Abschnitt 2.6. Dieser Definition zufolge ist jedem Zeitpunkt im betrachteten Ausführungspfad ein (Signal-)Zustand zugeordnet $s : \mathbb{Q}_+ \rightarrow 2^P$.

Für die Betrachtung von Ereignissen ist ein wertebehafteter Zustand nicht ausreichend. Es muss unterschieden werden, ob ein Ereignis zu einem Zeitpunkt anliegt oder nicht. Ereignisse können zusätzlich ein transientes Verhalten aufweisen, d.h. es können mehrere Ereignisse im selben Zeitpunkt auftreten, siehe Abschnitt 2.3.6 und Abschnitt 4.3.2. Da die Ereignisse in der Simulation eine Reihenfolge besitzen, könnte man diese als geordnete Liste modellieren.[23] Die Analyse von Beziehungen zwischen transienten Ereignissen wird in dieser Arbeit nicht weiter betrachtet, ein möglicher Ansatz, der dies berücksichtigt, wird in [PF07] dargestellt.

Die Realzeitlogiken sind für die Beschreibung dieses transienten Verhaltens zwar nicht ausgelegt, eine Erweiterung der temporalen Operatoren, speziell des F-Operators und U-Operators, wäre jedoch möglich.

Ereignisse - transiente Ereignisse ausgenommen - können nicht zeitlich unendlich dicht beieinander liegen. Daher existiert zwischen je zwei nicht transienten, aufeinander folgenden Ereignissen e_1 zum Zeitpunkt t_1 und e_2 zum Zeitpunkt t_2 ein Intervall $(t_1, t_2)|t_2 > t_1 \geq 0 \in \mathbb{Q}_+$, in dem kein Ereignis stattfindet. Eine gültige Diskretisierung (Interval-Covering $\mathcal{I}$) erreicht man demnach, indem man den quasikontinuierlichen Definitionsbereich abwechselnd in Punktintervalle und beidseitig offene Intervalle unterteilt. Einen Signalvektor mit Events zu n aufeinanderfolgenden Zeitpunkten $t_1, \ldots, t_n$ ergibt somit ein Interval-Covering

$$\begin{aligned}\mathcal{I} =& [t_0, t_0], (t_0, t_1), [t_1, t_1], \ldots, (t_n, \infty), t_0 = 0,\\ & \text{falls } t_1 > 0 \text{ anderenfalls ist } t_1 = 0 \text{ und}\\ \mathcal{I} =& [t_1, t_1], (t_1, t_2), [t_2, t_2], \ldots, (t_n, \infty).\end{aligned}$$

$\mathcal{I}$ stellt demnach ein gültiges Interval-Covering dar, mit dem zeitkontinuierliche boolesche Projektionen von n-dimensionalen Signalvektoren auf n-dimensionale ω-Sequenzen der Form $\xi : \mathbb{N} \rightarrow \mathbb{B}^n$ abgebildet werden können. Weiterhin lässt sich das Interval-Covering als Diskretsierungsfunktion $\sigma : \mathbb{Q}_+ \rightarrow \mathbb{N}$ beschreiben.

In Abbildung 4.6 sind zwei unterschiedliche Signaltypen als Beispiel dargestellt. Das boolesche Signal $p(t)$ repräsentiert die boolesche Projektion eines zeitkontinuierlichen Signals, die

[23] Der Discrete-Event Formalismus schreibt hier eine unterscheidbare Reihenfolge vor, vgl. Abschnitt 2.3. Die Semantik ist den Deltazyklen in Hardware Description Language (HDL) ähnlich.

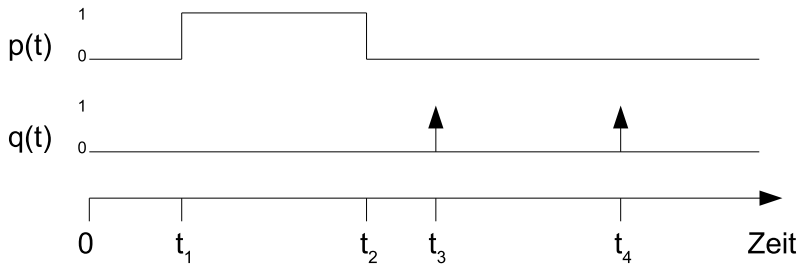


Abbildung 4.6.: Zeit- und Signalmodell für XFLTL

im Intervall $[t_1, t_2)$ den Wert 1 (*true*) und sonst den Wert 0 (*false*) besitzt. Das Eventsignal $q(t)$ repräsentiert die boolesche Projektion eines Eventsignals, die zu den Zeitpunkten t_3 und t_4 ein Event aufweist. Eine gültige Diskretisierung des Signalvektors $(p, q)^T$ wäre

$$\mathcal{I}_t = [0,0], (0,t_1), [t_1,t_1], (t_1,t_2), [t_2,t_2], (t_2,t_3), [t_3,t_3], (t_3,t_4), [t_4,t_4], (t_4,\infty).$$

4.5.2. Definition von XFLTL

Es wird eine auf Events basierende Realzeitlogik namens e*X*tended *F*inite *L*inear *T*emporal *L*ogic (XFLTL) benutzt, die durch die folgende Grammatik beschrieben werden kann:

$$\varphi ::= p \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \varphi_1 U_I \varphi_2 \tag{4.1}$$

Dabei ist p eine Aussage und I ein möglicherweise singuläres[24] Intervall, dessen Grenzen rationale Zahlen sind. Im Folgenden werden beidseitig geschlossene Intervalle verwendet, obwohl auch die Verwendung einseitig und beidseitig offener Intervallgrenzen möglich ist. Die Erfüllung einer XFLTL-Formel *varphi* im Zeitpunkt $t \in \mathbb{Q}^+$ einer Sequenz von Signalzuständen τ, wird durch $(\tau, t) \vDash \varphi$ entsprechend folgender Definition geschrieben:

$$(\tau, t) \vDash p \text{ iff } p \in \tau(t) \tag{4.2}$$

$$(\tau, t) \vDash \varphi_1 \wedge \varphi_2 \text{ iff } (\tau, t) \vDash \varphi_1 \text{ and } (\tau, t) \vDash \varphi_2 \tag{4.3}$$

$$(\tau, t) \vDash \neg\varphi \text{ iff } (\tau, t) \nvDash \varphi \tag{4.4}$$

$$\begin{aligned}(\tau, t) \vDash \varphi_1 U_I \varphi_2 \text{ iff } &\exists t' \in (t + I) \wedge (\tau, t') \vDash \varphi_2 \text{ and} \\ &\forall t'' \in (t, t'), (\tau, t'') \vDash \varphi_1\end{aligned} \tag{4.5}$$

Diese Definition ist ähnlich zur *Metric Interval Temporal Logic* (MITL) und *Event Clock Temporal Logic* (ECTL) [HRS99]. Die Nutzung rationaler Zahlen zeigt den Hauptunterschied

[24] Punktintervall der Länge 0.

zu Methoden, die auf enumerierter Zeit (natürliche Zahlen) basieren.[25] Aus den definierten Basisoperatoren lassen sich weitere Operatoren für eine vereinfachte Syntax ableiten, etwa die zeitbeschränkten temporalen Operatoren F (Finally) und G (Generally).

Bei der Anwendung von Extended Finite Linear Temporal Logic (XFLTL) für Event-basierte Signale kann die Menge von Aussagen P auf elementare Aussagen über einzelne Events e zurückgeführt werden. Beispielsweise beschreibt die Aussage $(\tau, t) \vDash e$, dass ein Event beim Eventsignal e zum Zeitpunkt t auftritt. Aus der Dichtheit von $\mathbb{Q}$ und der Singularität des Events können wir schließen, dass $(\tau, t) \vDash e \implies \exists \epsilon > 0 \in \mathbb{Q} \mid (\tau, t \pm \epsilon) \nvDash e$ gelten muss. Für Zeitintervall-beschränkte temporale Eigenschaften ohne singuläres Intervall können folgende Vereinfachungen abgeleitet werden:

$$G_{[a,b]} e \vDash \text{FALSE} \tag{4.6}$$

$$F_{[a,b]} \neg e \vDash \text{TRUE} \tag{4.7}$$

Durch diese Vereinfachungen können Zeitintervall-beschränkte Eigenschaften entschieden werden, indem nur diskrete Events unter den Gesichtspunkten von Ursache und Wirkung betrachtet werden.

Abbildung 4.7 zeigt die Zeitreferenzen im Sinne relevanter Intervalle δ von Teilformeln (Operatoren) der Formel $\psi = G_{[4,8]}((F_{[2,3]}x) \vee (F_{[3,6]}y))$, beginnend mit dem Zeitpunkt 0. Der obere Teil zeigt den Pfad von außen (Gesamtformel) nach innen (Teilformel) und zeigt die potentiell relevanten Zeitpunkte, zu denen eine Änderung des betreffenden Operators stattfinden kann, sowie die Abhängigkeit zu untergeordneten Operatoren. Im unteren Teil sind die potentiellen Auswirkungen von Events zu den Operatoren, also von innen nach außen, dargestellt. Es wird ersichtlich, dass die relativen Zeitreferenzen der Teilformeln erhöhen und damit die potentiell wichtigen Zeitintervalle im Vergleich zur Intervall der Operatoren größer werden, was im Folgenden als *Widening (Auffächern)* bezeichnet wird. Zusätzlich kann eine Menge zeitlich unterschiedlicher Events die Eigenschaft zu einem einzelnen Zeitpunkt beeinflussen, was im Folgenden als *Narrowing (Einengen)* bezeichnet wird. *Widening* und *Narrowing* wird auch als *Relaxing Punctuality* bezeichnet [HRS99].

Durch *Narrowing* wird die Wirkung auftretender Events zeitlich beschränkt, so dass eine temporale Eigenschaft prinzipiell auch iterativ beschrieben werden kann. *Widening* führt zu einer zeitlichen Überlagerung von Ursache-Wirkung Beziehungen, so dass verschiedene Ursachen (Events) einer Wirkung (Operatorzustand) zu unterschiedlichen Zeiten auftreten. Es besteht also eine relative Fehlausrichtung bezüglich der Zeit. Zusammen ergibt sich die Fragestellung nach der korrekten minimalen Partitionierung der Zeit in Intervalle, um ein korrektes Prüfen

[25] Es ist einfach zu zeigen, dass Formel mit natürlichen Zahlen und rationalen Zahlen die selbe Ausdrucksstärke besitzen. Allerdings sind Axiome, die auf Operatoren, wie *Next Previous* basieren nicht für ein quasikontinuierliches Zeitmodell geeignet.

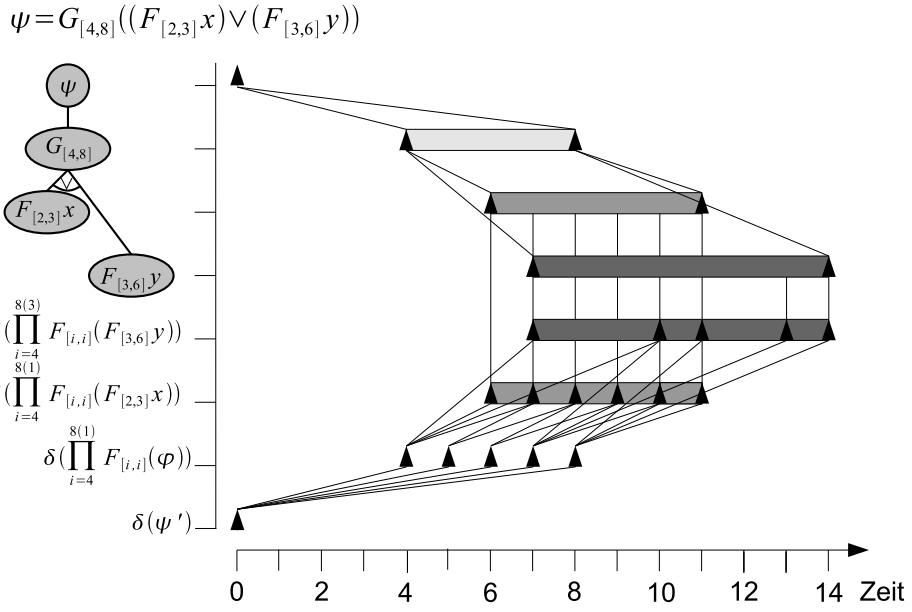


Abbildung 4.7.: Widening und Narrowing von Intervallen

der Eigenschaften zu ermöglichen. Die Betrachtungen zur Partitionierung werden zusammen mit der jeweiligen Prüfmethodik in Abschnitt 6.2 und Abschnitt 6.3 betrachtet.

4.5.3. Abstraktion heterogener Signale mittels Events

Analoge und Hybride Signale werden bisher durch Realzeitlogiken spezifiziert, deren Aussagen auf Zuständen und deren Änderungen basieren. Dadurch entsteht das Problem der zeitlichen Diskretisierung der Signale.[26] Die hier vorgeschlagene Lösung basiert auf der Umformung solcher Eigenschaften in Event-basierte Spezifikationen. Durch die Events selbst werden dann die Änderungen propagiert, wodurch automatisch eine korrekte Diskretisierung erreicht wird.[27]

Dazu werden folgende Annahmen getroffen. Eine Zustandsänderung eines Signals in einer analogen, digitalen oder Mixed-Signal Umgebung kann als einzelnes Event modelliert werden. Aus Sicht der Eigenschaft kann eine Sequenz von Zuständen auch als Folge von Events beginnend mit einem Anfangszustand betrachtet werden.

Abbildung 4.8 zeigt die erforderliche Konvertierung zustandsbasierter analoger und digitaler Signale. Das *AMS Modell* ist ein heterogenes Modell, das Module unterschiedlicher Modellierungsdomänen enthält, wohingegen der zur Verifikation genutzte *Assertion Checker* ein Discrete-Event Modell benutzt. Daher müssen die analogen und digitalen Signale aus dem AMS Modell in Eventsignale konvertiert werden. Dies kann durch zusätzliche Modellelemente oder automatische Konvertierungen an Domänengrenzen erreicht werden.

[26] Hierbei müssen in der Regel sämtliche Eingangssignale in der selben Weise diskretisiert werden.
[27] In der Regel ist keine weitere Diskretisierung der Eingangssignale erforderlich.

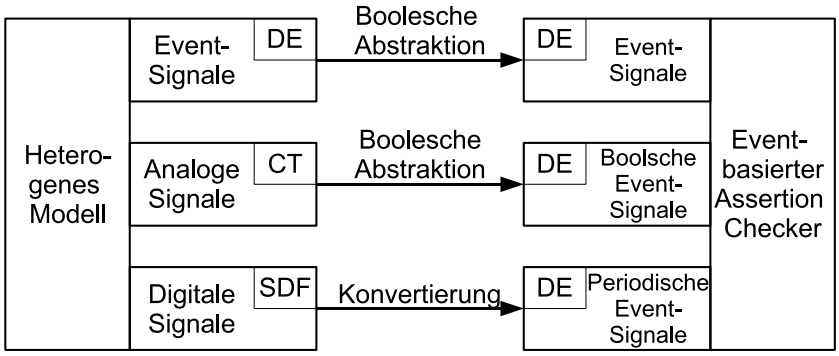


Abbildung 4.8.: Event basiertes Assertion Checking

Um ein analoges Signal durch eine temporale Eigenschaft beschreiben zu können, muss es zu einem booleschen Zustand mit endlicher zeitlicher Variabilität abstrahiert werden[28] [MNPB06]. Dies kann mathematisch durch eine Funktion $\pi : \mathbb{R}^n \rightarrow \mathbb{B}$ ausgedrückt werden. Für die praktische Realisierung kann eine komplexe arithmetische Funktion sowie ein *zero-crossing* Block für den Solver der analogen Berechnungsdomäne genutzt werden. Der zero-crossing Block sorgt dafür, dass die booleschen Übergänge durch den Solver exakt berechnet werden und ein Eventsignal resultiert. Eine nachgelagerte (offline) Berechnung kann dies in der Regel nicht leisten.

Digitale Signale ändern ihren Zustand synchron zu einem Clock-Signal. So wird meist von exakten Zeiten abstrahiert und stattdessen ein n-tes Ereignis des Clock-Signals referenziert. Um korrekte Aussagen bezüglich der Interaktion in einem Mixed-Signal Modell treffen zu können, ist es erforderlich, das referenzierte Clock-Signal mit zu betrachten. Eine Möglichkeit besteht darin, für ein betreffendes digitales Signal jeweils ein Event pro Event des assoziierten Clock-Signals zu erzeugen, statt nur die Änderungen durch ein Event zu signalisieren. Bei synchronen Signalen, wie beispielsweise in der SDF-Domäne, geschieht dies automatisch durch eine Signalkonvertierung. Im Folgenden betrachten wir nur boolesche digitale Signale, die Betrachtungen können auch auf komplexe Signale, wie Vektoren übertragen werden.

In den folgenden Abschnitten wird die Ersetzung von analoger, digitaler und von Mixed-Signal Assertions durch Event-basierte Assertion betrachtet.

4.5.4. Interpretation analoger Eigenschaften in XFLTL

Analoge Assertions nutzen eine temporale Logik mit booleschen Interpretationen n-dimensionaler analoger Signale als elementare Aussagen. Der Zustand dieser Aussagen wird durch reellwertig zeitbeschränkte lineare temporale Logiken beschrieben. Dies ermöglicht die Ersetzung analoger Assertions durch Event-basierte Assertions, indem Zustandsänderungen durch Ereignisse beschrieben werden.

[28] Diese Eigenschaft wird auch als *non-zeno* bezeichnet, und bedeutet, dass keine unendlich oszillierenden Signale abgebildet werden können, da diese aus Gründen der Komplexität nicht berechenbar sind.

Es werden folgende Schreibweisen benutzt. Ist $s(t)$ ein analoges Signal, das durch die Funktion π abstrahiert wird, dann ist $e_s = \pi(s(t))$ das boolesche Eventsignal. Dann ist $e{\downarrow_s}$ das Event, welches die Änderung von *true* nach *false* beschreibt und $e{\uparrow_s}$ ist das Event, das die Änderung des Signalzustandes von *false* nach *true* beschreibt. Der initiale Zustand des Signals $s(t)$ zu einem Zeitpunkt t_0 ergibt sich aus dem unmittelbar vorangegangenen Event, und wird mit $e_s(t_0)$ bezeichnet.

Nachfolgend sind zwei Beispiele für die Event-basierte Beschreibung analoger Assertions notiert.

$$\psi = G_{[a,b]}\pi(s) \Longleftrightarrow \psi' = e_s(a) \wedge \neg F_{[a,b]}e{\downarrow_s}$$

$$\psi = F_{[a,b]}\pi(s) \Longleftrightarrow \psi' = e_s(a) \vee F_{[a,b]}e{\uparrow_s}$$

4.5.5. Interpretation digitaler Eigenschaften in XFLTL

Der Zustand rein digitaler Signale wird synchron zu einem Clock-Signal beschrieben, weshalb Zeit durch Clock Ticks beschrieben werden kann. Daher kann in einem Event-basierten System Zeit als Anzahl von Events notiert werden, siehe Abschnitt 4.5.3. Innerhalb einer Mixed-Signal Echtzeitumgebung wird zwar oft eine konstante Periodendauer angenommen, prinzipiell ist aber auch eine schwankende Periodendauer möglich, was einen deutlichen Einfluss auf ein Mixed-Signal System haben kann.

Dieses Problem kann gelöst werden, indem ein Zählmechanismus für die betreffenden Signale genutzt wird, anstatt Zeit direkt über die Periodendauer zu messen. Für ein gegebenes digitales Signal $d(t)$, wird eine Aktualisierung mit der zugehörigen Clock als Eventsignal e_d notiert. Dabei beschreibt $e{\downarrow_d}$ ein Event, das den Zustandswechsel von *true* nach *false* beschreibt und $e{\uparrow_d}$ das Event, das den Zustandswechsel von *false* nach *true* beschreibt. Der Zustand des digitalen Signals $d(t)$ zu einem festgelegten Zeitpunkt t_0 ergibt sich aus dem unmittelbar vorangegangenen Event und wird als $e_d(t_0)$ beschrieben.

Der Ausdruck $O_{I@e_d}$ beschreibt, dass der temporale Operator O durch das ganzzahlige Zeitintervall I beschrieben wird, welches mit der Clock von e_d assoziiert ist. Der Unterschied zu den bisher betrachteten temporalen Operatoren besteht also darin, dass der Zeitmechanismus auf ganzzahligen Werten beruht und von Events eines Eventsignals abhängt. Sind reellwertige zeitliche Relationen zu berücksichtigen, so können diese auf das assoziierte Eventsignal zu rückgeführt werden.

Nachfolgend sind zwei einfache Beispiele für die ersatzweise Beschreibung digitaler Assertions durch Event-basierte Assertions angegeben, wobei die mit $d(t)$ assoziierte Clock als Zeitbasis genutzt wird.

$$\psi = G_{[a,b]}d \Longleftrightarrow \psi' = G_{[a,b]@e_d}e{\downarrow_d}$$

$$\psi = F_{[a,b]}d \Longleftrightarrow \psi' = F_{[a,b]@e_d}e{\uparrow_d}$$

4.5.6. Heterogene Eigenschaften in XFLTL

Werden analoge und digitale Assertions verbunden, so muss der Zeitbezug zwischen beiden berücksichtigt werden. Der einfachste Weg besteht darin, nur die Suffix-Implikation[29] zu verwenden, deren Semantik in digitalen Systemen beschreibt, dass nach der Erfüllung der Vorbedingung im nächsten Taktzyklus die Nachbedingung erfüllt wird. Falls die Vorbedingung analog und die Nachbedingung digital ist, so sollte der digitale Teil von einem Temporalen Operator, beispielsweise $F_{[0,0]@clock}$, umschlossen sein, um einen impliziten Zeitbezug herzustellen. Ist die Nachbedingung analog, so sollte diese ebenfalls mit einem temporalen Operator umschlossen sein.

Angenommen es existieren ein analoges Signal s und ein digitales Signal d. Weiterhin soll immer dann, wenn s für eine längere Zeit als t_1 größer als c_1 ist, das Signal d den Zustand true annehmen. Dies kann zustandsbasiert notiert werden als

$$\phi = (G[0, t_1](s > c_1)) \mapsto d).$$

Durch den Austausch analoger und digitaler Signale s und d durch Eventsignale e_s und e_d kann diese Assertion wie folgt notiert werden:

$$\phi' = (e_s(0) \wedge \neg F_{[0,t_1]} e{\downarrow_s}) \mapsto (F_{[0,0]@e_d} e{\uparrow_d})$$

4.5.7. Zusammenfassung heterogene temporale Eigenschaften

Durch die Verwendung Event-basierter temporaler Eigenschaften besteht die Möglichkeit, eine gemeinsame Grundlage für heterogen zusammengesetzte Eigenschaften zu definieren, welche die natürliche Strukturierung der Berechnungsdomänen nutzt. Dazu wurde zunächst die temporale Logik XFLTLdefiniert, die Aussagen über Events in einem kontinuierlichen Zeitmodell beschreibt.

Für analoge und digitale Signale lassen sich mit Hilfe von Events geeignete Ersatzkonstruktionen definieren. Unter Berücksichtigung des Zeitmodells werden für komplexe temporale Eigenschaften Ersatzkonstruktionen basierend auf XFLTL ermöglicht.

Somit steht ein Mechanismus zur Verfügung, indem heterogene temporale Eigenschaften in einer einheitlichen Semantik definiert werden. Dieser Mechanismus ist weiterhin dazu geeignet, in eine Simulationsumgebung integriert zu werden.

[29] Die Suffix Implikation beschreibt die das zeitliche nacheinanderfolgen zweier Eigenschaften.

4.6. Sprachdefinition nebenläufiger Prozess-basierter Eigenschaften

Für die formale Eigenschaftsprüfung müssen Eigenschaften definiert werden, die einerseits als Zusicherungen und Anforderungen verschiedene Rollen annehmen können, und andererseits eine geeignete Abstraktionssicht bieten. Für die Eigenschaftsbeschreibung zum Zwecke der formalen Verifikation gelten also andere Anforderungen als für die Assertion basierte Verifikation. Da insbesondere Prozess-basierte Spezifikationen eine geeignete Abstraktion für die Betrachtung der Performance von System-Level Modellen bietet, werden diese als Basis gewählt.

Ziel dieses Abschnittes ist daher eine Entwicklung einer Sprache zur Beschreibung von Eigenschaften basierend auf zeitbehafteten Prozess-basierten Spezifikationen, die die Besonderheiten der formalen Verifikation berücksichtigt.

4.6.1. Sichtbarkeit der Eigenschaften

Bei der Verwendung prozessbasierter Eigenschaften innerhalb eines komplexen Modells muss die Sichtbarkeit der Eigenschaften berücksichtigt werden. Geht man von einem Hierarchisch strukturierten Modell aus, so kann eine Eigenschaft entweder einem Modellelement zugeordnet werden, oder sie gilt anderenfalls global. Aus Gründen der Lokalität dürfen Eigenschaften, die einem Modellelement zugeordnet sind, nur die Bestandteile adressieren, die innerhalb des Modellelements auch direkt sichtbar sind.

Betrachtet man ein hierarchisch verfeinertes Modul, dann können sich Zusicherungen auf extern sichtbare Eventsignale beziehen, wohingegen sich Anforderungen sowohl auf extern sichtbare als auch auf interne Eventsignale beziehen können. Die Parametrisierung einer Eigenschaft kann jeweils von den extern sichtbaren Parametern des zugeordneten Moduls abhängen.

4.6.2. Semantik der Prozesskette als Eventfluss

Die kausale Abhängigkeit zwischen konsumierten und emittierten Events eines Prozesses wird im Folgenden als Eventfluss bezeichnet, die als temporale Abhängigkeit beobachtbar ist. Die kausale Abhängigkeit der Events ist dementsprechend als positive Ursache-Wirkungsbeziehung zwischen Events formulierbar (Ursache $\implies F[t_1, t_2]$Wirkung$|t_1 \leq t_2 \in \mathbb{R}^+$) und stellt eine Eigenschaft der betreffenden Modellelemente dar. Folgende Anforderungen werden allgemein an eine generierbare Ursache-Wirkung-Spezifikation gestellt:

- Realisierbarkeit
- Generierbarkeit
- zeitliche Kausalität
- Komponierbarkeit

Für die Erzeugung eines formal korrekten Modells ist die Generierbarkeit der Eigenschaften zu berücksichtigen. So besteht die Gefahr Eigenschaften derart zu spezifizieren, dass diese nicht sinnvoll in der Semantik eines Discrete-Event Modells realisiert werden können, siehe Tabelle 4.3. Eine solche nicht realisierbare Eigenschaft stellt z.B. die Forderung dar, dass zu jedem Zeitpunkt, an dem nicht Event x eintritt, ein Event y auftreten soll. Dementsprechend wäre die Erzeugung unendlich vieler Events des Typs y gefordert. Andererseits existieren Eigenschaftsspezifikationen, die zwar spezifizierbar und realisierbar sind, aber nicht direkt für eine Generierung genutzt werden können. Etwa die Forderung, dass beim Auftreten eines Events x kein Event y auftreten darf. Selbst ein leeres Modell (Event y tritt nie auf) erfüllt diese Eigenschaft. Andererseits ist im Discrete-Event Modell nicht prüfbar, ob zu einem gegebenen Zeitpunkt noch ein Event von Typ x auftreten wird[30], stattdessen können nur Aussagen über die Vergangenheit getroffen werden. Nicht kausal (negativ kausal) bedeutet, dass eine Ursache-Wirkungsbeziehung bezogen auf negierte Events vorliegt, z.B. nicht Events x führt zu nicht Event y. Kehrt man die Implikation um, so ergibt dies die Spezifikation, dass, damit Event y auftreten darf, bereits Event x aufgetreten sein muss, führt also zu einer Abhängigkeit bzw. Beschränkung.

Ursache	Wirkung	Bemerkung
Event x	Event y	positiv kausal
Event x	¬ Event y	Beschränkung
¬ Event x	Event y	ungültig
¬ Event x	¬ Event y	nicht kausal

Tabelle 4.3.: Spezifikation von Eventfluss als Ursache-Wirkung

Die zeitliche Kausalität muss erfüllt sein. Das bedeutet, dass die Generierung eines Events zu einem gegebenen Zeitpunkt nicht von zukünftigen Events abhängen darf, also alle Ursachen spätestens zum Zeitpunkt der Wirkung erfüllt sein müssen.

Sinnvoll ist die Komponierbarkeit verschiedener Eigenschaften. Bei der Komponierbarkeit muss zwischen der Verkettung und der logischen Komposition unterschieden werden. Bei der Verkettung stellt die Wirkung einer Eigenschaft die Ursache einer weiteren Eigenschaft dar. Bei der logischen Komposition existieren verschiedene Alternativen oder zusätzliche Beschränkungen bei der Generierung eines Signals. Werden rein positiv kausale Abhängigkeiten verwendet, so ist die Ursache zur Generierung eines Events als boolesche Verknüpfung von Events in Form einer Konjunktiv Disjunktive Normalform (KDNF) beschreibbar.

Beschränkende Eigenschaften können bei der Generierung genutzt werden, falls die Beobachtung der Ursache in der Vergangenheit liegt. Dann sind die Beschränkungen und Abhängigkei-

[30] Es ist im Allgemeinen vom Simulationswerkzeug abhängig, ob Informationen über ausstehende Events zu bestimmten Zeitpunkten im Modell selbst verarbeitet werden können. Selbst wenn dies der Fall ist, besteht die Möglichkeit, dass das gesuchte Event im Laufe der Simulation erst noch hinzugefügt wird.

ten als zeitliche Bedingung der Beobachtung der Ursache beschreibbar. Die Realisierung einer solchen zeitlichen Bedingung als Nebenbedingung erfordert das Speichern der Beobachtungen und erschwert eine Realisierung.

Es lässt sich also feststellen, dass die Generierung nur für positiv kausale Eigenschaftsspezifikationen effizient möglich ist. Beschränkungen und Abhängigkeiten lassen sich nur realisieren, wenn die Ursachen bereits in der Vergangenheit liegen. Für die Prüfung könnten hingegen auch Beschränkungen und Abhängigkeiten berücksichtigt werden, die den Zeitpunkt der Wirkungsbeschreibung betreffen[31]. Ungültige Spezifikationen können nicht realisiert werden und gelten bei der Prüfung als nicht erfüllbar.

Die Ausdrucksmächtigkeit der Ursache-Wirkung-Spezifikation ist eine Teilmenge von LTL, vergleichbar mit den SERE der *Property Specification Language*, siehe Abschnitt 2.5.3. Die positiv kausalen Eventfluss-Spezifikationen stellen wiederum eine Teilmenge der Eventfluss-Spezifikationen dar. Damit lassen sich die wesentlichen nicht zustandsbehafteten Eventfluss-Abhängigkeiten, wie etwa, *Fork*, *Join*, *Delay*, etc. realisieren. Daher ist es naheliegend, Eventflussspezifikationen auf positiv kausale Eventflussspezifikationen zu beschränken.

Für die Beschreibung des Verhaltens werden daher positiv kausale Spezifikationen des Eventflusses als Zusicherungen und Anforderungen verwendet. Beschränkungen werden dagegen nur für Anforderungen verwendet. Weiterhin lassen sich FSMs als Zusicherungen verwenden.

4.6.3. Aufbau Prozessbasierter Spezifikation

Die Sprachdefinition unterteilt sich nach dem Vorbild PSL in mehrere Ebenen.

Der *Verification Layer* ist für die Beschreibung der Rollen einer Eigenschaftsspezifikation geeignet. Insbesondere können hierbei Anforderungen und Zusicherungen unterschieden werden.

Der *Modeling Layer* definiert die Verknüpfung zwischen Modell und Eigenschaften. Hierbei müssen insbesondere verschieden Konfigurationen von Komponenten berücksichtigt werden.

Durch die Nutzung einer bestimmten Abstraktion und die Verwendung zur Eigenschaftsemulation bietet sich die Verwendung von Templates an. Eine Unterteilung in *Boolean Layer* und *Temporal Layer* erscheint daher nur wenig sinnvoll zu sein. Zwar erscheint auch prinzipiell die Nutzung temporallogischer Eigenschaften möglich, allerdings stellt die Emulation bzw. Synthese temporallogischer Eigenschaften ein eigenes, teilweise noch offenes Forschungsfeld dar. Die zur Verhaltensbeschreibung genutzte Ebene wird daher als *Template Layer* bezeichnet.

4.6.4. Einsatz der Constraint Language

Die Spezifikationen basierend auf Prozess-basierten Eigenschaften werden im weiteren Verlauf dieser Arbeit als *Constraint Language* bezeichnet. Dazu wurden in den vorangegangenen Ab-

[31] Im Allgemeinen ist es für die Prüfung einer Eigenschaft irrelevant, welche Rolle (Ursache/Wirkung) ein spezifiziertes Signal besitzt, bzw. wie die zeitliche Referenzierung erfolgt.

schnitten 4.6.1, 4.6.2 und 4.6.3 konzeptionelle Überlegungen dargestellt.

Die konkrete Definition der Sprache erfordert die Berücksichtigung von Aspekten, die in Zusammenhang mit dem Transformation-basierten Ansatz stehen, siehe Kapitel 5. Vor allem bezüglich der Eigenschaftsgenerierung ergeben sich Besonderheiten, die an dieser Stelle keine anschließende Definition zulassen. Die konkrete Syntax der im realisierten Prototyp genutzten *Constraint Language* ist im Anhang A.1 zu finden.

4.7. Zusammenfassung Eigenschaftsspezifikation

Ausgehend von Anforderungen an die Eigenschaftsspezifikation für Zeiteigenschaften in Performancemodellen wurden Prozessbasierte und Zustandsbasierte Eigenschaften als zwei mögliche Klassen von Eigenschaften identifiziert, untersucht und verglichen.

Weiterhin wurden aus der Semantik verschiedener Modellierungsdomänen Anforderungen an heterogene Eigenschaftsspezifikationen abgeleitet. Daraus wurde anschließend ein Konzept zur Organisation heterogener Eigenschaften entwickelt.

Ausgehend von diesen Analysen und den Anforderungen der zu betrachtenden formalen bzw. semiformalen Verifikationstechniken ist jeweils eine Spezifikationssprache für Eigenschaften entwickelt worden. Für die formale Verifikation eignet sich unter den betrachteten Voraussetzungen und der Beschränkung auf die DE-Domäne eine Spezifikation Prozess-basierter Eigenschaften (Constraint Language) am ehesten. Diese erlaubt die Beschreibung von Templates, die sowohl die Rolle von Zusicherungen als auch die Rolle von Anforderungen annehmen können. Gleichzeitig wird der kausalen Semantik der Events im Sinne einer Prozesskette (Eventfluss) Rechnung getragen. Damit bleiben auch Informationen über kausale Abhängigkeiten erhalten.

Für die semiformale Verifikation heterogener Modelle und Eigenschaften ist eine Spezifikation basierend auf heterogenen temporalen Eigenschaften (XFLTL) sinnvoll. Diese beschreibt quasikontinuierliche lineare temporale Beziehungen zwischen Events. Für heterogene Modelle und Eigenschaften muss eine Konvertierung erfolgen. Sämtliche analogen und digitalen Signale und Eigenschaften werden dabei auf Beziehungen zwischen Events abgebildet. Die Semantik der Spezifikationen bezieht sich auf die seiteneffektfreie Betrachtung in einem synchronisierten Zeitmodell. Das bedeutet, dass die vom Modell erzeugte Reihenfolge simultan auftretender Ereignisse (kausale Abhängigkeiten) nicht beobachtbar ist.

5. Transformation-basierte Verifikation von Systemmodellen

Das grundlegende Konzept der Transformation-basierten Verifikation wurde bereits im Abschnitt 2.6 erläutert. Aus dem Konzept ergeben sich Fragestellungen zur Spezifikation funktionaler Eigenschaften, auf die im Kapitel 4 eingegangen wurde. Die noch offenen Fragestellungen zur Transformation selbst sowie zur formalen Interpretation der Eigenschaften werden in diesem Kapitel betrachtet.

Dazu wird in Abschnitt 5.1 das Konzept im Detail vorgestellt. In Abschnitt 5.2 werden die Modellstruktur des Ausgangsmodells der Transformation mathematisch definiert sowie die Analyse und Bewertung der Modellstruktur betrachtet. In Abschnitt 5.3 wird die Transformation von Eigenschaften vorgestellt und es werden Interpretationsmöglichkeiten in verschiedenen Verifikationswerkzeugen diskutiert. Abschnitt 5.5 stellt die prototypische Realisierung der Methode vor und geht dabei detailliert auf technische Aspekte ein. In Abschnitt 5.6 werden Aspekte der Validierung der Methode diskutiert. Die Resultate werden anschließend in Abschnitt 5.7 zusammengefasst und bewertet.

5.1. Konzept zur Transformation-basierten Verifikation

Für die Realisierung einer Transformation müssen strukturelle und semantische Beziehungen zwischen Quell- und Zielmodell hergestellt werden. Den Kern des Ausgangsmodells stellen funktionale Eigenschaften dar, die einzelnen Elementen innerhalb der Modellstruktur des Quellmodells zugeordnet sind. Die Struktur des Quellmodells beschreibt die Verbindung der enthaltenen Elemente und Funktionen, insbesondere auch aus Sicht der Kommunikation. Daher muss die Kommunikationsstruktur während der Transformation erhalten bleiben.

Moderne Ansätze zur Systemmodellierung benutzen eine Modulare bzw. Objektorientierte Strukturierung. Das bedeutet, dass Funktionalitäten in Blöcken gekapselt und durch Schnittstellen (Interfaces) zugänglich gemacht werden. Ein solches Konzept wird praktisch in allen relevanten Beschreibungssprachen verwendet. Des Weiteren besteht in den genannten Beschreibungssprachen die Möglichkeit, eine hierarchische Strukturierung von Blöcken vorzunehmen, um die Wiederverwendbarkeit von Teilmodellen zu ermöglichen und eine übersichtlichere Strukturierung zu erhalten.

Das dynamische Verhalten ist dabei innerhalb der Blöcke gekapselt, wobei die Dynamik eines Blockes als Zustand repräsentiert werden kann. Die Modelle realisieren dabei eine serialisierte Ausführung einzelner Blöcke, d.h. es kann sich in einem Simulationsschritt nur der Zustand eines einzelnen Blockes ändern. Dies gilt auch dann, wenn der Benutzer die elementaren Simulationsschritte nicht explizit sieht, wie etwa bei Very High Speed Integrated Circuit Hardware Description Language (VDHL)[32].

Aus Gründen der Lokalität sollten funktionale Anforderungen (Assertions) möglichst nahe am Designobjekt hinterlegt werden [FKL04]. Im Idealfall ist die funktionale Spezifikation sogar Bestandteil des Interfaces. Diese Anforderungs- und Eigenschaftsspezifikationen besitzen die selben Schnittstellen, wie das Designobjekt und können daher strukturell äquivalent betrachtet werden.

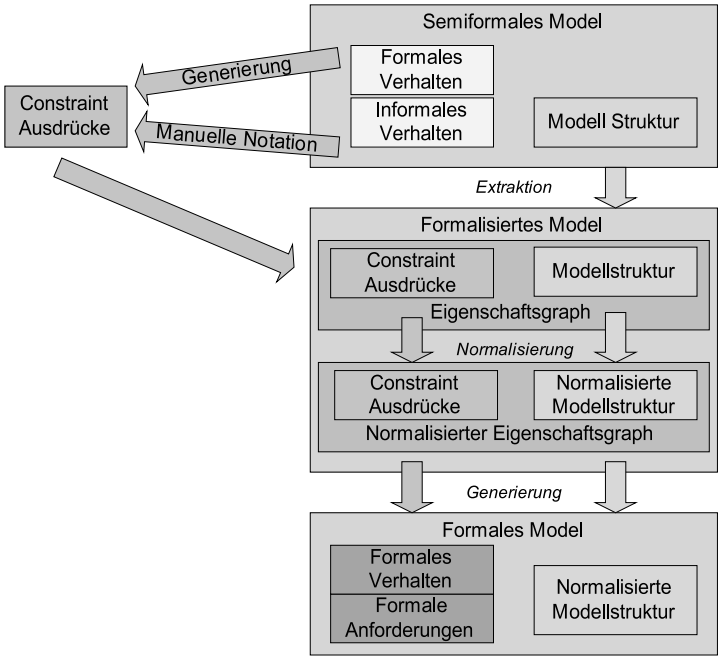


Abbildung 5.1.: Struktur der Transformation-basierten Verifikation

Für die funktionale Verifikation hierarchisch strukturierter Modelle im Sinne einer strukturellen Hierarchie ist eine Normalisierung der Modellstruktur erforderlich, da nur wenige Verifikationsmethoden eine echte hierarchische Modellbeschreibung akzeptieren [Bey02]. Ist ein Modell aus Teilmodellen zusammengesetzt (komponiert), so ist die Hierarchie rein kompositionaler (struktureller) Natur. Anderenfalls enthält das Modell eine Verschachtelung (Inklusion) von Teilmodellen unterschiedlicher Detailgrade, so dass es sich um eine funktionale Hierarchie handelt, wie etwa bei hierarchischen Zustandsmodellen. Der wesentliche Unterschied besteht

[32]Hier wird das Modell getaktet betrachtet, der Benutzer nimmt die Änderung des Zustandes der einzelnen Blöcke als synchron war.

darin, dass die Funktion einer strukturell verfeinerten Komponente ohne die innere Struktur nicht analysierbar ist, während die funktional verfeinerte Komponente eine Aussage über die abstrakte Funktion zulässt.

Abbildung 5.1 zeigt die Aktivitäten zur Realisierung einer Transformation-basierten Verifikation. Für die Transformation-basierte Verifikation ist zunächst die Struktur der Modelle unter Berücksichtigung der Schnittstellen und kompositionaler Effekte formal zu definieren, siehe Abschnitt 5.2. Dies beinhaltet wird die Analyse und Normalisierung der Struktur im Kontext der enthaltenen Eigenschaften und Anforderungen. Dabei werden die Eigenschaften in die bestehende Struktur des Modells eingefügt. Danach werden die einzelnen Bestandteile, also Eigenschaften und Anforderungen, unter Beibehaltung der strukturellen Merkmale in ein Modell zur Verifikation transformiert, das sich von einem Werkzeug zur Verifikation direkt verarbeiten lässt, siehe Abschnitt 5.3. In weiteren Abschnitten folgen Erläuterungen zur Umsetzung und Validierung dieser Methode.

5.1.1. Kombinierter Transformations-/ Verifikationsalgorithmus

Die Tatsache, dass nicht alle Bestandteile eines Modells auf einmal, sondern inkrementell überprüft werden sollen, kann dazu benutzt werden, die Verifikation ebenfalls inkrementell durchzuführen. Der hierarchische Verifikationsansatz unterstützt diese Vorgehensweise.

So kann die Transformation in mehrere Schritte aufgeteilt werden, siehe Abbildung 5.2:

- Import des Modell und der zugehörigen Eigenschaften entsprechend der betrachteten Hierarchieebenen.
- Integration der Eigenschaften in die Modellstruktur.
- Normalisierung der Modellstruktur (Auflösung der Modellhierarchie).
- Analyse der Struktur und Ersetzung nicht annotierter Funktionsblöcke.
- Optimierung der Modellstruktur.
- Transformation des Modells, Generierung von Anforderungen und Zusicherungen.
- Verifikation des Transformierten Modells.
- Ggf. Iteration über mehrere Hierarchieebenen

Es lassen sich für die Verifikation von Eigenschaften drei wesentliche Aspekte unterscheiden:

- Struktur,
- Verhalten und
- Zeit.

Diese drei Aspekte schlagen sich auch in der Gestaltung der Constraint Language nieder. Dabei ist zu bemerkten, dass MLDesigner als Darstellungsmittel die Strukturbeschreibung unterstützt, aber kaum eine formale Beschreibung von Verhalten und Zeiteigenschaften, weshalb ergänzend die annotierten Eigenschaften gemäß Abschnitt 4.6 genutzt werden.

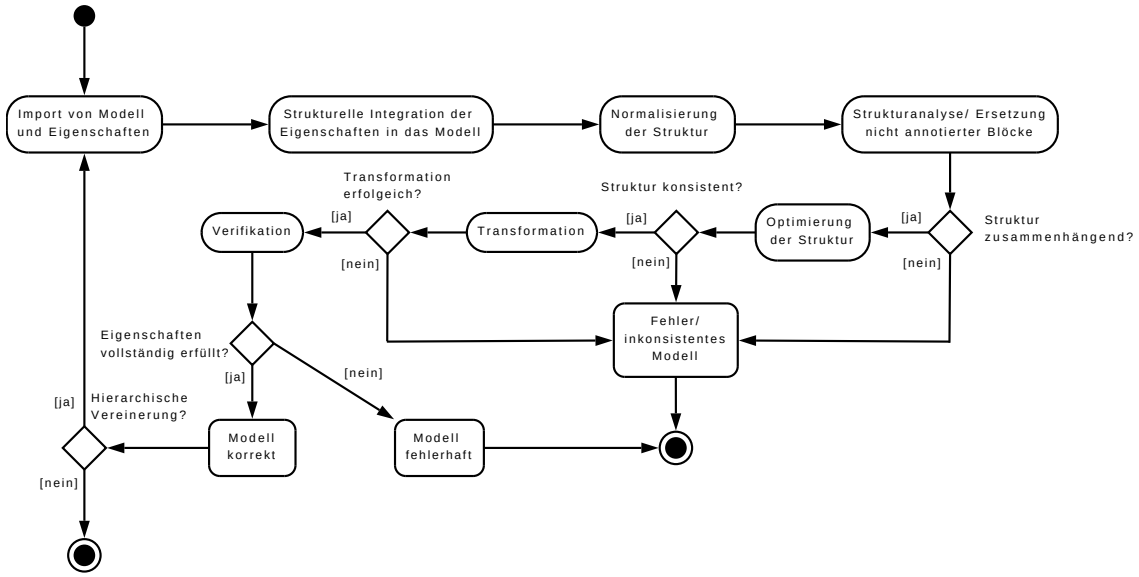


Abbildung 5.2.: Vorgehensweise zur Transformation-basierten Verifikation

5.1.2. Hierarchie und Eigenschaften

Die Anwendung von Verfeinerungen kann dazu benutzt werden, eine hierarchische Modellstruktur zu erzeugen. Die verschiedenen Hierarchieebenen repräsentieren dann jeweils eine Stufe der Verfeinerung, so dass in jedem Schritt überprüft werden kann, ob und welche Eigenschaften sich ändern, bzw. welche zusätzlichen Constraint im verfeinerten Modell als Eigenschaften gelten müssen. Dabei ist egal, ob die Hierarchie für den Anwender sichtbar ist oder nicht. Bei Software etwa können die verwendeten Kontrollstrukturen entsprechend ihrer Verwendung als Baum, also als Hierarchie angezeigt werden.

Abbildung 5.3 zeigt ein hierarchisches Modell. Im oberen Teil ist die abstraktere Ebene dargestellt, wobei die definierten Anforderungen (C0_1, C0_2) durch realisierte Eigenschaften (C1_1, C1_2) eines oder mehrer Blöcke abgebildet werden müssen. Auf nächst niedrigerer Hierarchieebene (Abbildung 5.3 unten) dienen diese Eigenschaften als Anforderung und werden wiederum durch Eigenschaften (C2_1, C2_2, C2_3) realisiert.

Im Folgenden wird davon ausgegangen, dass die hierarchische Struktur (Baum) eines Modells (in Richtung der Blätter) die Schritte der Verfeinerungen angibt. Weiterhin wird davon ausgegangen, dass die zu prüfende Eigenschaft für einen Knoten k_i der Ebene k definiert ist und nicht direkt von benachbarten oder übergeordneten Knoten abhängt. Falls Nebenbedingungen existieren, können diese auch unabhängig von der Realisierung in benachbarten oder übergeordneten Knoten erzeugt werden. Bezüglich der betrachteten Eigenschaft ist dann die Abspaltung eines hierarchischen Eigenschaftsgraphen mit dem ursprünglichen Knoten k_i als Wurzel möglich.

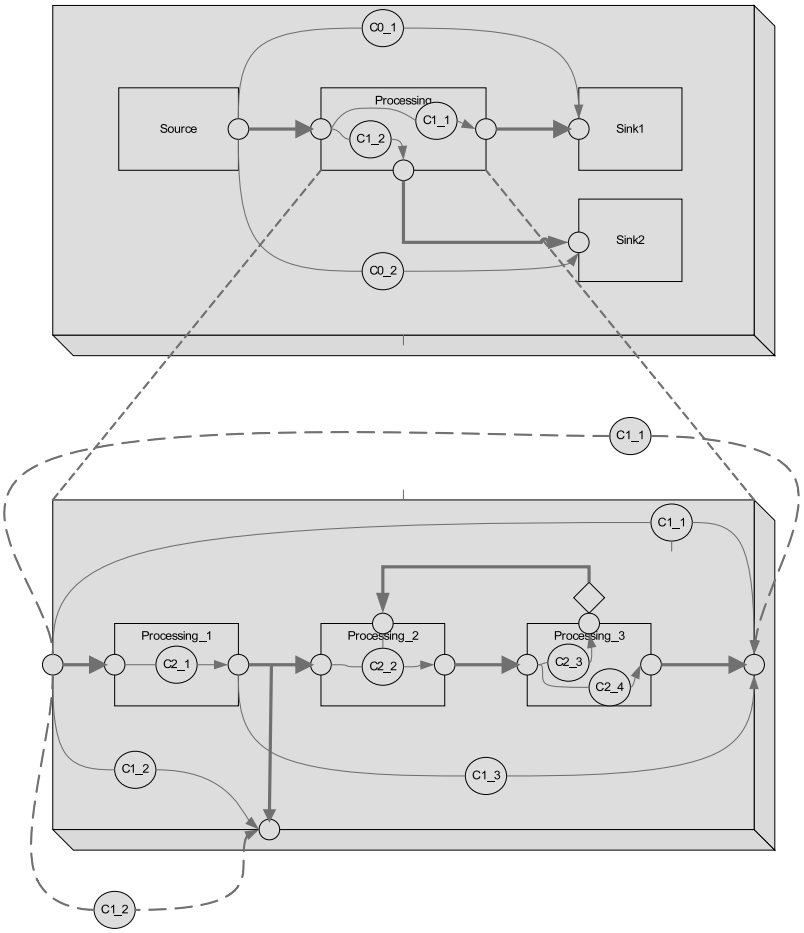


Abbildung 5.3.: Hierarchische Eigenschaften

Auf nächst niedrigerer Hierarchieebene existieren dann mehr Teilkomponenten, weshalb auch mehr Eigenschaften beschrieben werden. Es ist also zu erwarten, dass sich die Gesamtzahl der beschriebenen Eigenschaften zu den Blättern des Eigenschaftsbaums hin erhöht und dort sein Maximum annimmt. Zwischen zwei Ebenen muss also eine Abbildung von Eigenschaften der niedrigeren auf die Eigenschaften der höheren Ebene gefunden werden (können).

5.2. Strukturanalyse

Die Strukturierung erfolgt hierarchisch durch Blöcke. Diese besitzen jeweils ein Interface (*BlockInterface*) und eine Realisierung. Ein Block kann entweder als Basisblock (*Primitive*) oder hierarchisch zusammengesetzter Block (*ModulBlock*, *Composite*) realisiert werden. Eine Sonderrolle nimmt das Gesamtsystem ein, das ein *ModulBlock* ohne eigene Interfaces darstellt. Die resultierende Systembeschreibung ähnelt einem Kompositionsstrukturdiagramm (KSD)(engl. composite structure diagram) der UML, Version 2.0.

Allerdings unterscheiden sich die verwendeten Blöcke in ihrer Definition, siehe Tabelle 5.1. Während die UML beliebige Klassen (Classifier) zulässt, werden für die verallgemeinerte Strukturbeschreibung von MLD-Modellen Einschränkungen vorgenommen, um unterschiedliche Simulationssemantiken (*Domänentypen*) unterstützen zu können. Blöcke in MLDesigner (MLD) dürfen im Gegensatz zu UML-Klassen keine Methoden enthalten[33]. Stattdessen wird das Verhalten in Form von überladener Standardmethoden realisiert. Wie in der UML werden Ports zum Zwecke der Kapselung strukturell äquivalent verwendet. Die Verwendung von Ports ist bei MLD im Gegensatz zur UML zwingend und dient bei Blöcken der Modellierung dynamischer Informationsflüsse. Ports dienen im Gegensatz zur UML nicht der Delegation von Methoden. Die Aktivierung einer Funktion im Sinne eines Kontrollflusses erschließt sich erst durch die Simulationssemantik, die im Simulationsframework Domänen-spezifisch in Form eines Koordinators hinterlegt ist. Blöcke besitzen keine Attribute im Sinne der UML. Es gibt zwei Konzepte, die eine Realisierung von Attributen erlauben - Parameter und Memories. Parameter ermöglichen eine statische Parametrisierung zur Modellierungszeit (Kompilationszeit). Daher besteht die Möglichkeit Parameter über Funktionen mit Parametern eines übergeordneten Blockes zu verknüpfen oder manuell festzulegen. Memories (engl. für Speicher) erlauben das Speichern von Informationen. Durch die Sichtbarkeit von Memories wird die Verfügbarkeit der enthaltenen Informationen in expliziter Weise strukturell von einem Block auf Teile seiner hierarchischen Verfeinerung begrenzt. Bezüglich dieses Blocks ist der betreffende Memory intern, und bezüglich der Blöcke seiner Verfeinerung extern definiert. Ist ein Memory intern, so ist seine Sichtbarkeit nach oben hin auf den betreffenden Block beschränkt und wird dort auch instantiiert. Ist ein Memory extern, so stellt er innerhalb des betreffenden Blocks nur eine Referenz auf einen Memory der nächsthöheren Hierarchieebene dar.[34] Wegen des referenzierenden Charakters eines Memory muss ein Zugriff stets atomar, also asynchron geschehen.

Bedeutung	UML	MLD	Bemerkung
Type/Bock	Klasse	Model	Block im Modell
Port	Port	Port	Interaktionspunkt von Blöcken
Link	Konnektor	Relation	Verbindung zwischen Blöcken
Konstante	konstantes Attribut	Parameter	Konstanter Parameter
lokale Variable	Attribut	interner Memory	lokale Blockvariable
globale Variable	Attibutreferenz/ Assoziation	externer Memory	globale Variable, im Block genutzt

Tabelle 5.1.: Vergleich der Strukturmerkmale von UML- und MLDesigner-Modellen

[33] Interne Methoden sind zwar Möglich, spielen aber im Kontext der Modellstruktur keine Rolle.

[34] Der Einfachheit halber werden interne Memories von PrimtivBlöcken als C/C++ Variablen deklariert. Für die formale Betrachtung werden diese Variablen als interne Memories betrachtet.

5.2.1. Definition

Die hierarchische Strukturierung eines Systems ist unter Vernachlässigung von Informationsflüssen als gewurzelter Baum beschreibbar. Dabei sind die Knoten als Blöcke definiert, während die Kanten die direkte hierarchische Inklusion zwischen zwei Blöcken beschreiben. Die Wurzel eines solchen Baumes wird als *SystemBlock* (Menge SB), die inneren Knoten als *ModulBlock* (Menge MB) und die Blätter als *PrimitivBlock* (Menge PB) bezeichnet. Die Menge aller Blöcke eines hierarchischen Systems ergibt sich somit aus der Vereinigung der disjunkten Mengen $B = SB \cup MB \cup PB$.

Besitzt ein Block von der Wurzel aus gesehen die Distanz n, so wird er als Block der n-ten Hierarchieebene bezeichnet. Direkte Informationsflüsse können genau dann zwischen zwei Blöcken bestehen, wenn sie im Hierarchiebaum durch eine Kante verbunden sind, sich also auf benachbarten Hierarchieebenen befinden und sich in einer direkten hierarchischen Inklusionsbeziehung zueinander befinden.

Um die Wiederverwendbarkeit zu gewährleisten, werden Bockinstanzen B und Blocktypen BT unterschieden.[35] Ein hierarchischer Blocktyp (ModulBlockTyp) ist rekursiv definiert als Tupel $MBT = (B, P, L, V, M)$, wobei

- $B = \{B_1, B_2, \ldots B_{N_l}\}$ ist eine Menge von BlockTyp-Instantiierungen $B \subset BT \times \mathbb{N}$.
- $P = \{P_1, P_2, \ldots P_{N_m}\}$ ist eine Menge von Ports. Ein Port ist ein Zugriffspunkt für Informationsflüsse und ist dem BlockTyp zugeordnet. Ports werden, entsprechend der Sichtweise des zugehörigen Blockes, in Eingabeports P^{in} und Ausgabeports P^{out} unterschieden, wobei $P^{in} \cap P^{out} = \emptyset$ gilt.
- $L \subset (2^{P^{in}} \cup 2^{P_b^{out}}) \times (2^{P^{out}} \cup 2^{P_b^{in}})$ ist eine Menge von gerichteten Links, die Ports des hierarchischen Blocktyps sowie Ports seiner enthaltenen Blocktyp-Instantiierungen verbindet. $(P_b = \bigcup P(B_i))$
- $V = \{V_1, V_2, \ldots V_{N_n}\}$ ist eine Menge von Parametern. Ein Parameter ist dem BlockTyp zugeordnet und besitzt einen Datentypen.
- $M = \{M_1, M_2, \ldots M_{N_o}\}$ ist eine Menge von Memories. Ein Memory ist dem BlockTyp zugeordnet und besitzt einen Datentypen. Memories werden in *intern* (M^{int}) und *extern* (M^{ext}) unterschieden. $(M^{int} \cup M^{ext} = M; M^{int} \cap M^{ext} = \emptyset)$

Bei Primitivblöcken PB gilt $B = \emptyset$ und $L = \emptyset$. Bei Systemblöcken SB wird nicht zwischen Instanzen und Typen unterschieden, da jeweils genau eine Instanz auf Wurzelebene der Modellhierarchie existiert. Bei der Instantiierung eines BlocktTyps werden die formalen Parameter

[35] Dementsprechend werden die Abkürzungen SBT (SystemBlockTyp), MBT (Modulblocktyp) und PBT (PrimitivBlockTyp) verwendet.

durch aktuelle Parameter ersetzt, während die Attribute (B, P, L, M) instantiiert werden. Dabei werden für die instantiierten Attribute neue IDs vergeben. Bei der Instantiierung eines PrimitivBlocks PB wird direkt ein Objekt der beschreibenden Klasse erzeugt.

5.2.2. Normalisierung von kompositional-hierarchischen Modellen

Die hierarchische Strukturierung bietet Vorteile bei der Beschreibung von komplexen Modellen, da Teilstrukturen wieder verwendet werden können. Andererseits bieten formale Analysemethoden oft nicht die Möglichkeit hierarchische Modelle direkt einzubeziehen.

Da einzig die ModulBlöcke zur Bildung hierarchischer Strukturen verwendet werden können[36] und diese keine eigenständige Funktionalität bieten, besteht die Möglichkeit, eine Normalisierung auf struktureller Ebene vorzunehmen. Einzig die Übergänge zwischen Domänen müssen weiterhin berücksichtigt werden.

Ein hierarchisches System S kann normalisiert werden, in dem man, unter Beibehaltung der Informationsflüsse, seine Struktur in einen Baum der Tiefe 1 umstrukturiert. Da ein Baum der Tiefe 1 keine inneren Knoten mehr enthält, besitzt das normalisierte System S' demnach keine ModulBlöcke mehr ($|MB(S')| = 0$). Bei der Normalisierung wird die Modulhierarchie entsprechend der Tiefensuche [Tur04] beginnend beim SystemBlock SB durchlaufen. Ist ein Knoten vom Typ *ModulBlock*, so werden die Bestandteile seines Interfaces verarbeitet und in den Systemblock eingegliedert:

- Werte von Parametern werden aus Werten des übergeordneten Blockes errechnet. Dies ist möglich, weil der *SystemBlock* nur absolute Parameter enthält [37] und die Parameter hierarchisch enthaltener Blöcke nur von Parametern des übergeordneten Blockes abhängen dürfen.

- Referenzen von *Memories* werden im Falle externer Memories auf Referenzen des übergeordneten *Blockes* gesetzt. Im Falle eines internen Memories wird ein neuer *Memory* im *SystemBlock* erzeugt und darauf referenziert.

- Interne Signale werden im *SystemBlock* neu erzeugt und die betreffenden Ports mit dem neu erzeugten Signal verbunden. Die Struktur von Links beschreibt Relationen zwischen Multimengen von Ports. Je nach betreffender Modellsemantik sind hierbei jedoch Einschränkungen zu berücksichtigen [MLD07]. Relationen zwischen Multimengen werden in 1:1 Relationen sowie Zusammenführungen (*Merge*) und Verzweigungen (*Fork*) aufgelöst. Sind interne Signale mit externen Ports verbunden, so wird ein neuer Verbindungsblock (*OneWay*) entsprechend der Richtung des externen Ports erstellt werden.

[36] FSM Module bieten zwar über die Option *SlavePath* die Möglichkeit zur Einbindung weiterer Module, allerdings dient dies nicht zur Bildung hierarchischer Strukturen.

[37] Bestehen mehrere sog. Parametersets, so müssen die Parameter für jedes Parameterset separat aufgelöst werden.

5.2.3. Normalisierung und Hierarchische Verifikation

Bei der hierarchischen Verifikation wird versucht, geforderte Eigenschaften (*assume*) eines Modells durch die Komposition zugesicherter Eigenschaften (*guarantee*) von Teilmodellen strukturell nachzuweisen. Dazu können u.a. Methoden des Theorem Prooving (Higher Order Logic) zum Einsatz kommen [Kro99, CH04].

Die Nutzung dieser Grundidee ist auch mit Techniken des Model Checking möglich, indem geprüft wird, ob die Komposition von Teilmodellen geforderte Eigenschaften des Gesamtmodells aufweisen[38]. Im verwendeten Kontext wird die Nutzung eines Ansatzes zur hierarchischen Komposition unterstützt, indem die Ausgangsmodelle eine hierarchische Struktur beschreiben, wobei meist eine strukturelle Komposition vorliegt.

Genutzt werden kann das Prinzip der Hierarchischen Verifikation, indem einzelne Hierarchieebenen im Modell komplett ausgespart werden, also die Normalisierung des Modells nur bis zu einer bestimmten Abstraktionsebene erfolgt. Statt dem garantierten Verhalten der enthaltenen Module und Primitive wird dann das spezifizierte (und bereits verifizierte) Verhalten eines kompletten Moduls verwendet. Dadurch sinkt die Größe des zu verifizierenden Modells. Allgemeinen geht mit dieser Abstraktion ein Verlust von Informationen einher und es findet im Kontext des Zeitverhaltens eine Worst-Case Betrachtung statt.

Die Anwendbarkeit einer solchen Methode ist jedoch an die Bedingung geknüpft, dass die Spezifikationen jeweils vollständig sind. Anderenfalls kann es passieren, dass aufgrund fehlender Teilmodelle oder Events fehlerhafte Aussagen getroffen werden. Die Analyse struktureller Dekompositionsbeziehungen zwischen zwei Modellen kann durch die Anwendung von Dekompositionsregeln geprüft werden. Für nicht nebenläufige Interpretationen von Modellen (zeitbewertete Petri Netze [FGM98, PZH96] , Timed Automata [Löt99]) existieren dazu zahlreiche Untersuchungen. Ein Ansatz für nebenläufige Prozesse ist im Anhang D dargestellt.

5.2.4. Eigenschaften des normalisierten Systems

Das normalisierte System besitzt keine Hierarchie. Die hierarchische Struktur ist ein entarteter Baum mit einem Wurzelknoten (SystemBlock) und Blättern (PrimitivBlöcke), jedoch ohne innere Knoten. Globale Informationen, in Form von Memories, befinden sich sämtlich im SystemBlock[39]. Die Parameter der PrimitivBlöcke enthalten Konstanten, alle weiteren Abhängigkeiten sind aufgelöst. Die Vernetzung der Primitivblöcke ist äquivalent zu der des hierarchischen Modells. Dies schließt möglicherweise Kreise und Eigenschleifen ein. Kreise werden strukturell durch gerichtete Links zwischen Primitiven erzeugt. Eigenschleifen bezeichnen Kreise, die nur aus einfachen Strukturen, wie *Merge*- und *Fork*- Primitive, bestehen und somit

[38] Wenn die betreffenden Eigenschaften in einem Kontext kombiniert werden, lässt sich die Gültigkeit einer komponierten Eigenschaft entscheiden.

[39] Interne Memories der PrimitivBlöcke werden hierbei nicht mitbetrachtet, da diese Bestandteil der Primitive sind.

bezüglich Events eine Endlosschleife darstellen[40]. Solche Endlosschleifen können im Modellierungswerkzeug erkannt werden und stellen einen Modellierungsfehler dar.

5.3. Eigenschaftstransformation

Die Strukturierung von Modellen wurde im Abschnitt 5.2 ausführlich beschrieben. Damit eine Verifikation von Eigenschaften durchgeführt werden kann, müssen geforderte und zugesicherte Eigenschaften relativ zur Modellstruktur definiert werden. Die betreffenden Modellelemente sind das Blockinterface, der SystemBlock und der ModulBlock (Composite).

5.3.1. Eigenschaftsannotation

Die Implementierung der DE-Primitive mittels C++-Code bietet eine hohe Ausdrucksmächtigkeit und ermöglicht eine sehr effiziente Simulation, bringt allerdings auch Probleme bei der formalen Interpretation mit sich. Folgende Varianten wurden für die Annotation von DE-Primitiven ermittelt:

- Formale Interpretation des C++-Codes im Kontext der DE-Domäne.
- Hinterlegung einer Primitive mit einer formalen Beschreibung.
  - Beschreibung als Automat
  - Beschreibung als temporale Eigenschaft
  - Beschreibung als Eventfluss

Die formale Interpretation des C++ Codes zielt auf eine automatisierte Programmanalyse in Form einer Automatengenerierung ab. Dabei werden statische Variablen des zu analysierenden Primitivs als Zustandsvariablen betrachtet, während die Ablauffunktion des Primitivs sowohl die Zustandsüberführungsfunktion als auch die Ausgabefunktion repräsentiert. Die Interpretation dieser Ablauffunktion muss daher im Kontext der DE-Domäne unter Berücksichtigung der Struktur des Primitivs geschehen. Eine solche Analyse ist mitunter komplex und erfordert möglicherweise Abstraktion bezüglich Kontrollfluss und Datenfluss.

Die Hinterlegung einer formalen Beschreibung stellt eine Alternative dar, bei der vom Designer eine alternative, möglicherweise abstraktere Repräsentation hinterlegt wird. Soll eine Beschreibung als Automat erfolgen, so kann direkt die FSM-Domäne verwendet werden. Hierbei besteht, mit der Ausnahme von Datenoperationen, die Möglichkeit, die gleiche vollständig formale Beschreibung sowohl zur Simulation als auch für die Analyse zu verwenden. Allerdings ist die Simulation von FSM-Primitiven teils weniger effizient, da eine Interpretation des FSM Modells wesentlich aufwendiger ist als die Abarbeitung von C++ Code.

[40] In der Interpretation als Petri-Netz ergibt sich eine Falle (Trap), also ein Teilnetz in der Marken (Ereignisse) gefangen werden und dieses nicht mehr verlassen können [DA05].

Die Verwendung temporaler Eigenschaften stellt eine weitere Alternative für die Beschreibung der Funktion eines Primitivs dar. Zunächst wäre zu entscheiden, welche temporale Beschreibungssprache verwendet wird (z.B. LTL/CTL), und wie die Semantik bezüglich Zuständen und Events definiert ist. Zudem kann sich die Beschreibung als aufwändig gestalten, einerseits bezüglich der Korrektheit[41], andererseits im Hinblick auf die Vollständigkeit der Eigenschaftsspezifikation. Für die weitere Verwendung einer temporalen Eigenschaftsbeschreibung ist weiterhin zu beachten, dass eine Methode zur Simulation von Eigenschaften bzw. zur Automatengenerierung aus Eigenschaften (Eigenschaftssynthese) erforderlich ist. Hierzu sind zwar Ansätze entwickelt worden, für die betrachtete Klasse von Eigenschaften stellt dies jedoch eine sehr aufwendige Realisierung dar.

Betrachtet man die Semantik der DE-Domäne, so erscheint eine Beschreibung als Eventfluss als naheliegend. Viele Primitive stellen bezüglich der Bearbeitung von Events eine nichtdeterministische Entscheidung dar, führen eine Synchronisation oder einen möglicherweise zeitbehafteten Prozess aus. Solche Primitive oder Module können als Grundelemente einer wohlgeformten Struktur betrachtet werden. Dementsprechend erscheint es sinnvoll, Grundelemente zur Beschreibung solcher Modellelemente zur Verfügung zu stellen. Dazu lassen sich Templates verwenden.

Tabelle 5.2 stellt die vorgestellten Varianten der Eigenschaftsannotation anhand von Abschätzungen gegenüber.

Formalität der Beschreibung bezeichnet die formale Eindeutigkeit der Annotation. Hier gibt es bei der FSM-Domäne Abstriche aufgrund möglicher C++ Fragmente in Bedingungen und Aktionen. Beim Eventfluss gibt es Abstriche, da die Eindeutigkeit erst indirekt durch die Templates erreicht wird.

Die *Ausdrucksstärke der Beschreibung* ist ein Indiz für die notwendige Abstraktion der DE-Primitive für die Realisierung der Annotationen. Bei einer stark reduzierten Ausdrucksmächtigkeit besteht die Gefahr, dass Modellaspekte nicht unter Beibehaltung der ursprünglichen Modellstruktur abgebildet werden können. Entsprechend ist die Modellbeschreibung als DE-Primitiv von Vorteil, während bei der Abstraktion als Eventfluss mitunter Abstriche gemacht werden müssen. Bei der Codeanalyse ist die Ausdrucksstärke von der gewählten Abstraktion abhängig.

Mit *Formalität der Methode* wird das Vorgehen der entsprechenden Variante bezeichnet. Die Codeanalyse der Primitive wäre hier die beste Alternative, da keine oder nur wenige zusätzliche Informationen erforderlich sind. Bei den anderen Varianten ist für jedes Primitiv eine Ersatzbeschreibung zu hinterlegen, was zu Fehlern führen kann. Problematisch bei der direkten Verwendung von Automaten ist die Gefahr, unterschiedliche Semantiken in Quell- und Zielmodell zu realisieren.

[41] Die direkte Spezifikation mittels Temporaler Logik, wie CTL und LTL, gilt im Allgemeinen als schwierig und fehleranfällig.

Kriterium	Codeanalyse Primitiv/C++	Annotation Automat/FSM	Annotation temporal	Annotation Eventfluss
Formalität der Beschreibung	++	++/+	++	+
Einfachheit der Beschreibung	−	+/++	−−	+
Ausdrucksstärke der Beschreibung	+[42]	∘/+	∘	−
Formalität der Methode	+	−/∘	∘	∘
Konsistenz zum Modell	++[43]	−−/+	−−	−
Automatisierbarkeit	+[44]	−−/∘	−−	−−
Aufwand Modellierung	++	−−/−	−−	−
Aufwand Umsetzung	−−	++/∘	−	∘

Tabelle 5.2.: Vergleich von Annotationsmöglichkeiten (++: sehr gut, +: gut, ∘: neutral, −: schlecht, —: sehr schlecht)

Die *Konsistenz zum Modell* betrachtet die methodisch bedingte Sicherstellung der Konsistenz zwischen Modell und Annotation. Für die Codeanalyse ist die Sicherstellung garantiert. Dies gilt ebenso für FSM-Modelle, falls diese ersatzweise für FSM-Primitive eingesetzt werden. Für alle anderen Annotationen müssen Abstriche gemacht werden, da eine manuelle Zuordnung erfolgt. Da Beschreibungen mittels temporaler Logik als komplex gelten, wurde eine weitere Abwertung vorgenommen. Bei reinen Automaten besteht die Gefahr, dass die Semantik des Zielmodells nicht korrekt eingehalten wird.

Ein weiteres Kriterium ist die *Automatisierbarkeit* der Annotation. Falls korrekt umgesetzt, ist die Codeanalyse am ehesten für die automatisierte Erzeugung von Annotationen geeignet, einzig Entscheidungen zur Steuerung der Analyse im Hinblick auf mögliche Abstraktionen müssen berücksichtigt werden. Annotationen mittels FSM-Primitiv werden als neutral betrachtet, da diese zwar nativ verwendet werden können, bei Verwendung als zusätzliche Annotation jedoch Konsistenzprobleme auftauchen können. Bei den anderen Varianten sind Konsistenzprobleme ebenfalls prinzipiell möglich.

Ein entscheidender Faktor für die Akzeptanz einer Annotation ist der zusätzlich erforderliche *Aufwand der Modellierung*. Bei der Codeanalyse fällt kein zusätzlicher Aufwand für den Model-

[42] Je nach Abstraktion.
[43] Sofern keine Abstraktion erforderlich.
[44] Falls umsetzbar.

lierer an. Die anderen Methoden erfordern einen ggf. recht hohen Aufwand für die zusätzliche Annotation von Primitiven.

Der geschätzte Aufwand für die Realisierung der jeweiligen Annotation wird im Kriterium *Aufwand Umsetzung* betrachtet. Der Aufwand für die Umsetzung der Codeanalyse ist als sehr hoch einzuschätzen, da viele Einflussfaktoren berücksichtigt werden müssen. Die direkte Annotation eines Automaten für das Verifikationsmodell ist augenscheinlich am einfachsten umzusetzen. Die Umsetzung von Templates und FSM-Primitiven ist ähnlich aufwendig. Die Verwendung temporaler Annotationen ist dagegen schwieriger, da zwar die Annotation an sich einfach ist, sich die notwendige Automatensynthese jedoch als komplex erweisen kann.

Als Resultat der verschiedenen Alternativen zur Annotation von Eigenschaften wäre die Codetransformation am günstigsten. Allerdings scheint die Umsetzung dieser Methode sehr aufwändig zu sein und große Risiken bezüglich der Umsetzbarkeit zu bergen. Die Verwendung von Automaten der Zieldarstellung ist die einfachste Möglichkeit. Allerdings kann hier keine feste Semantik zwischen Quellmodell und Zielmodell sichergestellt werden. Die Verwendung von FSM Primitiven stellt einen Mittelweg zwischen Codeanalyse und Annotation von Automaten dar. Hier besteht die Möglichkeit der formalen Beschreibung unter Beibehaltung der Konsistenz.

Die Nutzung temporaler Assertions stellt eine interessante Variante dar, insbesondere wegen der Formalität der Beschreibung, sowie im Hinblick auf die Weiterverwendung. Allerdings zeichnen sich Einschränkungen in der Praktikabilität ab, insbesondere auch Probleme in Bezug auf die Vollständigkeit der Annotationen. Zudem ist die erforderliche Eigenschaftssynthese nicht trivial. Annotationen zum Eventfluss besitzen prinzipbedingte Einschränkungen der Ausdrucksmächtigkeit. Allerdings ermöglichen sie eine praktikable Annotation einfacher Eigenschaften.

Nach Abwägung der Möglichkeiten erscheint die Kombination der Varianten FSM-Primitiv und Eventfluss als sinnvoll, da sich beide Varianten ergänzen. Die Annotation mit temporalen Eigenschaften ist als sehr komplex zu bewerten, einerseits in der Erstellung der Annotation und andererseits in der Interpretation zur Analyse bzw. zur Generierung.

5.3.2. Interpretation von DE-Modellen

Die Interpretation von DE-Modellen gestaltet sich schwierig, da sich der zugrunde liegende Formalismus von denen zustandsbezogener Modelle unterscheidet. Während die zustandsbezogenen Modelle als zusammenhängende Strukturen realisiert werden, bei denen eine gleichberechtigte Propagierung und Bewertung von Änderungen erfolgt, wird bei DE-Modellen die Ereignispropagierung von der Ereignisbearbeitung getrennt.

In Bezug auf die Semantik ist die Forderung nach der atomaren Bearbeitung von Events von besonderer Bedeutung. Die Ereignisbearbeitung erfolgt atomar, so dass ein weiteres Ereignis erst dann propagiert wird, wenn der vorherige, möglicherweise komplexe Bearbeitungsschritt beendet ist. Erfolgen zur Bearbeitung eines Events ein oder mehrere Zugriffe auf globale Infor-

mationen (Memories), so muss die Bearbeitung atomar erfolgen. Ebenso muss ausgeschlossen sein, dass gleichzeitig zwei Events desselben Typs bearbeitet werden, da ansonsten lokale Informationen inkonsistent werden können.

Forderung 1: *Atomare Bearbeitung von Events.*

Weiterhin ermöglichen DE-Modelle das Scheduling eines Events in einer abgekoppelten Datenstruktur, wodurch beliebig viele nebenläufige Events desselben Typs realisiert werden können. In einem zustandsbezogenen Paradigma ist die Emittierung von Events nur direkt durch Zustandsübergänge realisierbar, weshalb für nebenläufige Events zusätzliche (statische) Strukturen für „wartende" Events erforderlich sind.

Forderung 2: *Unterstützung nebenläufiger Events.*

Das zugrunde liegende DE-Modell des MLDesigner benutzt ein absolutes, globales Zeitmodell. Da die Zeit als orthogonaler Bestandteil in die Zustandsbeschreibung eingeht, ist ein solches Zeitmodell nur bedingt für die Verifikation geeignet. Mathematische Zustandsrepräsentationen verwenden daher in der Regel ein relatives, lokales Zeitmodell.

Forderung 3: *Verwendung lokaler relativer Zeitinformation*

Tabelle 5.3 zeigt einen Vergleich der Eigenschaften verschiedener mathematischer Modelle bei der Interpretation von DE-Modellen. Die erste Spalte zeigt das Vergleichskriterium, die zweite die Eigenschaften des DE-Modells. Darauf folgen die Eigenschaften von zeitbewerteten Petri-Netzen und Timed Automata, siehe Abschnitt 2.4.2. Die letzte Spalte zeigt die Eigenschaften eines selbst entwickelten, alternativen Modellierungsansatzes [PF07], siehe auch Anhang E. Man erkennt, dass unterschiedliche Mechanismen zur Ereignisdarstellung verwendet werden. Die Ereignissimulation, die beim DE-Modell durch einen Scheduler getrennt von der Ereignisbearbeitung betrachtet wird, ist bei zeitbewerteten Petri-Netzen durch das Schalten von Transitionen und bei Timed Automata durch Synchronisationsevents realisiert, während beim Formalen DE-Modell, ein spezieller Transitionstyp verwendet wird. Damit werden, mit Ausnahme der zeitbewerteten Petri-Netze, unterschiedliche Mechanismen für die Darstellung von Ereignissen und Ereignisbearbeitung verwendet. Im Gegensatz zum DE-Modell verwenden die mathematischen Modelle, entsprechend *Forderung 3* ein relatives, lokales Zeitmodell. Timed Automata und zeitbewertete Petri-Netze können *Forderung 2* nach nebenläufigen Events nur durch zusätzliche Strukturen erfüllen, wobei der Grad der Nebenläufigkeit bekannt sein muss. Das formale DE-Modell besitzt diese Einschränkung nicht. Die Datenmodellierung gestaltet sich prinzipiell als schwierig. Bei skalaren Typen sind Daten als Faltung auf ein Event modellierbar. Bei Petri-Netzen können unterschiedliche Farben genutzt werden, während bei den anderen Beschreibungsmitteln eine Entfaltung von Events mit skalaren Daten zu verschiedenen Events erforderlich ist.

Die Modellierung von verteilten Daten wird zur Propagierung von sichtbaren Systemzuständen im Modell genutzt und stellt ein sehr ausdrucksstarkes Merkmal dar. Die betrachteten Beschreibungsmittel erlauben zwar keine Einschränkung der Sichtbarkeit, allerdings ist die Ver-

wendung global sichtbarer Datenstrukturen möglich.

Die Hierarchie der DE-Modelle kann von keinem der betrachteten Beschreibungsmittel direkt abgebildet werden. Da die Hierarchie in DE-Modellen jedoch rein kompositionaler Natur ist, kann die Hierarchie jedoch komplett aufgelöst werden, so dass ein flaches Modell zur Transformation genutzt werden kann.

Die Primitive der DE-Domäne werden atomar, also im wechselseitigen Ausschluss abgearbeitet. Dieser Mechanismus wird von Petri-Netzen und Timed Automata zwar nicht direkt unterstützt, kann jedoch emuliert werden.

Kriterium	DE-Modell	Zeitbewertete Petri-Netze	Timed Automata	Formales DE-Modell
Ereignis-darstellung	Event-Objekt	Transition	Synchronisations-event	Event-Transition
Daten-Modellierung	Datenstruktur, komplex	Farben	globale Variable	Datenstruktur, skalar
verteilte Daten	shared Memory	globaler Platz	globale Variable	globale Variable
Ereignis-simulation	Event-Queue & Scheduler	Schalten von Transitionen	Synchronisation von Übergängen	Eventliste & Scheduler
Zeit-darstellung	absolut, globale Zeit	relativ, lokale Zeitvariable	relativ, lokale Zeitvariable	relativ, lokale Zeitvariable
Zeit-simulation	Scheduler	Elapse	Elapse	Scheduler/Elapse
Nebenläufige Ereignisse	ja	nein[45] / nur strukturell	nein/ nur strukturell	ja
Hierarchie/ Typ	ja/ kompositional	Toolabhängig/ funktional	ja (nur 2 Ebenen)/ kompositional	nein
Atomare Bearbeitung	ja/ Scheduler	nein/ emulierbar	nein/ emulierbar	ja/ Scheduler
Verifikation/ Sprache	nein/ –	ja[46]/ LTL, CTL	ja/ CTL	Prototyp[47]/ LTL

Tabelle 5.3.: Vergleich von Paradigmen zur Abbildung von DE-Modellen

Im den nachfolgenden Abschnitten werden die vorgeschlagenen Interpretationen nochmals im Detail vorgestellt.

[45] Zwar sind Plätze und Transitionen nebenläufig, allerdings ergibt sich eine andere Semantik, wenn der gemeinsame Vorplatz zweier zeitbehafteter Transitionen mehrere Marken enthält, so dass die Nebenläufigkeit hier strukturell aufgelöst werdem muss.

[46] Es existieren verschiedene Ansätze für verschiedene Netztypen

[47] Nutzt Umweg über Zustandsraum/Erreichbarkeitsgraphen als Petri-Netz.

Interpretation als zeitbewertetes Petri-Netz

In diesem Abschnitt wird die Interpretation von DE-Modellen in zeitbewerteten Petri-Netzen betrachtet, siehe auch Abschnitt 2.4.2. Dazu wird die im Werkzeug *TINA* [Ber06] verwendete Definition genutzt. Für die Bildung von Erreichbarkeitsklassen[48] wird jeder aktiven Transition eine zweistellige Zeitvariable zugeordnet, die beginnend mit dem Schaltintervall rückwärts läuft. Eine schaltfähige Transition muss spätestens dann schalten, wenn ihre obere Intervallgrenze 0 erreicht, es sei denn, ihr wird die Schaltfähigkeit entzogen. Zum Rücksetzen der Zeitvariablen muss der Transition kurzzeitig die Schaltfähigkeit entzogen werden. Das genutzte Werkzeug unterstützt keine Modularisierung oder Hierarchie. Über den Erreichbarkeitsgraphen ist eine Prüfung erreichbarer Zustandskonfigurationen oder Schaltreihenfolgen als LTL-Eigenschaften möglich[49].

Um Modelle der DE-Domäne als Petri-Netze darzustellen, werden Primitive (funktionale Teile) anders umgesetzt als Module (kompositionale Teile).

Events werden im Petri-Netz durch das Schalten von (herausgehobenen) Transitionen im kompositionalen Teil modelliert. Die kompositionalen Teile dienen zur Verknüpfung von Ausgabeevents und Eingangsevents. Dazu können entweder Eingangsevents oder-verknüpft werden (*Merge*) oder es kann eine Verteilung von Ausgabeevents auf mehrere Eingänge erfolgen (*Fork*). Ein Ausgang eines funktionalen Teils wird als Transition modelliert, und besitzt demnach die Semantik eines Ausgabeevents. Als Eingang eines funktionalen Teiles wird ein Platz genutzt. Alle dem Eingangsplatz vorgelagerten Transitionen besitzen genau eine ausgehende Kante. Das Schalten eines dieser vorgelagerten Transitionen besitzt daher die Semantik eines Eingangsevents.

Da die Primitive im wechselseitigen Ausschluss abgearbeitet werden müssen, wird im Petri-Netz ein globaler Semaphor benutzt, der beim Eintreten eines Eingangsevents dem entsprechenden Primitiv zugeordnet wird. Die Freigabe erfolgt nach erfolgter Abarbeitung des funktionalen Teils. Zeiten bzw. Zeitverzögerungen, die im DE-Modell global durch den Scheduler erzeugt werden, werden im Petri-Netz durch zusätzliche Elemente des funktionalen Teils nachgebildet, die einem Ausgabeplatz vorgelagert sind. Dazu kann ein Platz als Pseudoausgang verwendet werden, der über eine, die Verzögerung erzeugende, Transition mit dem eigentlichen Ausgabeplatz verbunden wird. Dabei tritt gleichzeitig ein Problem zu Tage. Können zwischen Erzeugung und Konsumierung des Ausgabeevents weitere Ausgabeevents erzeugt werden (nebenläufige Events), so ist eine komplexere Ersatzkonstruktion erforderlich, etwa durch parallele Transitionen. Dazu muss der Grad der Nebenläufigkeit bekannt sein.

[48] Zeitabhängige Erreichbarkeitsklassen aus der Analyse, siehe auch Abschnitte 2.4.2 und 2.6.1.
[49] Dazu kann der LTL-Model-Checker *SELT* aus der Tina Toolbox [Ber06] genutzt werden.

Interpretation als Timed Automata System

In diesem Abschnitt wird die Interpretation von DE-Modellen als Timed Automata betrachtet, wie sie im Werkzeug *Uppaal* [Dep] in den Versionen 3.4 bzw. 3.6 als so genannte *Uppaal Timed Automata* umgesetzt sind [BDL04, BY03], siehe auch Abschnitt 2.4.2. Dabei werden einzelne Prozesse als Timed Automata beschrieben, die sich über so genannte *Channels* synchronisieren und Daten über globale Variablen austauschen können. Prozesse werden als Templates beschrieben und können mittels Variablen[50] parametrisiert werden. Zeit wird durch Zeitvariablen gemessen, wobei Zeitvariablen immer aktiv sind und durch Aktionen nur auf konstante Werte gesetzt werden können[51].

Bei der Synchronisation werden zwei oder mehrere Transitionen derart synchronisiert, dass diese gleichzeitig (synchron) stattfinden. Dabei nimmt ein Prozess die sendende und der andere Prozess die empfangende Rolle ein. Es existieren mehrere Synchronisationstypen, für paarweise Synchronisation (Channel), priorisierte paarweise Synchronisation (Urgent Channel) und mehrfache Synchronisation (Broadcast Channel). Existieren bei der (priorisierten) paarweisen Synchronisation mehrere Sender bzw. Empfänger, so wird eine zufällige[52] Auswahl getroffen. Vor allem die normale Transition ist für die Nachbildung der Events eignet, wobei eine Aufteilung von Events (Fork) durch mehrere paarweise Synchronisation in nichtdeterministischer Reihenfolge modelliert werden muss. Die mehrfache Synchronisation ist nicht geeignet, da eine Synchronisation zwischen Sender und Empfänger nicht zwingend ist, also der sendende Prozess auch dann synchronisiert, wenn kein empfangsbereiter Prozess vorhanden ist.

Weiterhin existieren mehrere Zustandstypen (L), normale, zeitlose (Urgent) Zustände (L_u) und priorisierte (committed) zeitlose Zustände (L_c). Dabei gilt $L \cap L_u = L \cap L_c = L_u \cap L_c = \emptyset$. Zeitlose Zustände müssen zum selben Zeitpunkt verlassen werden, in dem sie betreten worden sind, allerdings besteht keine Priorisierung bezüglich der Transitionen anderer Zustände. Ausgehende Transitionen priorisierter Zustände werden hingegen priorisiert behandelt, wodurch eine Modellierung atomarer Abschnitte möglich ist.

Ein Automat muss sich zu jedem Zeitpunkt in einem definierten Zustand befinden. Die Automaten können nichtdeterministisch sein. Das Werkzeug *Uppaal* ermöglicht eine Prüfung der erreichbaren Zustandskonfigurationen mittels eingeschränkter CTL (Templates), allerdings wird keine explizite Analyse der Grenzen von Zeitvariablen ermöglicht[53].

Der Synchronisationsmechanismus der Timed Automata ermöglicht die Abbildung einer Struktur, die zu den Events der DE-Domäne äquivalent ist. Dabei wird das Auftreten eines

[50] Integer, Booleans werden als Basistypen unterstützt und können auch in Arrays oder Structs verwendet werden.

[51] In neueren Versionen besteht die Möglichkeit, Zeitvariablen Werte aus Integer-Variablen zuzuordnen.

[52] In neueren Versionen ist auch eine Priorisierung von Channels bzw. Prozessen möglich, was hier jedoch nicht weiter berücksichtigt wird.

[53] Zwar kann geprüft werden, ob eine bestimmte Zeitgrenze überschritten wird, jedoch nicht, wie groß der maximale Wert wirklich ist.

Events an seiner Quelle, durch das Senden auf einem Channel, und der Empfang, durch das empfangen auf einem Channel, modelliert. Ein Primitiv der DE-Domäne kann daher durch einen Timed Automata abgebildet werden. Eine Verzögerung eines Ereignisses durch die Primitive einer DE-Domäne kann im Automaten durch eine Verzögerung vor der Synchonisation modelliert werden. Für nebenläufige Events ist, ähnlich wie bei den Petri-Netzen, eine nebenläufige Struktur erforderlich, für den der Grad der Parallelität bekannt sein muss.

Um die atomare Abarbeitung von DE-Primitiven nachzubilden, muss die Logik (Zustandsübergangsfunktion) eines Primitivs nach der Synchronisation (empfangenes Event) mit dem priorisierten zeitlosen Zustandstypen (commited states) modelliert werden. Erfolgt eine Mehrfachsynchronisation, so ist zur Synchronisation das Zählen eingehender Events mittels einer Variable erforderlich. Sollen gleichzeitig[54] mehrere Events emittiert werden, so sind dafür zusätzliche zeitlose Zustände (urgent states) zwischen den einzelnen Synchronisationen erforderlich[55]. Enthält ein Primitiv eine Verzögerung, so ist das Primitiv zweigeteilt. Der erste Teil realisiert das Zustandsverhalten bzw. Zustandsübergangsverhalten. Der zweite Teil realisiert die Verzögerung und ist entsprechend der Nebenläufigkeit mehrfach ausgelegt. Enthält ein Primitiv keine Verzögerung, so kann der zweite Teil weggelassen werden, im Falle des fehlenden Zustandsverhaltens, wie beim *Delay*-Primitiv, der erste Teil.

Interpretation im formalen DE-Modell

In diesem Abschnitt wird die Interpretation von Modellen der DE-Domäne durch ein formales DE-Modell gemäß [PF07] diskutiert, siehe auch Anhang E. Ziel des genutzten formalen DE-Modells war die Unterstützung einer Ereigniswarteschlange, um nebenläufige Events abbilden zu können.

Im Formalen DE-Modell wird entsprechend der DE-Domäne eine Warteschlange für Ereignisse definiert. Gleichzeitig werden zustandsbehaftete Prozesse definiert, die ihren Zustand durch die Konsumierung von Ereignissen ändern können. Ein Primitiv wird dementsprechend als ein Prozess abgebildet. Die Zustandsautomaten der Prozesse besitzen dazu stabile und instabile Zustände, wobei die Bearbeitung eines Events mit dem Erreichen eines stabilen Zustandes beendet ist.[56] Dadurch kann sichergestellt werden, dass zu einem Zeitpunkt je nur ein Prozess aktiv ist. Verzögerungen werden als Intervalle an den emittierten Events notiert, so dass keine strukturelle Erweiterung erforderlich ist.

[54] Gleichzeitig im Sinne der Kausalität. Da Events nur nacheinander konsumiert werden, ändert eine sequentielle Emittierung die Semantik nicht.
[55] Hierbei ist auf eine nichtdeterministische Reihenfolge zu achten.
[56] Diese Semantik wird auch als Run-To-Completion Semantik bezeichnet.

Auswahl des Zielmodells

Für die Auswahl eines Zielmodells sind neben den theoretischen Erwägungen zur Ausdrucksmächtigkeit, siehe Tabelle 5.3, auch Aspekte der Transformation und Verifikation zu berücksichtigen.

Für die Realisierung und Bewertung des Transformation-basierten Ansatzes an sich ist dabei zu berücksichtigen, dass die Werkzeuge zur Verifikation des Zielmodell bereits erprobt sind und ggf. weiterentwickelt werden. Diese Voraussetzungen sind nur bei den zeitbewerteten Petri Netzen und den Timed Automata erfüllt.

Die Verifikation ist bei den Zeitbewertetn Petrinetzen nicht integraler Bestandteil des Werkzeugs *TINA*, insbesondere ist es erforderlich den Zustandsraum (Erreichbarkeitsgraphen) explizit und vollständig zu erzeugen. Bei den Timed Automata ist die Verifikation des Modells im Werkzeug *UPPAAL* derart integriert, das eine vollständige Erzeugung des Zustandsraumes nicht zwingend notwendig ist.

Für die Transformation relevant ist die Möglichkeit die Komposition eines Modells aus einfachen funktionalen Blöcken beschreiben zu können. Dies ist im Tool *UPPAAL* durch die Verwendung von Templates gegeben, während viele Petri Netz Werkzeuge eine direkte Spezifikation in nur einem netz erfordern.

Daher scheinen Timed Automata besonders geeignet zu sein, um bei der Realisierung des Transformation-basierten Ansatzes als Zielmodell verwendet zu werden.

5.3.3. Transformation der FSM Primitive

Für die Transformation ist entscheidend, wie das Verhalten spezifiziert ist, also welche Semantik bestimmten Modellelementen zu Grunde liegt, und wie die Modellelemente entsprechend im Zielmodell abgebildet werden können. Die FSM-Primitive (eingebettet in ein DE-Modell) nutzen eine Syntax und Semantik, die weitestgehend denen der UML-Statemachines entspricht [MLD07, OMG04]. Zustände dürfen hierarchisch sein, aber im Vergleich zu den UML-Statemachines keine Parallelität enthalten[57]. In Statemachines wird grundsätzlich zwischen Auswertung und Aktion unterschieden[58]. Bei der Auswertung, initiiert durch ein eingehendes Event, wird solange abwechselnd ein Folgezustand ermittelt und der entsprechende Zustandsübergang durchgeführt, bis ein stabiler Zustand erreicht ist, der erst durch eingehende Events (oder sich ändernde Conditions) verlassen werden kann. Beim Zustandsübergang werden Aktionen ausgeführt, die ggf. Memories verändern und/oder Events emittieren.

Ein Zustand kann jeweils verschiedene Aktionen enthalten: *Entry-Action*, *Do-Action*, und *Exit-Action*. Während *Entry-Action* und *Exit-Action* vergleichbar sind, unterscheidet sich die

[57] Eine mögliche Parallelität über Slave-Modelle wird hier nicht betrachtet.

[58] Wegen der Verwendung von C-Code für die Beschreibung von Conditions und Actions besteht prinzipiell die Möglichkeit diese Trennung zu umgehen, was hier jedoch nicht berücksichtigt wird.

Do-Action der FSM-Primitive. Hier muss dem betreffenden Zustand ein eigenes Slave-Modell hinterlegt werden, das dieselbe Interfacedefinition wie die FSM aufweist. Findet bei der Konsumierung eines Ereignisses im betreffenden Zustand kein Zustandswechsel statt, so wird das Slave-Modell als *Do-Action* abgearbeitet. Im Vergleich dazu ist die Abarbeitungsrate/Dauer der *Do-Action* in der UMLnicht spezifiziert. Zustandsübergänge werden durch *Trigger*, *Condition* und *Action* beschrieben. Als *Trigger* dürfen nur Events verwendet werden, wobei ein Event nur einmal konsumiert werden darf. *Conditions*, also Bedingungen stellen boolesche Ausdrücke über Variablen und Memories dar, wobei auch komplexe Funktionen verwendet werden dürfen. Alle Anschriften sind optional, eine fehlende Anschrift wird als immer erfüllt betrachtet. Wegen der Hierarchie kann es Zustandsübergänge zwischen Zuständen verschiedener Hierarchieebenen geben. Dann werden die verschiedenen *Actions* entsprechend der logischen Reihenfolge durchlaufen: zuerst alle *Exit-Actions* der verlassenen Zustände, dann die Action des betreffenden Zustandsüberganges und abschließend alle *Entry-Actions* der neu betretenen Zustände.

Die gewählten *Uppaal Timed Automata* bieten keine Möglichkeit hierarchische Zustandsautomaten direkt abzubilden. Daher muss die Hierarchie zunächst entsprechend aufgelöst werden. Ein entsprechendes Verfahren wird in [RS04] vorgestellt. Weitere Unterschiede bestehen bezüglich der Zeitmodellierung. Während Timed Automata lokale Zeitvariablen definieren, werden Zeitaspekte nicht in der FSM, sondern extern modelliert, so dass sich hieraus keine Einschränkungen ergeben. Die Verarbeitung eines Events durch die FSM erfolgt elementar unteilbar[59]. Bei der Transformation in Timed Automata muss diese elementare Bearbeitung durch konstruktive Maßnahmen sichergestellt werden, siehe dazu auch Abschnitt 5.3.2.

5.3.4. Transformation der Modellstruktur

Wie bereits erläutert wurde, muss die Transformation des Quellmodells in mehrere Schritte unterteilt werden. Bisher wurden die Möglichkeiten der Annotation sowie mögliche Interpretationen betrachtet. Zudem wurde analysiert, wie FSM-Primitive als formale Teile des Modells transformiert werden können. DE-Primitive selbst können, aufgrund der unterschiedlichen Abstraktion auf Modellebene nicht direkt, sondern nur über Annotationen transformiert werden. Offen ist also noch die Interpretation der Modellstruktur, die die Verknüpfung (Kommunikation) der einzelnen Teile (Primitive) untereinander beschreibt.

Gemäß der Interpretation in Timed Automata, siehe Abschnitt 5.3.2, werden Ports des Quellmodells durch Channels der Timed Automata modelliert. Verbindungen zwischen Ports werden dann auf Verknüpfungen der Channels abgebildet. Die Transformation wird dann in folgenden Schritten realisiert:

1. Normalisierung der Modellstruktur, siehe Abschnitt 5.2,
2. Hinzufügen impliziter Fork- und Mergeprimitive in die Struktur,

[59] Effekte durch Slave Modelle werden hier nicht berücksichtigt.

3. Zusammenfassen von Forks und sequentiellen Verknüpfungen als Optimierung der Modellstruktur,
4. Ersetzen von nicht realisierten Modellteilen (Inputchannels) durch Platzhalter[60],
5. Generierung des Zielmodells.

5.3.5. Eigenschaftsprüfung mit Timed Automata

Markierte Timed Automata (im Folgenden als Timed AR-Automata bezeichnet) eignen sich für die Repräsentation zeitbeschränkter temporaler Eigenschaften zum Zwecke der Eigenschaftsprüfung. Die grundlegende Idee wurde bereits in [PFSV05] vorgestellt, wobei die Eigenschaften als Templates in ein System von *Uppaal Timed Automata* eingebunden werden. Dabei wird ein markierter deterministischer Prüfautomat mit der automatenbasierten Systembeschreibung über Signale verknüpft. Anschließend kann über eine Erreichbarkeitsanalyse (CTL-basiert) die Erfüllung der Assertions geprüft werden. Als Markierung werden die drei Symbole S (Start), A (Accept) und R (Reject) verwendet.

Der nachfolgend beschriebene Ansatz geht davon aus, dass sich die zu prüfenden Eigenschaften in die Form einer Implikation bringen lassen, also aus *Prämisse* und *Konklusion* bestehen, wobei die Prämisse zeitlich vor der Konklusion steht, und die Eigenschaften somit eine kausale Abhängigkeit beschreiben. Ist dies der Fall, so ist eine Zerlegung der Eigenschaften wie folgt möglich: $\varphi = Prämisse \implies Konklusion = \neg Prämisse \vee (Prämisse \wedge Konklusion)$. Das bedeutet, dass immer zeitlich nach der Erfüllung der Prämisse die Konklusion erfüllt werden muss.

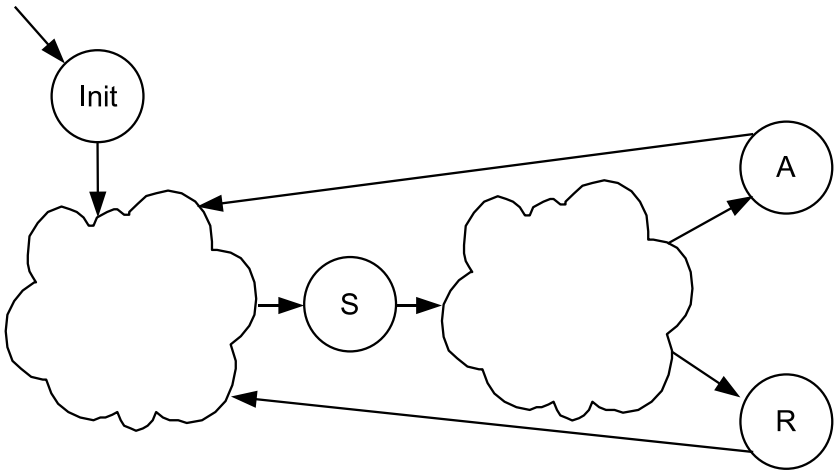


Abbildung 5.4.: Grundaufbau Timed AR-Automata

Abbildung 5.4 zeigt das Grundprinzip eines Timed AR-Automata. Der Automat beginnt

[60] Dies ist erforderlich, um eine mögliche Blockierung des Sendeprozesses zu vermeiden, siehe Abschnitt 5.3.2

mit einem Initialzustand, durch den ggf. Initialisierungen von Zeitvariablen realisiert werden. Wird die in der zu prüfenden Eigenschaft enthaltene Prämisse durch eine geeignete Struktur detektiert, so wird der mit S markierte Zustand erreicht. Daraufhin wird durch eine geeignete Struktur die Konklusion detektiert und schließlich entweder der Zustand A oder R erreicht. Anschließend wird wieder der Initialzustand eingenommen, so dass erneut eine Prämisse detektiert werden kann. Die markierten Zustände sind transient, werden also sofort wieder verlassen.

S beschreibt demnach das Erfüllen einer Initialbedingung. Anschließend lassen sich für deadlockfreie Systeme die Erreichbarkeitsklassen nach Tabelle 5.4 finden[61]. In der ersten Spalte ist die Nummer der Belegung von Erreichbarkeitsklassen (Erfüllbarkeitsklasse) notiert. In den nachfolgenden Spalten sind die Erreichbarkeiten für die Zustände S (EFS), A (EFA), R (EFR) sowie die Implikation (G(S$->$ FA)) in temporaler Logik notiert. Ohne erreichte Startbedingung ist die Erfüllbarkeit einer Assertion nicht entscheidbar (Erfüllbarkeitsklasse 1). Wird S erreicht, so lässt sich anhand des Erreichens von A und R die Erfüllung der Assertion entscheiden (Erfüllbarkeitsklasse 4-6). Anhand der Implikation (G(S -> F A))) lässt sich die Existenz unendlicher Pfade prüfen (Klassen 4 und 5). Ist nach Erfüllung der Startbedingung weder A noch R erreichbar, so liegt ein ungültiger Automat vor, oder aber es tritt ein Deadlock auf. Weitere Erreichbarkeitsklassen existieren für die betrachteten Prüfautomaten nicht[62].

Klasse	EF S	EF A	EF R	G(S -> F A))	Bedeutung
1	False	False	False	True	nicht gestartet
2	True	False	True	False	nie erfüllt
3	True	True	True	False	manchmal erfüllt
4	True	True	False	True	immer erfüllt
5	True	True	False	False	unendlicher Pfad
6	True	False	False	False	ungültiger Automat

Tabelle 5.4.: Mögliche Erreichbarkeitsklassen von Timed AR-Automata

Existieren nebenläufige Aktivierungen der Assertion, so besteht allgemein das Problem, dass mehrere Instanzen der Eigenschaft geprüft werden müssen. Wird ein formales Verfahren angewandt, so kann die Erkennung der Startbedingung nichtdeterministisch erfolgen, um eine vollständige Prüfung zu gewährleisten[63]. Alternativ ist bei der Kenntnis des Grades der Nebenläufigkeit auch die Nutzung eines komplexeren Automaten möglich, der mehrere Instanzen von Vor- und Nachbedingungen abbilden kann. Vorteilhaft für die formale Prüfung ist, dass die Simulation das Timed AR-Automata automatisch abläuft.

[61] Ein Deadlock bedeutet, dass eine Simulation ohne gültigen Folgezustand existiert.

[62] Ein Erreichen der Zustände A und R setzte das Erreichen des Zustandes S voraus (7 Belegungen). Die Implikation kann nach Erreichen von S nicht ohne Erreichen von A erfüllt werden (2 Belegungen). Theoretisch besteht beim deterministischen Prüfautomat die Möglichkeit S, A, R und die Implikation zu erfüllen. Dies wird durch eine nichtdeterministische Auslegung des Prüfautomaten vermieden (1 Belegung).

[63] Dabei wird nach der Erfüllung der Startbedingung entweder der Zustand S erreicht und anschließend die Assertion geprüft, es wird einfach die nächste Startbedingung abgewartet.

5.4. Schlussfolgerungen zur Transformation in Timed Automata

Der in Abschnitt 5.1 skizzierte Ansatz wurde sukzessive verfeinert. Dazu wurde zunächst der strukturelle Aufbau kompositional hierarchischer Modelle am Beispiel des Entwicklungssystems MLDesigner analysiertund Ansätze zur Normaisierung betrachtet, siehe Abschnitt 5.2. Die dabei gewonnenen Erkenntnisse, vor allem über die interne strukturelle Verknüpfung Teilmodellen dient als Ausgangspunkt der weiteren Betrachtungen. Hervorzuheben ist dabei, dass nach der Normalisierung eine nicht hierarchisches Modell vorliegt (2 Ebenen), bei denen die Funktionalität durch Blöcke mit definierten Schnittstellen gekapselt wird, so dass sich die Gesamtfunktionalität des Modells aus der einfachen Verknüpfung ergibt.

Die Art und Weise der Beschreibung der Funktionalitäten innerhalb von Blöcken ist nicht direkt für eine formale Verifikation verwendbar, da die resultierenden Modelle zu komplex für das Model Checking wären. Daher besteht die Notwendigkeit, Informationen über die abstrakte Funktionalität verfügbar zu machen. Dazu wurden in Abschnitt 5.3.1 verschiedene Möglichkeiten der Spezifikation betrachtet.

Die Wahl eines geeignetes Zielmodells für die Durchführung der Verifikation und die Realisierung der Transformation wird in Abschnitt 5.3.2 betrachtet. Dabei wurden verschiedene Zielmodelle betrachtet und bezüglich der Eignung analysiert. Die Wahl fiel schliesslich auf die Uppaal Timed Automata.

Die eigentliche Transformation in das Zielmodell wurde in den Abschnitten 5.3.3 und 5.3.4 betrachtet. Die Vorgehensweise zur Verifikation wurde schließlich in 5.3.5 vorgestellt.

Somit wurden alle Aspekte untersucht, die für die Realisierung eines Transformationbasierten Ansatzes relevant sind. Offen geblieben ist bisher die Realisierung des Transformationbasierten Ansatzes sowie dessen Validierung.

5.5. Prototyp zur Transformation-basierten Verifikation

Das im Abschnitt 5.3 dargestellte Verfahren ist in einem Prototyp *MLD2UPPAAL* umgesetzt worden [Mül06, MPF07], der die in diesem Kapitel bisher betrachteten Konzepte umgesetzt. In diesem Abschnitt erfolgt Betrachtungen zur Realisierung des Prototyps. Die Entwicklung erfolgte in JAVA™.

5.5.1. Architektur

Abbildung 5.5 zeigt die Architektur des entstandenen Prototyps. Man erkennt, dass die Transformation vom Quellmodell in das Zielmodell in mehreren Phasen abläuft. Dem liegen zwei prinzipielle Überlegungen zu Grunde. Einerseits lässt sich die Transformation in mehrere Teilschritte untergliedern, siehe auch Abschnitt 5.1, so dass sich durch speziell angepasste Modelle

eine bessere Kapselung einzelner Teile erreichen lässt. Andererseits besteht die Möglichkeit, unterschiedliche Quell- und Zielmodelle einzubinden, um so eine größere Flexibilität zu erreichen.

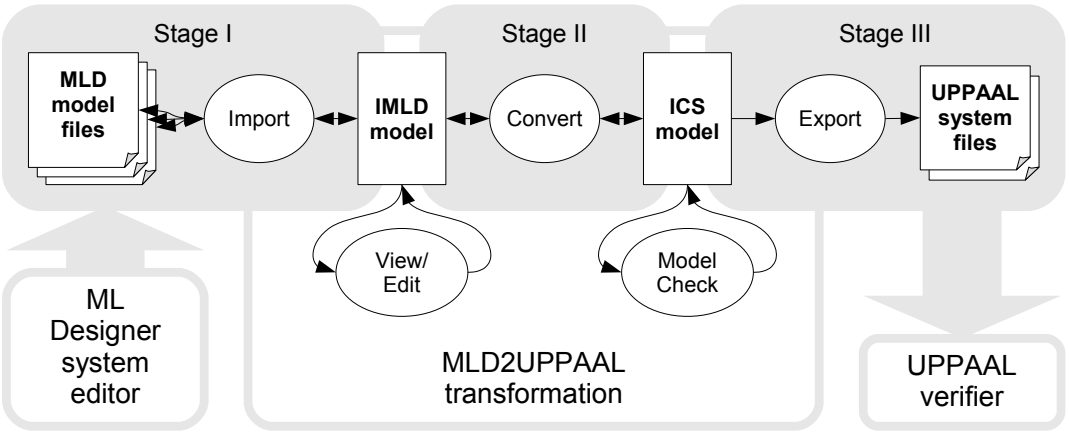


Abbildung 5.5.: Architektur des Prototyps zur Transformation-basierten Verifikation

In der ersten Phase wird das Quellmodell importiert. Da das MLD-Modell in einzelne Klassen zerlegt gespeichert wird, also Definitionen von Elementen und Subsystemen separat gehalten werden, ist es zunächst erforderlich, das Quellmodell einzulesen und unter Beibehaltung der hierarchischen Struktur zusammenzusetzen. Das resultierende *Intermediate MLD Modell* (IMLD Modell) enthält die komplette Struktur des Originalmodells sowie die Constraints mit einer entsprechenden Zuordnung zum Originalmodell.

In der zweiten Phase wird das Constraint-Modell erzeugt. Dazu werden die Annotationen in einem *Intermediate Constraint System Modell* (ICSM Modell) instantiiert. Dazu wird das hierarchische Ausgangsmodell in ein flaches Modell überführt und die Constraints der einzelnen Blöcke werden zusammengesetzt. Dabei können Signale zusammengefasst werden, die sich aufgrund der Hierarchie zuvor auf unterschiedlichen Hierarchieebenen befunden haben.

In der dritten Phase wird das Zielmodell erstellt. Dabei werden die Struktur und das mittels der Annotationen spezifizierte Verhalten ins Zielmodell (Uppaal Timed Automata) überführt. Während die Struktur des Zwischenmodells erhalten bleibt, wird das Verhalten der Annotationen mittels Templates generiert.

5.5.2. Metamodelle und Templates

Bei der Realisierung der einzelnen Phasen wurde darauf geachtet, dass sich die Modelle möglichst einfach durch Konfiguration anpassen lassen. Dazu wurde ein Framework entwickelt, mit dem eine dynamische Modelltransformation vergleichsweise einfach umgesetzt werden kann. Die interne Repräsentation der Zwischenmodelle wurde mittels EMF-Metamodellen [Bud03] spezifiziert. Dadurch lassen sich die verwendeten Datenstrukturen anpassen, so dass Änderungen im Quellcode durch Codegenerierung weitestgehend automatisch propagiert werden können.

Die Transformation selbst erfolgt regelbasiert, bezogen auf die definierten Metamodelle. Dazu wird das *Model Transformation Framework* (MTF) [IBM07, DGS05] von IBM genutzt. Dadurch wird erreicht, dass Transformationsregeln und Modelle zunächst entkoppelt sind und sich die Regeln vergleichsweise einfach anpassen lassen. Gleichzeitig bleiben die Referenzen zwischen den transformierten Modellen zur Laufzeit des Prototyps, also während der Transformation, erhalten, so dass eine Rückgabe der Ergebnisse prinzipiell möglich ist.

5.5.3. Constraints und Templates

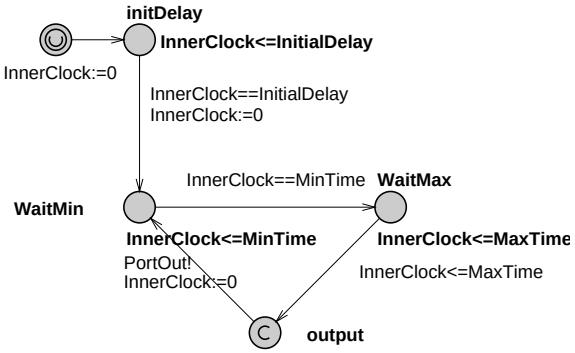
Die semantische Verbindung zwischen Quell- und Zielmodell wird durch die Constraints realisiert. Grundidee ist die Beschreibung der Eigenschaften im Quellmodell und deren Generierung im Zielmodell. Grundlegend ist dabei die Unterscheidung zwischen zugesicherten und geforderten Eigenschaften, siehe Abschnitt 4.6.

Die Sprache zur Beschreibung der Constraints ist mittels EBNF realisiert. Dabei wird die Sprache in mehrere semantische Ebenen unterteilt. Die äußerste Ebene ist äquivalent zum Verification Layer. Dadurch wird die Nutzung der Eigenschaften als Zusicherung oder Anforderung beschrieben. Die mittlere Ebene, der Modeling Layer, definiert, wie die einzelnen Signale des Quellmodells verwendet werden können, also in welcher Form Signale und Parameter in den Constraints verwendet werden können. Die innerste Ebene beschreibt Funktionalität in Form von Templates, eine Unterteilung in Boolean Layer in Temporal Layer findet hierzu nicht statt. Die Sprachdefinition ist im Anhang A.1 zu finden

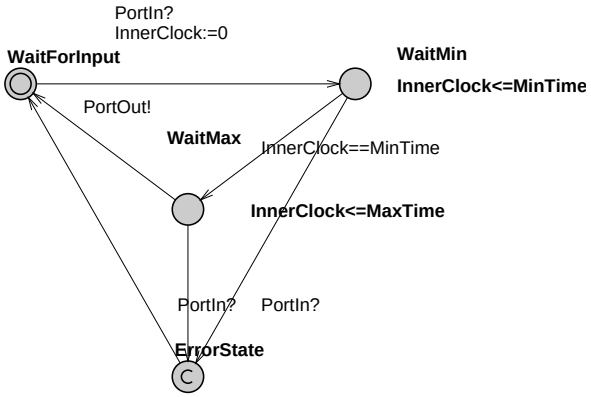
Während das Framework zur Transformation nur Verification Layer und Modeling Layer benötigt, um die Verwendung der Constraints zu prüfen, erfolgt erst in der dritten Phase, der Generierung, die Auswertung der Templates. Dazu wird entsprechend dem Verfication Layer entweder ein Automat generiert, der die zugesicherte Eigenschaft realisiert, oder ein Automat mitsamt den Erreichbarkeitsanfragen erstellt, der die geforderten Eigenschaften prüft. Die Prüfanfragen werden entsprechend Tabelle 5.4 erzeugt. Da nur Eigenschaften der Klasse 6 (Eigenschaft immer erfüllt) geprüft werden, genügt die Erzeugung der Anfragen $EF\text{S}$ und $G(\text{S} -> F\text{A})$.

Die Realisierung der Templates erfolgt über einen Compiler-Generator durch die Attributierung der Grammatik der Constraint Language. Während die Grammatik die Parameter und die Variabilität der Templates beschreibt, wird durch die attributierte Grammatik die Semantik der Templates beschrieben. Dabei werden die Automaten entsprechend der Variabilität des konkreten Ausdruckes erzeugt und die Parameter instantiiert. Änderungen und Erweiterungen der Templates können daher über die Anpassung der attributierten Grammatik erfolgen.

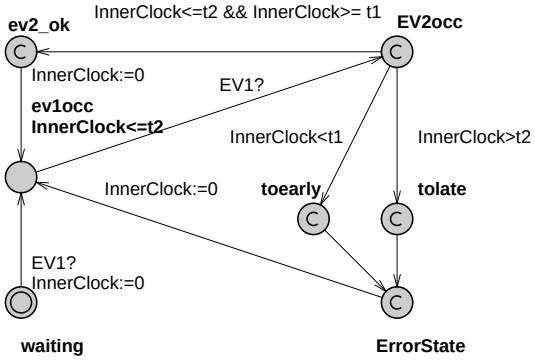
Abbildung 5.6 zeigt drei Basistemplates für die Transformation ohne instantiierte Parameter. Abbildung 5.6a zeigt das Template für eine Eventquelle, Abbildung 5.6b das Template für einen Prozess und Abbildung 5.6c das Template zur Prüfung zyklischer Events.



(a) Source Template



(b) Processing Template



(c) Constraint Template

Abbildung 5.6.: Templates für die Transformation

5.5.4. Bedienung des Prototyps

Der Prototyp *MLD2UPPAAL*, siehe auch Anhang B, nutzt ein komplettes MLD-Modell basierend auf den beschreibenden *.mml-Dateien als Ausgangspunkt für die Analyse. Um Konflikte mit der Entwicklungsumgebung *MLDesigner* zu vermeiden, wird *MLD2UPPAAL* extern gestartet.

Zur Verwendung ist einmalig eine Konfiguration erforderlich. Dabei werden die Verzeichnispfade für *MLDesigner*, *Uppaal* sowie für die Quell- und Zieldateien konfiguriert. Weiterhin können versionsabhängige Optionen konfiguriert werden.

Die Nutzung erfolgt in drei Schritten:

- **Eingabe und Strukturanalyse:** Nach der Auswahl des zu prüfenden Modells und dem Import des Quellmodells besteht die Möglichkeit, erkannte Syntaxfehler es editieren und in das Modell zurückzuschreiben.

- **Transformation:** Durch Optionen kann die Generierung von Anfragen für die Implementierungen und FSMs unterdrückt werden. Weiterhin besteht die Möglichkeit das zu

generierende Modell strukturell zu optimieren, indem die Generierung ungenutzter Teile unterdrückt wird. Nach der Generierung steht das Uppaal-Modell in der gewünschten Version zur Verfügung.

- **Verifikation:** Zur Verifikation bestehen zwei Möglichkeiten. Es kann das erzeugte Modell entweder mit *Uppaal* direkt geöffnet und dort die generierten Anfragen verifiziert werden. Alternativ besteht die Möglichkeit den Verifier direkt zu verwenden und die Eigenschaften direkt über das Werkzeug zu prüfen.

Eine detaillierte Beschreibung zur Nutzung des Prototyps ist im Anhang B zu finden.

Das Modell ist bezüglich der annotierten Eigenschaften korrekt, wenn alle Prüfanfragen erfüllt worden sind, also alle geforderten Eigenschaften immer gelten und alle zugesicherten Eigenschaften korrekt umgesetzt worden sind. Durch die Zuordnung der Prüfanfragen zu den Eigenschaften ist es möglich, die Fehlerursache schrittweise einzugrenzen. Tabelle 5.5 beschreibt Fehlertypen, Ursachen und Mögliche Maßnahmen. Die Behebung der Fehlerursachen und erneute Verifikation werden solange durchgeführt, bis kein Fehler mehr festgestellt wird. Dann handelt es sich um ein bezüglich Struktur, Zusicherungen und Anforderungen korrektes Modell.

Nr.	Fehlertyp	Ursache	Maßnahme
1	Deadlock	Modellstruktur unvollständig	Modellstruktur prüfen
2	Fehler in den Zusicherungen	Beschränkung der Zusicherung nicht eingehalten.	Fehlerpfad auswerten: Fehlerhafte Eigenschaft, falsche Parameter oder fehlende Eigenschaften in anderen Komponenten des Modells.
3	Fehlendes Startevent bei Eigenschaften	Modellstruktur unvollständig	Modell prüfen: Fehlende oder fehlerhafte Eigenschaften in anderen Komponenten des Modells (Zusicherungen)
4	Fehler in einer Implikation	Eigenschaft nicht erfüllt	Modell prüfen: Modellstruktur unvollständig oder Parameter fehlerhaft Eigenschaft nicht erfüllt.

Tabelle 5.5.: Fehlersuche im Modell

5.6. Validierung der Methode zur Transformation-basierten Verifikation am Beispiel

Das Beispiel, das im Anschnitt 5.6.1 vorgestellt wird stammt aus [Lic04] und stellt ein bezüglich der zeitlichen Anforderungen relativ simples Beispiel dar, da im Wesentlichen nur sequentielle Eigenschaften modelliert werden. Trotzdem ist es zur Demonstration der prinzipiellen Umsetzbarkeit des Gesamtkonzepts geeignet, auch weil es in Form eines komplexen UML Modells in einer weiteren Form vorliegt. In [Lic04] wurde eine objektorientierte Modellierungsmethodik

für ein Automatisierungssystem entwickelt. Automatisierungssysteme enthalten im Wesentlichen vernetzte Steuerungen, die durch eine übergeordnete Kommunikationsschicht (Prozessleitebene) administriert werden können. Die vernetzten Steuerungen stellen Echtzeitsysteme mit harten Echtzeitforderungen dar. Ein systematisch bedingter Fehler ist nicht tolerierbar und daher mit formalen Methoden auszuschließen. Diese Anforderungen gelten auch für viele Anwendungen von eingebetteten Rechnersystemen, die zunehmend als Systems-on-Chip realisiert werden. Deshalb kann hier das Beispiel aus der Dissertation [Lic04] verwendet werden, bei dem das System mit seinen Echtzeitforderungen mittels prozessorientierten UML Diagrammen modelliert wurde. Diese Beschreibung stellt also die Verknüpfung von vernetzten Steuergeräten, potentielle SoCs, dar, die intern entsprechend ihrer Realisierung weiter verfeinert sein können. Anschließend erfolgte eine manuelle Transformation in ein Realzeitautomatenmodell des Tools *Uppaal*.

5.6.1. Beispiel Prozessleitsystem

Im Prozessmodell werden Rohteile einem Lager entnommen, transportiert, ggf. in mehreren Schritten bearbeitet und abschließend wieder dem Lager zugeführt. Sämtliche Schritte sind zeitbehaftet und lassen sich (teilweise) im Sinne einer hierarchischen Modellierung weiter zerlegen. Das Modell eignet sich also gut, um verschiedene Aspekte der Methodik (abstraktes Modell, Kontrollflusseigenschaften, Hierarchie/Verfeinerungen) zu untersuchen. Ein genereller Kritikpunkt kann jedoch gefunden werden: das Modell ist rein sequentiell angelegt, eine parallele Bearbeitung verschiedener Werkstücke wird nicht modelliert. Das Modell spiegelt also nicht alle industriellen Erfordernisse wieder und ist bezüglich seines Verifikationsumfanges nur wenig komplex.

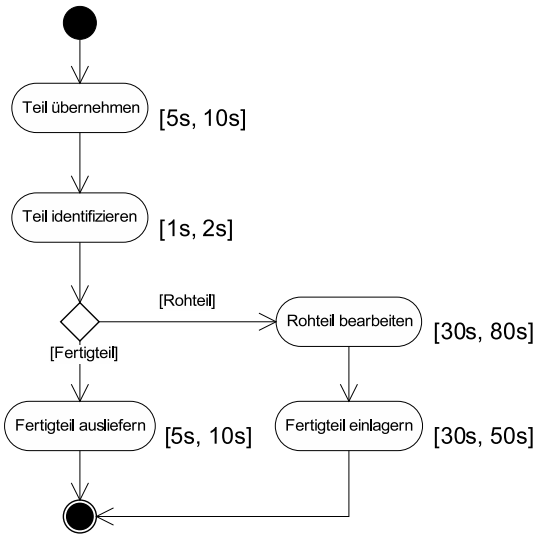


Abbildung 5.7.: Aktivitätsdiagramm „Fertigen“

Das Ausgangsmodell ist mittels UML Verhaltensdiagrammen, Aktivitätsdiagramm, Sequenzdiagramm, Zustandsdiagramm modelliert und mit Intervallbeschreibungen ergänzt worden. Der Zusammenhang zwischen den einzelnen Diagrammen wird über die in [Lic04] entwickelte Methodik hergestellt, wobei Verfeinerungen der Eigenschaften in der Reihenfolge Aktivitätsdiagramm, Sequenzdiagramm, Zustandsdiagramm stattfinden. Das übergeordnete Diagramm für den Prozess *Fertigen* ist in Abbildung 5.7 dargestellt. Es besteht aus den Prozessen *Teil übernehmen*, *Teil identifizieren*, *Rohteil bearbeiten*, *Fertigteil ausliefern* und *Fertigteil einlagern*. Die Funktionalitäten *Teil übernehmen* (Abbildung 5.8) und *Rohteil bearbeiten* (Abbildung 5.9) sind in Sequenzdiagrammen dargestellt. Die Funktionalität des Bearbeitungsgreifers wird schließlich in einem Zustandsdiagramm (Abbildung 5.10) weiter verfeinert.

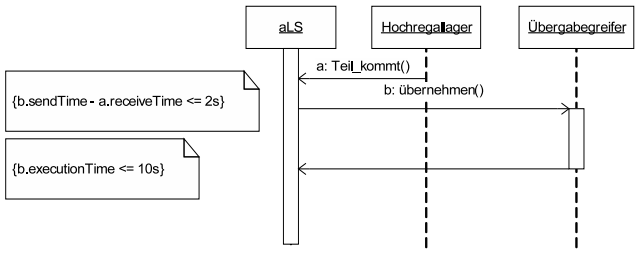


Abbildung 5.8.: Sequenzdiagramm „Teil übernehmen“

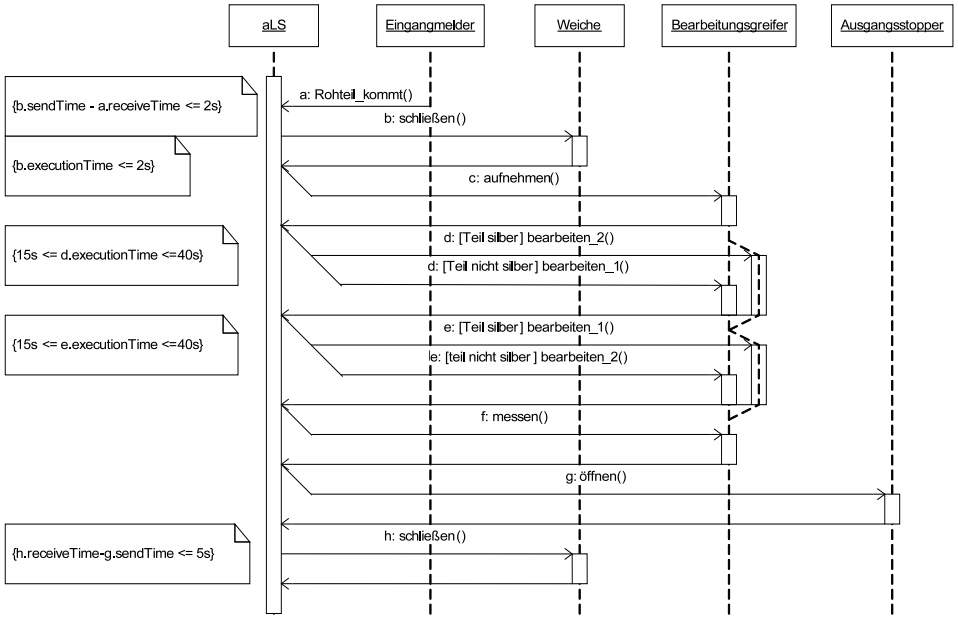


Abbildung 5.9.: Sequenzdiagramm „Rohteil bearbeiten“

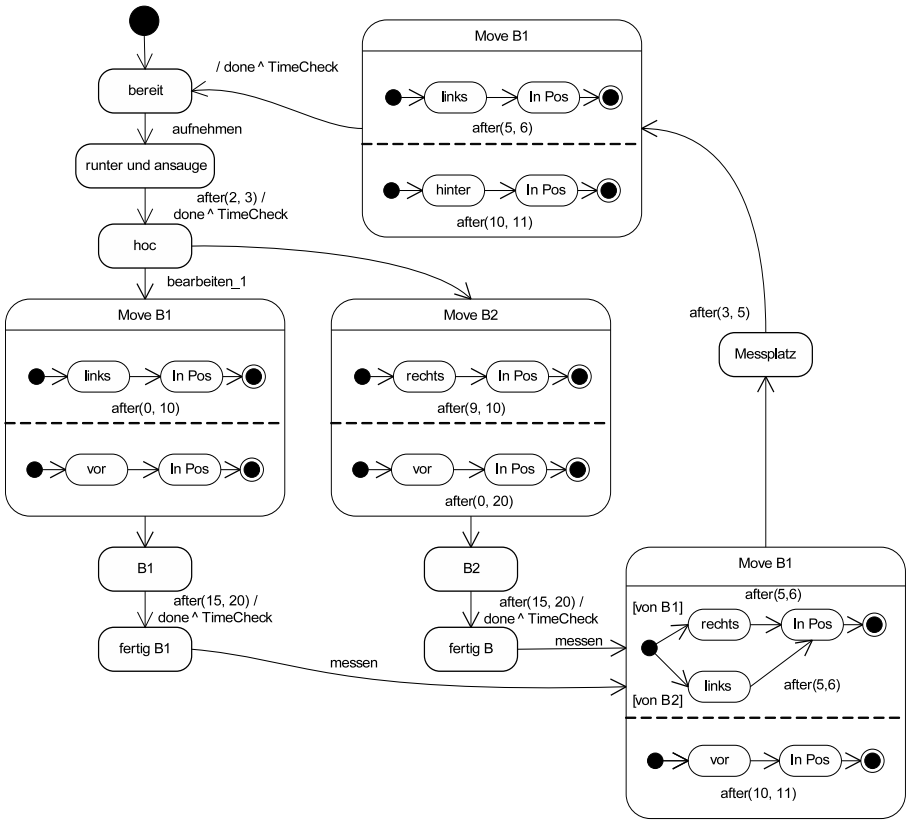


Abbildung 5.10.: Zustandsdiagramm „Bearbeitungsgreifer“

5.6.2. Umsetzung in MLDesigner

Die Umsetzung des Modells erfolgte in MLDesigner [MLD07], einem Modellierungswerkzeug, das eine strukturelle und funktionale Modellierung bzw. Simulation erlaubt. Innerhalb dieses Modellierungstools wird die Domäne DE verwendet, die eine entsprechende Eventsemantik besitzt. Um den zeitlichen Verlauf auch simulieren zu können, wurden Delay-Primitive verwendet. Der Fluss des zu bearbeitenden Teils durch das Automatisierungssystem wird durch den Eventflow nachgebildet. Die Notation der Eigenschaften selbst erfolgte durch Notation der entsprechenden Eigenschaft als strukturierter Text (kontextfreie Grammatik). Benutzt wurden dabei folgende Ausdrücke:

- A) `impxor property always` *InEvent* `in` $[t_{min}, t_{max}]$ `produce` *OutEvent*; Definition der Zeitdauer eines Prozesses.
- B) `impxor property always` *Inevent* `in` $[t_{min}, t_{max}]$ `produce` *OutEvent1* `or` *OutEvent2*; Definition der Zeitdauer eines Prozesses sowie alternativer Verzweigungen.
- C) `impxor property always` *InEvent1* `and` *InEvent2* `in` $[t_{min}, t_{max}]$ `produce` *OutEvent*;

Definition der Zeitdauer eines Prozesses sowie der Eingangssynchronisation (Parallelsynchronisation).

- D) `impxor property every` $[t_{min}, t_{max}]$ `produce` *OutEvent*; Definition einer Eventquelle.

Parallelverzweigung (Fork) und Alternativzusammenführung (Merge) werden in MLD bereits intern abgebildet, so dass keine speziellen Elemente dafür verwendet wurden.

Im Originalmodell enthält das Sequenzdiagramm sowohl die Ansteuerfunktionen als auch die Spezifikation der Zeitdauer, die für einzelne Funktionen des Zustandsdiagramms erlaubt sind. Während die Abbildung der Aktivierungsfunktion zwischen Sequenzdiagramm und Zustandsdiagramm eindeutig ist, gestaltet sich die Bestimmung der korrekten Rückgabefunktion als schwierig. Eine geeignete Realisierungsmöglichkeit besteht in der parallelen Komposition von Sequenzdiagramm in der Domäne DE und dem Statechart in der Domäne FSM.

Der Teilprozess *Fertigen* aus dem Aktivitätsdiagramm wurde in ein DE Modell gemäß Abbildung 5.11 rechts überführt. Für den Zeitverbrauch wurden Ausdrücke des Typs *A* hinzugefügt, für die Verzweigung wurde entsprechend ein Ausdruck des Typs B verwendet. Die Ausdrücke stellen ergänzende Notationen für die Blöcke dar, wobei die entsprechenden Zeitangaben für das Intervall eingesetzt wurden. Für die Beschreibung der Events wurde Bezug auf die entsprechenden lokalen Ports der Komponenten genommen, da hierdurch eine Extraktion der Modellstruktur möglich ist. Auf der linken Seite von Abbildung 5.11 ist die für das Eventmodell notwendige Umgebung realisiert. Ein Ausdruck des Typs *E* dient als Eventquelle, wobei das Zeitintervall größer als die Bearbeitungszeit gewählt werden muss. Ein Ausdruck des Typs *F* dient als Eventsenke im Modell. Das Teilmodell *Fertigen* ist in dieses Umgebungsmodell eingebettet. Die Sequenzdiagramme *Teil übernehmen* und *Rohteil bearbeiten* wurden entsprechen

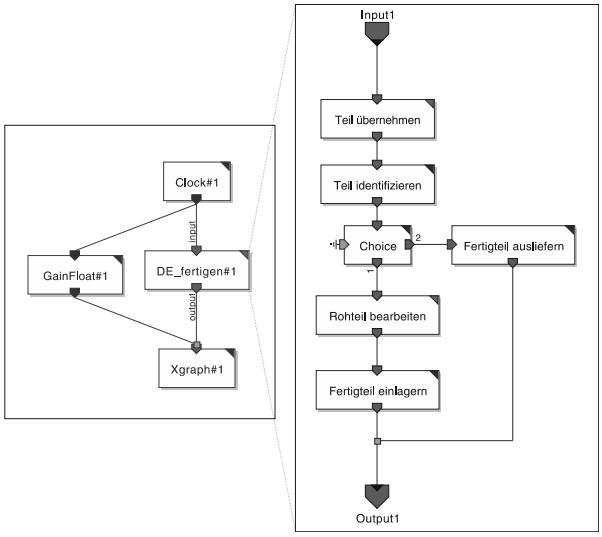


Abbildung 5.11.: Teilprozess „Fertigen“ und Umgebungsmodell

Abbildung 5.12 in DE Modelle umgesetzt, Prozesszeiten mit Ausdrücken des Typs A und Verzweigungen mit Ausdrücken des Typs B genauer spezifiziert. Die gewählte Umsetzung ist stark an den Kontrollfluss angelehnt. Eine Abbildung von Objektzugehörigkeiten wäre durch Kapselung in entsprechende Modellteile möglich, die Übersichtlichkeit würde jedoch stark darunter leiden.

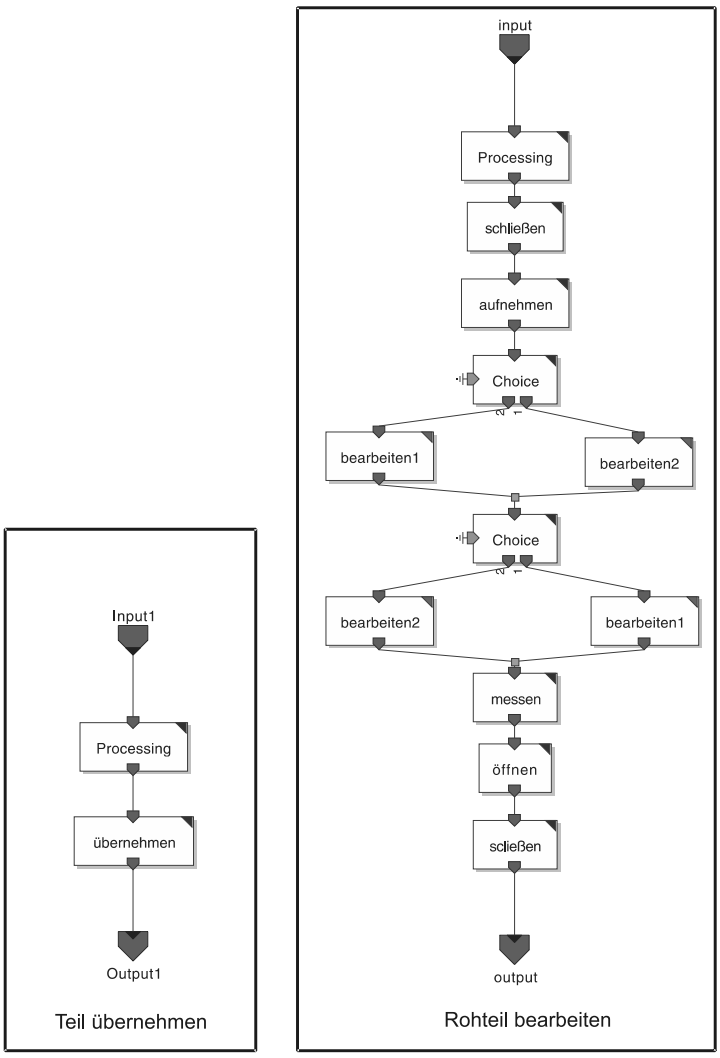


Abbildung 5.12.: DE Modelle „Teil übernehmen“ und „Rohteil bearbeiten“

Aus dem Zustandsdiagramm *Bearbeitungsgreifer* wurde die Funktion *Bearbeiten*, wie in Abbildung 5.13 dargestellt, modelliert.

Bei der Transformation in *Uppaal Timed Automata* wurden die elementaren Funktionen sowie die Verbindungsstrukturen durch Templates generiert. Je nach transformierter Modelltiefe wurden die abstrakteren Zeitbeschränkungen durch Anforderungstemplates und die konkreteren Zeitbeschränkungen durch Eigenschaftstemplates umgesetzt. In Abbildung 5.14 sind die

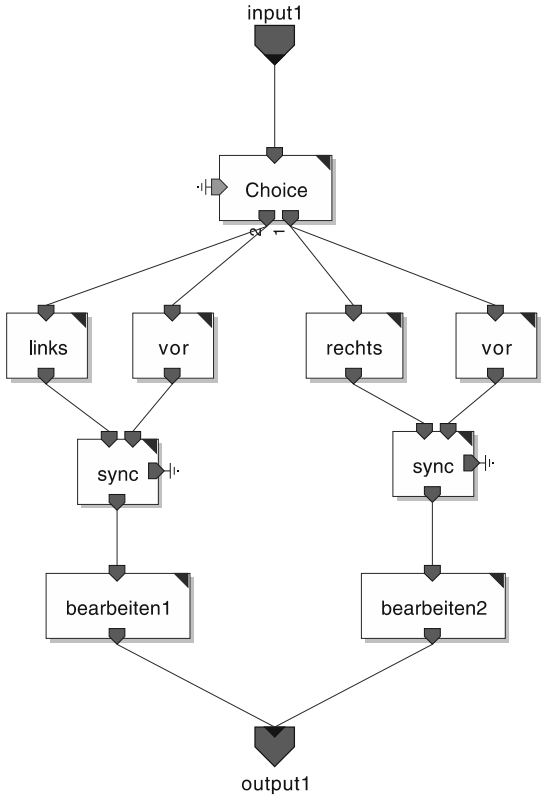
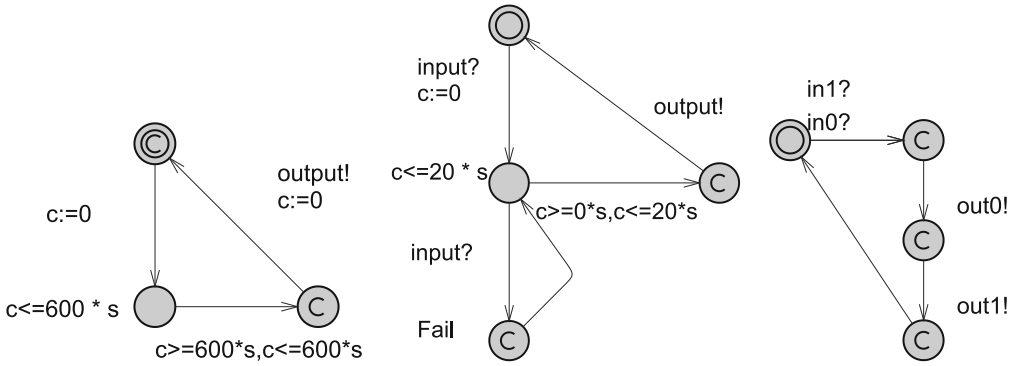


Abbildung 5.13.: Die Funktion „Bearbeiten“

verwendeten Basistemplates für (von links nach rechts) Eventgenerierung (Source), Verzögerung (Processing) und Relation dargestellt. Das Template für die Eventgenerierung besitzt keine Eingangssignale. Ausgabesignale von committed States besitzen keine Guards. Die Invariante des Wartezustandes lässt ein Schalten der Ausgabetransition zu. Das Verzögerungstemplate kann bei beiden zeitbehafteten Zuständen Events (`input?`) empfangen. Im Falle des unteren Zustandes wird dabei das Maximum der gleichzeitig bearbeitbaren Events (=1) überschritten, der Fehlerzustand (`Fail`) wird markiert. Entsprechend wird eine Prüfanfrage `A[] !Fail` generiert, um den Fahler zu prüfen. Das Relationtemplate besitzt keine Zeitfunktion. Eingangsevents sind disjunkt und führen jeweils zur Erzeugung sämtlicher Ausgabeevents. In dem zeitbehafteten Zustand können sämtliche Events empfangen werden, Guards existieren nicht.

Abbildung 5.15 zeigt zwei Templates für die Eventsynchronisation. Die Synchronisation wird dabei mittels einer binär kodierten Integervariablen (`cnt`) geprüft. Im linken Automat können im Zustand rechts unten keine Events empfangen werden, die Guards an committed States sind jedoch vollständig. Dies kann zu einer fehlerhaften Abbildung der Funktion führen, etwa wenn sich die Transition eines zeitbehafteten Zustandes aufgrund seiner Zeitinvariante mit dem Eingangsevent synchronisieren will und dadurch den Übergang verhindert. Dies führt dazu, dass die Synchronisation verzögert wird, ohne dass die Zeitvariable des durch die Invariante be-



(a) Eventgenerierung (b) Verzögerung (c) Relation

Abbildung 5.14.: Basistemplates für das Beispiel

schränkten Zustandes aktualisiert wird - die Zeiten laufen dann nicht mehr synchron. Im rechten Automat ist dieser Mangel behoben worden, es wurde ein Fehlerzustand eingefügt. Weiterhin wurde für den Initialzustand eine Prüfung eingefügt, so dass bei einer n-Synchronisation genau n verschiedene Ereignisse eine Synchronisation auslösen, pro Ereignistyp innerhalb einer Synchronisationsphase also nur noch ein Ereignis erlaubt ist.

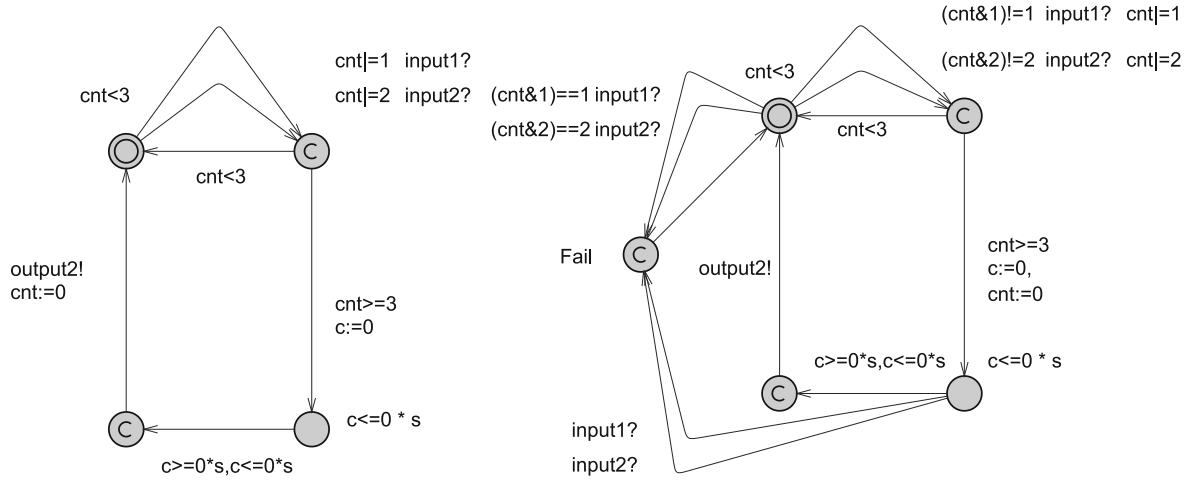


Abbildung 5.15.: Synchronisation

Abbildung 5.16 zeigt den für die Prüfung verwendeten Automaten. Er wartet auf ein Ereignis (`input?`) und überprüft dann die Zeitspanne bis zum Auftreten des zweiten Ereignisses (`output?`). In den zeitbehafteten Zuständen können jeweils beide Events empfangen werden. Beim Prüfzustand führen das Überschreiten der maximalen Zeitdauer sowie das Empfangen eines zweiten Ereignisses des Typs1 zur Markierung des Fehlerzustandes (`Fail`). Ansonsten wird nach dem Empfangen des zweiten Events die Korrektheit der Zeitspanne ausgewertet. Die Ausgabetransitionen erfüllen das Kriterium der Vollständigkeit. In der Prüfanfrage werden die

folgenden Fälle entsprechend Tabelle 5.4 überprüft:

- `E<> Ev1occ`
- `A[] !Fail`
- `Ev1occ -> Ok`
- `(A[] (Ev1occ -> EF Ok))`

Dabei nimmt `Ev1occ` die Rolle des Start-Zustandes, `Fail` die Rolle des Reject-Zustandes und `Ok` die Rolle des Accept-Zustandes an.

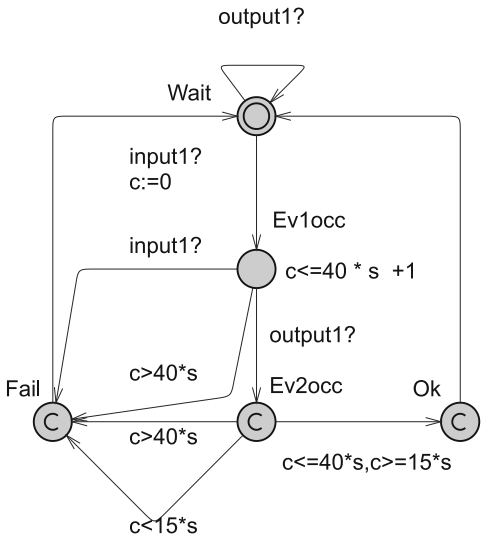


Abbildung 5.16.: Prüfautomat

Nach der Transformation des annotierten MLD-Modells liegt ein komplett generiertes Uppaal-Modell vor. Neben den erzeugten Templates und den zugehörigen Prüfanfragen wird noch eine allgemeine Anfrage `A[] !deadlock` erzeugt, siehe Abschnitt 5.3.5. Die Prüfung erfolgt anschließend über das Transformationstool durch den Verifier des Tools *Uppaal*, alternativ ist auch eine Verwendung des Tools *Uppaal* möglich.

5.6.3. Ergebnisse

Die Korrektheit des Gesamtmodells bzw. vorhandene Fehler können durch Verifikation der automatisch erstellten Prüfausdrücke nachgewiesen werden. In einem gültigen Modell werden sämtliche Prüfanfragen erfüllt. Es wurden zudem Untersuchungen zu ungültigen und fehlerhaften Modellen durchgeführt. Ein ungültiges Modell kann durch verschiedene Konstellationen hervorgerufen werden:

- Falsche Parameter: sind die annotierten Parameter nicht korrekt, so werden die Eigenschaften nicht erfüllt, oder die Beschränkungen, etwa des „Processing“ Templates verletzt.

- Syntaxfehler: Syntaxfehler werden durch das Transformationswerkzeug erkannt und dem Benutzer gemeldet, so dass diese korrigiert werden können.
- Leeres / Unvollständiges Modell: Ein leeres Modell äußert sich im Vorhandensein eines Deadlocks, da kein Folgezustand existiert. Selbiges tritt auf, falls ein strukturell ungültiges Modell vorliegt. Bei einem unvollständigen Modell kann zudem erkannt werden, dass bestimmte Eigenschaften gar nicht getriggert werden.

Durch Einschränkung der Modelltiefe ist so eine hierarchisch gestaffelte Verifikation des Eventmodells unter Verwendung von Realzeitautomaten möglich.

5.6.4. Bewertung

Gleichzeitig zeigen sich bei diesem Modell auch einige Nachteile der Methodik. So enthält das vollständige Realzeitautomatenmodell viele Einzelteile, wobei, auch wenn die Teile für sich sehr einfach sind, die Übersichtlichkeit stark eingeschränkt ist. Das Realzeitautomatenmodell ist also nur noch als Backend verwendbar. Die Umsetzung einer Modellebene als einen Realzeitautomaten würde jedoch eigene Probleme hervorrufen, da jeweils parallele Bestandteile in eigene Teilautomaten zerlegt werden müssten. Das geeignete Maß an Zerlegung in Teilmodelle, bzw. die mögliche Zusammenfassung rein sequentiell arbeitender Modellteile lässt Raum für Optimierungen zu. Sinnvoll erscheint auch hier die Verwendung automatischer Reduktionsmethoden auf der Ebene der Realzeitautomaten. Die gewählte Form der Zeitbeschreibungen durch eine kontextfreie Grammatik führt zu einem erhöhten Aufwand beim Erstellen der Modelle.

Zur Realisierung der hierarchischen Verifikation als automatischer Prüfprozess ist die Beschränkung auf ein allgemeines Umgebungsmodell, sowie die Implementierung eines Verfahrens zur Abhängigkeitsanalyse der Events erforderlich. Zudem werden im verwendeten Prototyp nicht alle Modellierungsmöglichkeiten des Simulationswerkzeuges unterstützt. Die Unterstützung der Typisierung der Events sowie die Verwendung von Memories für die Analyse sollten unterstützt werden. Dafür sollte zudem untersucht werden, inwiefern eine Unterstützung für die Extraktion von Eigenschaften aus Primitiven möglich und sinnvoll ist.

5.7. Zusammenfassung Transformation-basierte Verifikation

Durch die Transformation-basierte Verifikation wird eine Methode bereit gestellt, um temporale (zeitlich qualitative) und zeitlich quantitative Eigenschaften in Discrete-Event Modellen, die als ausführbare Spezifikationen dienen, nachzuweisen zu können. Um das zeitlich-funktionale Verhalten des Modells explizit darzustellen, so dass eine formale Analyse durchgeführt werden kann, ist die Erweiterung des Spezifikationsmodells um Annotationen in Form der Constraint Language erforderlich. Das zeitlich- funktionale Verhalte, sowie die zu prüfenden Eigenschaften werden während der Strukturanalyse des Modells in Form der annotierten Eigenschaften

extrahiert. Die Analyse selbst erfolgt dann dadurch, dass sowohl Eigenschaften als auch Anforderungen als jeweils eigene Modelle transformiert und gegeneinander geprüft werden.

Im Rahmen der Arbeit wurde ein Prototyp erstellt, mit dem die Anwendbarkeit der Methode demonstriert wurde. Für eine praktische Verwendbarkeit ist jedoch die Unterstützung weiterer Modellmerkmale (typisierte Events, Memories) zum Zwecke der Analyse erforderlich.

Im Gegensatz zu existierenden Ansätzen, die bestimmte Einschränkungen voraussetzen, oder bewusst eine domänenspezifische Modellierung verwenden, wird hier eine Methode untersucht, die allein die Verwendung der Discrete-Event Semantik voraussetzt. Dadurch ergeben sich einerseits weniger Beschränkungen bezüglich der Nutzung einer konkreten Modellierung, andererseits stehen weniger spezialisierte Analysemethoden zur Verfügung. Dabei zeigen sich mögliche Erweiterungen der Methodik:

- **Vereinfachung der Modellstruktur:**
  - Durch Zusammenfassung von prozessbasierten Zusicherungen gemäß Anhang D könnte eine automatische Vereinfachung realisiert werden.
  - Durch Zusammenfassung von Zusicherungen, z. B. durch Verwendung von Axiomen des Theorem Prooving.
  - Durch Eigenschaft erhaltende funktionale Abstraktion von zustandsbehafteten Teilmodellen, siehe [Löt99].
- **Anpassung der Analysemethodik:**
  - Verbesserung der Fehlersuche durch Erweiterung der Prüfanfragen entsprechend Tabelle 5.4.
  - Das im Prototyp verwendete Zielmodell nutzt Timed Automata. Es hat sich gezeigt, dass sich dadurch Einschränkungen bezüglich der Zeitanalyse und der Modellierung nebenläufiger Ereignisketten ergeben. Hier ist daher die Nutzung einer erweiterten Technik sinnvoll, die quantitative Zeiteigenschaften in Regionengraphen mit Berücksichtigung der Nebenläufigkeiten entsprechend der Modellstruktur bzw. Kausalität analysieren kann. Eine entsprechende Technik wird im Anhang E vorgestellt.
- **Modellextraktion:**
  - Die Annotation von Modellelementen sollte durch weitere Maßnahmen unterstützt werden. Denkbar ist die Erforschung einer Methode, die, unter Nutzung der Discrete-Event Semantik, eine direkte Transformation von Modellprimitiven ermöglicht.
  - Unterstützung der hierarchischen Verifikation auf Subsystemebene durch vollständige/hinreichende Bedingungen an das Umgebungsmodell.

6. Assertion-basierte Verifikation von Systemmodellen

In diesem Kapitel wird die Methode zur Assertion-basierten Verifikation heterogener temporaler Eigenschaften in Systemmodellen darstellt. Dazu wird zunächst das Konzept in Abschnitt 6.1 vorgestellt. Die Besonderheiten der Prüfung aufgrund der Heterogenität und Schritte der Vorverarbeitung werden in Abschnitt 6.1.2 erläutert. Dem schließen sich die Darstellungen der Algorithmen der Ansätze zur algorithmischen und symbolischen Prüfung der Eigenschaften in den Abschnitten 6.2 und 6.3 an. Die prototypische Implementierung wird in Abschnitt 6.4 dargestellt und in Abschnitt 6.5 wird die Validierung der Methode betrachtet. Abschließend werden in Abschnitt 6.6 die Ergebnisse zur Assertion-basierten Verifikation zusammengefasst und bewertet.

6.1. Konzept zur Assertion-basierten Verifikation von Systemmodellen

Im Abschnitt 3.5 wurde der Ansatz zur semiformalen Prüfung von Eigenschaften für heterogene Systemmodelle vorgestellt. Die Analyse der in heterogenen Modellen verwendeten Berechnungsdomänen erfolgte in Abschnitt 4.3. Die gewonnenen Erkenntnisse dienten der Entwicklung einer temporalen Beschreibungssprache für Eigenschaften, die auf Events basiert und die Beschreibung von Signalen verschiedener Berechnungsdomänen ermöglicht, siehe Abschnitt 4.5 und Anhang A.2.

Prinzipiell handelt es sich bei den Eigenschaften jeweils um eine endliche oder unendliche formale Sprache, deren Alphabet die elementaren Aussagen über Ereignisse der betrachteten Signale darstellen. Die Entscheidung, ob eine betreffende Eigenschaft gilt oder nicht, kann demzufolge als Akzeptanztest durch einen endlichen Automaten beschrieben werden [Sch01].

Es muss also eine Methode gefunden werden, um eine konkrete XFLTL-Eigenschaft in einen Automaten zu überführen. Dazu muss zunächst eine geeignete Automatenklasse definiert werden. Anschließend ist ein Algorithmus für die Generierung eines entsprechenden Automaten zu definieren. Abschließend muss die Verwendung der Automaten während der Cosimulation, insbesondere Fragestellungen zur Instantiierung der Automaten, geklärt werden.

In diesem Kapitel werden zwei Ansätze untersucht, bei denen die Konstruktion der Automaten implizit bzw. explizit realisiert ist. Der implizite Ansatz realisiert einen Automaten (Produktautomaten) dadurch, dass jedem Operator der Formel direkt ein Automat zugeordnet wird, dessen Verknüpfung sich aus der Baumstruktur der rekursiv definierten Formel der Eigenschaft ableitet, im Folgenden auch als algorithmische Prüfung bezeichnet, siehe Abschnitte 6.2 und 6.4. Der explizite Ansatz generiert einen Automaten, indem die Formel zwar analysiert, aber nicht strukturell als Ausgangsbasis dient, im folgenden auch als symbolische Prüfung bezeichnet, siehe Abschnitt 6.3.

In den folgenden Abschnitten werden verschiedene Ansätze zur symbolischen Prüfung temporaler Eigenschaften betrachtet. Dabei werden Ansätze zur Prüfung diskreter Eigenschaften herangezogen und mögliche Weiterentwicklungen für kontinuierliche Systeme aufgezeigt.

6.1.1. Techniken für diskrete Eigenschaften

Für die Assertion-basierte Prüfung zeitbeschränkter diskreter temporaler Eigenschaften wurden verschiedene Ansätze entwickelt. In [RHKR00] wird ein Ansatz vorgeschlagen, bei dem die Eigenschaftsprüfung implizit erfolgt. Grundidee ist die Realisierung der Prüfung einer Eigenschaft entsprechend der rekursiven Definition seiner Operatoren. Dazu wird jedem temporalen Operator ein nichtdeterministischer Automat zugeordnet, der dynamisch um neue Strukturen erweitert werden kann. Der Zustand eines Operator ist dabei in dreiwertiger Logik beschrieben, wobei jeder Operator den Status *true* (T), *false* (F) oder *pending* (P) annehmen kann. Der Status *pending* beschreibt, dass die Eigenschaft noch nicht entscheidbar ist, da bei zukunftsorientierter temporaler Logik der Zustand temporaler Aussagen zu einem Zeitpunkt der Auswertung unbekannt sein kann, siehe Tabelle 6.1. Vorteil dieses Ansatzes ist neben der vergleichsweise einfachen Realisierung die Aktualität der Prüfung.

Um diesen Ansatz für die Prüfung kontinuierlicher Zeiten verwenden zu können, ist eine Erweiterung der temporalen Operatoren um kontinuierliche Zeitbedingungen erforderlich. Dabei stellt sich die Frage *wann* und *wie oft* geprüft werden muss, um eine korrekte Auswertung zu erreichen, bzw. welche Einschränkungen für eine Prüfung kontinuierlicher Zeiten bestehen.

x	¬x
T	F
P	P
F	T

∨	T	P	F
T	T	T	T
P	T	P	P
F	T	P	F

∧	T	P	F
T	T	P	F
P	P	P	F
F	F	F	F

Tabelle 6.1.: Logische Operatoren für dreiwertige Logik nach [RHKR00]

Ein weiterer Ansatz wurde in [WRKR05] vorgestellt, bei dem die Eigenschaften explizit als Automat repräsentiert werden. Dabei wird, ausgehend von der jeweiligen Definition, für jeden einzelnen Operator zunächst ein nichtdeterministischer Automat erzeugt. Dieser wird

anschließend durch Verschmelzung potentieller zeitgleicher Zustände in einen deterministischen Automaten überführt. Die resultierenden Automaten besitzen zwei Terminalzustände, die mit *A*ccept bzw. *R*eject gekennzeichnet werden. Diese Methode stellt eine Weiterentwicklung des impliziten Ansatzes dar. Vorteilhaft ist die statisch bekannte Struktur, die für beliebig viele Instanzen einer Eigenschaft wiederverwendet werden kann [64]. Möchte man diesen Ansatz für die Prüfung kontinuierlicher Zeiten verwenden, so stellt man fest, dass die Verschmelzung nicht ohne weiteres möglich ist. Ursache dafür ist, dass die abzudeckenden Zeitintervalle nicht mehr diskret abzählbar sind und dadurch keine einfache Zuordnung der zu verschmelzenden Zustände möglich ist. Es bestehen demnach zwei Möglichkeiten. Man könnte aus den Parametern der Eigenschaften eine gültige Diskretisierung ermitteln und das bestehende diskrete Verfahren nutzen. Alternativ könnte die Verschmelzung auf der Basis kompletter Pfade realisiert werden. Hier stellt sich also die Frage, wie ein Automat für die Prüfung kontinuierlich zeitbeschränkter temporaler Eigenschaften konstruiert werden kann.

6.1.2. Partitionierung von XFLTL-Eigenschaften in Intervalle

Grundlegend bei der Verwendung kontinuierlicher Zeiten ist, dass die zu prüfenden Zeiten nicht abzählbar sind. Daraus ergibt sich die wesentliche Herausforderung eine möglichst optimale[65] Zerlegung der Eigenschaft in Zeitintervalle während der Prüfung zu erreichen. Durch das Event-basierte Signalmodell lässt sich prinzipiell eine Partitionierung des zu prüfenden Signalvektors finden, da der Signalvektor abgesehen von den Events jeweils stückweise konstant inaktiv ist, siehe Abschnitt 4.5.1.

Die benötigten Partitionierungen haben unterschiedliche Ursachen. Durch die Intervallgrenzen der zu prüfenden Eigenschaft ergeben sich offensichtlich notwendige Prüfintervalle. Durch die Änderung von Signalen innerhalb der bestimmten Intervalle ergeben sich jedoch dynamisch zusätzliche Zeitpunkte, zu denen eine Prüfung erfolgen muss. Abbildung 6.1 verdeutlicht diesen Zusammenhang. Dargestellt ist die Partitionierung der Eigenschaft $\varphi = G_{[4,5]}(F_{[2,6]}e_x)$, die besagt, dass während eines Zeitintervalls $[4,5]$ jeweils innerhalb des Intervalls $[2,6]$ ein Event des Typs x auftreten muss. Links ist die Zerlegung der Eigenschaft in seine Operatoren als Baum dargestellt, rechts die Intervalle für die betreffenden Operatoren. Daraus ergeben sich, bezogen auf den Startzeitpunkt von φ, zwei Intervalle für den $F-$Operator: $[6,10]$ und $[7,11]$. Im unteren Teil ist der Einfluss eines tatsächlich auftretenden Events zum Zeitpunkt $t = 6.5$ dargestellt. Im betreffenden Fall wird dadurch zwar der F-Operator für das erste Intervall erfüllt, allerdings ergibt sich die Notwendigkeit eines zusätzlichen Intervalls im Intervall $[6.5, 10.5]$. Allerdings erweist sich das Einfügen des zusätzlichen Intervalls als schwierig, da der betreffende Zeitpunkt $t = 4.5$ bereits verstrichen ist. Betrachtet man die Struktur der Formel, so zeigt sich,

[64] Lediglich der aktuelle Zustand einer Eigenschaftsinstanz muss separat abgelegt werden.
[65] Die Zerlegung in möglichst wenige Intervalle, die eine korrekte Prüfung ermöglichen, soll als optimal gelten.

dass nur die Zeitintervalle bezüglich Wurzel und Blätter erhalten bleiben müssen, der Zeitbezug innerer Knoten darf dagegen geändert werden. Abbildung 6.2 zeigt eine mögliche Änderung, bei der das Intervall des G-Operators soweit nach rechts verschoben wird, dass die linke Intervallgrenze des F-Operators 0 beträgt. Dadurch erreicht man eine Formelstruktur, in der das Einfügen zusätzlicher Prüfintervalle automatisch in Abhängigkeit von auftretenden Signalen realisiert werden kann. Allerdings kann nicht jede Formel auf diese Weise vereinfacht werden. Enthält die Struktur der Formel mehrere Blätter (Signale), so werden besondere Bedingungen an die Struktur der Formel gestellt, um eine gültige Umformung vornehmen zu können. Die beiden gewählten Ansätze verfolgen unterschiedliche Strategien, um, sofern möglich, eine korrekte Umformung zu realisieren.

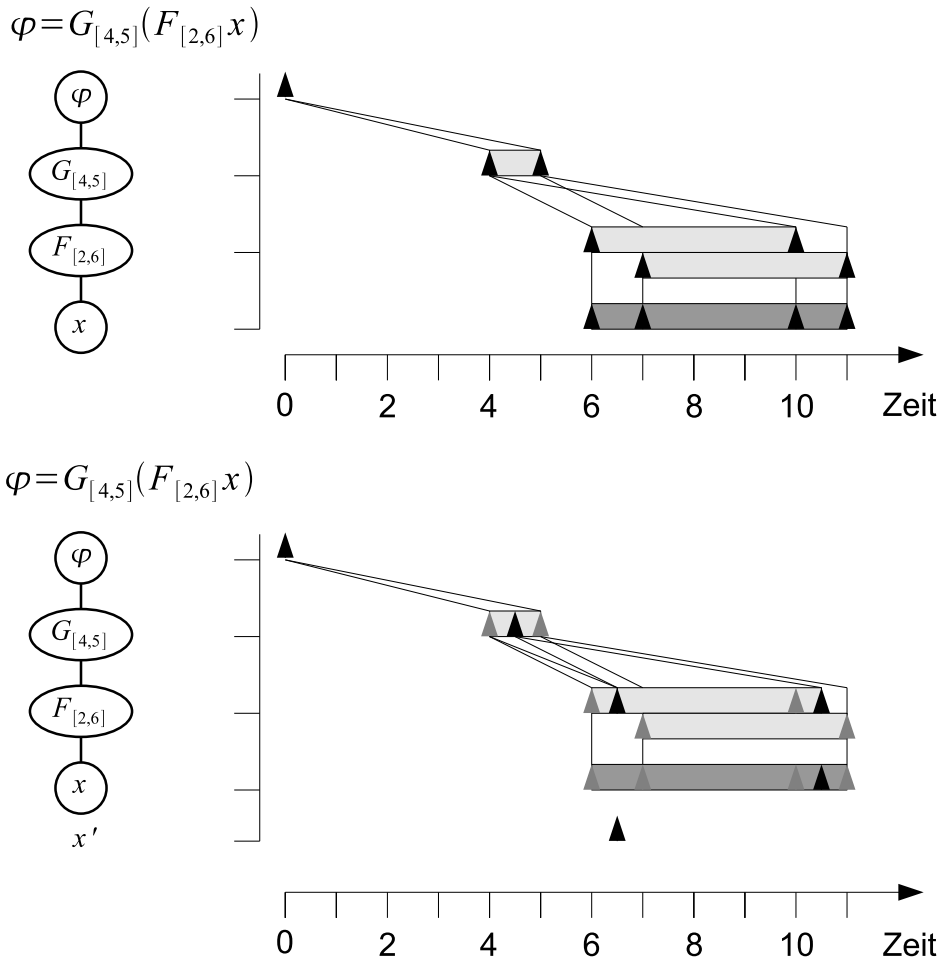


Abbildung 6.1.: Signalabhängige Partitionierung von Intervallen

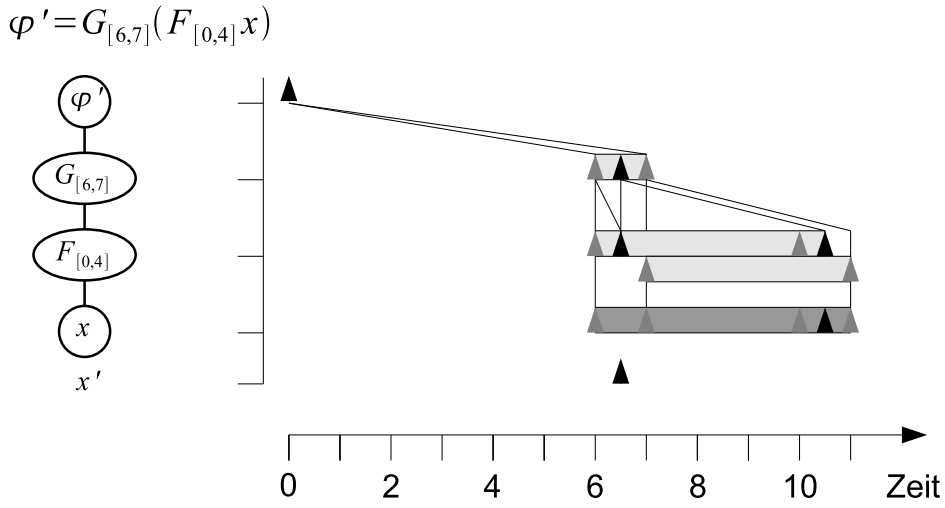


Abbildung 6.2.: Verschiebung der Intervalle von temporalen Operatoren

6.2. Algorithmische Prüfung von Eigenschaftstemplates

Das Grundkonzept der algorithmischen Prüfung besteht in der dynamischen Instantiierung der zu prüfenden Eigenschaft, basierend auf den in der Formel enthaltenen booleschen und temporalen Operatoren, siehe Abschnitt 6.1.1. Während die booleschen Operatoren logische Funktionen gemäß Tabelle 6.1 realisieren, können die temporalen Operationen als Automat, speziell als Timed Automat, realisiert werden.

6.2.1. Einschränkungen

Für diese Art der Realisierung sind einige Einschränkungen der instantiierbaren Formen zu beachten. Wie in Abschnitt 6.1.2 erläutert, besteht bei der Prüfung temporaler Operatoren in kontinuierlicher Zeit allgemein das Problem der zeitlichen Partitionierung der Prüfung, also der Diskretisierung der Zeit relativ zu auftretenden Signalen und Intervallen an Operatoren. Speziell ergibt sich beim gewählten Ansatz das Problem der Prüfung eines endlichen Zeitintervalls durch eine endliche Menge an Operatoren.

Dies bedeutet, dass nur diejenigen Formeln geprüft werden können, die sich aufgrund ihrer Struktur entsprechend realisieren lassen. Eine variable Partitionierung wird prinzipiell durch G- und U-Operatoren bedingt, und erfordert, dass die direkt umschlossenen temporalen Operatoren jeweils konjunktiv mit einem positiven[66] Signal verknüpft sind. Die Response-Eigenschaft gemäß Formel (6.1) besteht aus einem G-Operator der einen F-Operator umschließt, die Intervalle seien beliebige gültige Intervalle. Durch äquivalente Umformungen kann eine Darstellung gemäß Formel (6.3) erfolgen, in der der F-Operator konjunktiv mit dem Eventsignal e_x ver-

[66] nicht negierten

knüpft ist. Dies bewirkt, dass eine Instantiierung des F-Operators nur jeweils dann erfolgen muss, falls ein Event auftritt, also je endlichem Zeitintervall nur endlich oft[67].

$$varphi_1 = G_I(e_x \implies F_I(e_y)) \tag{6.1}$$
$$= G_I((\neg e_x) \vee F_I(e_y)) \tag{6.2}$$
$$= G_I((\neg e_x) \vee (e_x \wedge F_I(e_y))) \tag{6.3}$$

In Abschnitt 6.1.2 wurde die Verschiebung von Intervallen temporaler Operatoren dargestellt. Dies ist erforderlich, um eine Signal-abhängige Prüfung der Eigenschaften realisieren zu können. Für die algorithmische Prüfung müssen die Intervalle der Operatoren so verschoben werden, dass, von der Wurzel aus betrachtet, ausgehend von jedem temporalen Operator mit variabler Anzahl an Intervallen nur temporale Operatoren mit 0 als unterer Grenze vorkommen, bis eine konjunktive Verknüpfung mit einem positiven Eventsignal auftritt. Tritt ein solches konjunktiv verknüpftes Event ein, so muss prophylaktisch eine zusätzliche Partitionierung erfolgen, um bei (zeitlich verzögerter) Erfüllung des zweiten Terms eine korrekte Partitionierung zu realisieren. Formel 6.4 beschreibt eine Safety-Eigenschaft, die sich in zwei Teilbedingungen mit *F-Operator* aufteilt. Der Index an dem temporalen Operator weist hier auf ein beliebiges positives Intervall (I) bzw. auf ein mit 0 beginnendes Intervall (0) hin. Die F-Operatoren enthalten jeweils konjunktiv verknüpfte Eventsignale e_x bzw. e_y. Tritt ein Event e_y ein, so ist ggf. der zweite Term $F_I(e_z)$ erfüllbar und es muss von G aus gesehen für den gesamten Term $(F_0 e_x) \vee F_0(e_y \wedge F_I(e_z))$ eine Partitionierung vorgesehen werden, falls die Intervallgrenze des Operators G_I noch nicht erreicht ist. Der Operator G_I muss also immer genau dann eine Partitionierung vornehmen, wenn im Signalvektor ein Event e_x oder e_y aktiv ist.

$$\varphi_2 = G_I((F_0 e_x) \vee F_0(e_y \wedge F_I(e_z))) \tag{6.4}$$

Um die konkreten Events für eine minimale Partitionierung bestimmen zu können, sind demnach globale Informationen über die zu prüfende Eigenschaft erforderlich. Zusätzliche Partitionierungen verändern zwar grundsätzlich nicht das Ergebnis der Prüfung, führen jedoch zu einem erhöhten Aufwand. Eine einfachere Realisierung der Partitionierung unter den genannten Einschränkungen der Formeln besteht dann darin, bei jedem eintretenden Event eine Partitionierung der Terme vorzunehmen, ohne zu prüfen, ob im konkreten Fall eine solche Partitionierung erforderlich ist.

[67] Vergleiche die Eigenschaften Event-basierter Signale in Abschnitt 4.5.2

6.2.2. Operatorinstanzen

Operatoren liegen zur Laufzeit als Objekte vor und bilden n-stellige Verknüpfungen aus referenzierten Formelobjekten, im Folgenden als *Terme* bezeichnet. Die Auswertung von Operatoren wird von außen zu bestimmten Zeitpunkten gestartet. Dazu können neue Instanzen der *Terme* erstellt werden, *Terme* können ausgewertet werden, nicht mehr benötigte (bereits vollständig ausgewertete) *Terme* können entfernt werden. Eine Instanz eines Terms gilt als vollständig ausgewertet, wenn sich sein Zustand nicht mehr ändern kann, also True oder False beträgt. Bei der Instantiierung eines Terms wird der Zeitpunkt der Instantiierung als Referenzzeit für die Intervalle der enthaltenen temporalen Operatoren verwendet.

Die booleschen Operatoren erfüllen die dreiwertige Logik entsprechend Tabelle 6.1. Dazu werden zuerst die Terme ausgewertet und anschließend deren Ergebnisse verknüpft.

Die temporalen Operatoren (Instanzen) können als Zustandsautomat, speziell als markierter Timed Automata beschrieben werden, siehe Abbildungen 6.3 und 6.4. Der Initialzustand S_0 ist mit einem eingehenden Pfeil markiert. Neben der Zustandsbezeichnung ist der Status des Operators notiert. Dieser beträgt in den nichtterminalen Zuständen P und in den terminalen Zuständen entweder T oder F. Im Zustand wird zusätzlich eine optionale Entry-Action notiert, wie z.B. das Setzen einer Zeitvariablen (clock c:=0) oder die Auswertung aller instantiierten Terme ($check(\varphi_i)$). Der Status der temporalen Operatoren kann sich während der Auswertung ändern. Zustandsübergänge sind in der Form $event[Bedingung]/Aktion$ notiert. Die Auswertung findet bei eingehenden Events (markiert mit *event*) oder bei zusätzlichen zeitgesteuerten Pseudoevents (markiert mit τ) statt. Weiterhin existieren Übergangsbedingungen bezüglich Zeiten und/oder Teilformeln. Als Aktion wird an den Zustandsübergängen lediglich die Instantiierung einer Teilformel betrachtet.

Der F-Operator arbeitet wie folgt. Ist die untere Zeitschranke erreicht, so wird ein Term instantiiert und ausgewertet. Mit jedem weiteren Event wird eine neue Instanz des Terms angelegt und ausgewertet, bis die obere Zeitschranke erreicht wurde, oder ein Term mit true bewertet wurde. Ist die obere Zeitschranke erreicht, so werden nur noch bestehende Instanzen ausgewertet, bis entweder alle Terme den Zustand false besitzen oder ein Term den Wert true besitzt.

Der G-Operator arbeitet wie folgt. Ist die untere Zeitschranke erreicht, so wird ein Term instantiiert und ausgewertet. Mit jedem weiteren Event wird eine neue Instanz des Terms angelegt und ausgewertet, bis die obere Zeitschranke erreicht wurde, oder ein Term mit false bewertet wurde. Ist die obere Zeitschranke erreicht, so werden nur noch bestehende Instanzen ausgewertet, bis entweder alle Terme den Zustand true besitzen oder ein Term den Wert false besitzt.

Durch die temporalen Operatoren wird also beschrieben, ob und wann ein enthaltener Term relativ zu seiner Instantiierungszeit ausgewertet werden muss. Gleichzeitig wird sichergestellt,

F-Operator: $F_{[a,b]}\varphi$

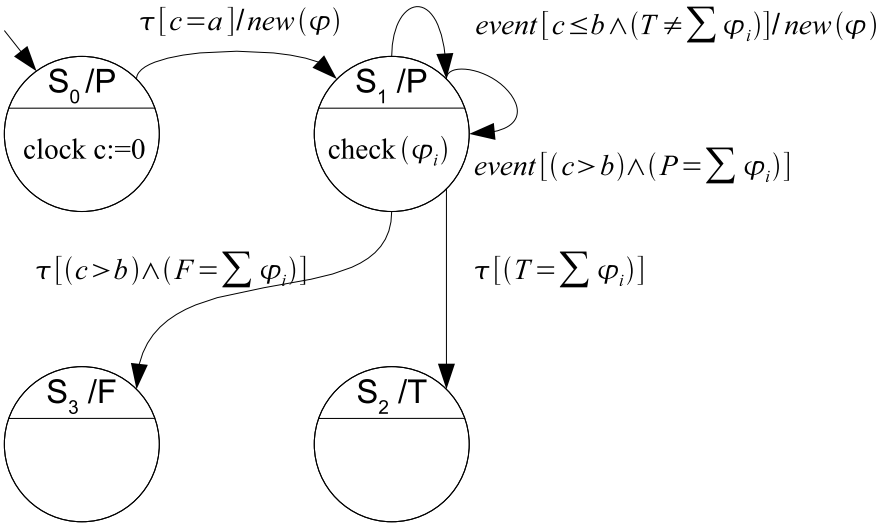


Abbildung 6.3.: Timed Automat für den F-Operator

G-Operator: $G_{[a,b]}\varphi$

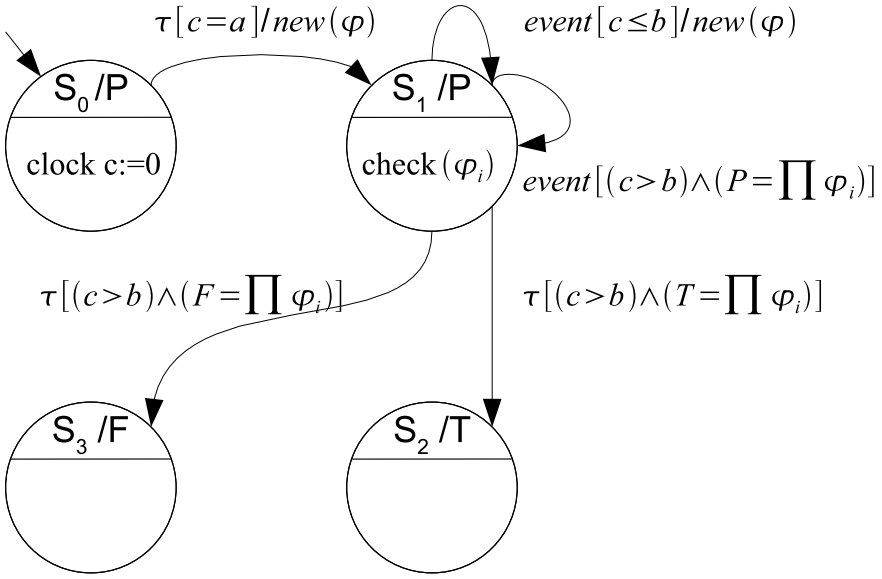


Abbildung 6.4.: Timed Automat für den G-Operator

dass Formeln entsprechend der Struktur rekursiv, beginnend mit den Signalinstanzen ausgewertet werden.

6.2.3. Signalinstanzen

Während die Funktionsweise der Operatoren bereits erläutert wurde, ist die Realisierung der elementaren Aussagen über Signale noch offen geblieben. Prinzipiell stellen die elementaren Aussagen spezielle Operatoren dar, müssen aber gesondert betrachtet werden. Auch hier muss eine Verknüpfung in Form von Objekten erfolgen, die im Folgenden als Signalinstanzen bezeichnet werden.

Signalinstanzen sind also Referenzen zu den sich zeitlich ändernden Signalen des zu bewertenden Simulationsmodells. Da ein Term gleichzeitig Signale mit unterschiedlichen Zeitreferenzen bewerten können muss, ist eine Verknüpfung der Signalinstanzen mit konkreten Zeitpunkten zu realisieren. Dies ermöglicht weiterhin die wiederholte Bewertung eines Terms.

Eine Lösung ist wie folgt realisiert. Nach der Instantiierung ist eine Signalinstanz ungebunden, besitzt also den Signalwert zum Auswertezeitpunkt. Während der Bewertung erfolgt eine Verknüpfung mit dem Auswertezeitpunkt, so dass die Signalbewertung bei den nachfolgenden Bewertungen konstant bleibt. Um neue Zeitpunkte referenzieren zu können, müssen also lediglich neue Signalinstanzen erzeugt werden.

6.2.4. Kopplung mit dem Simulationsmodell

Bisher wurde die Funktionsweise der Eigenschaftsbewertung aus der Sicht der Eigenschaft, repräsentiert durch die beschreibende Formel, dargelegt. Noch zu betrachten sind die Anbindung der Signale an das Simulationsmodell, die Ermittlung von Zeitpunkten zur Bewertung sowie die Instantiierung von Formeln an sich.

Die Signale besitzen ein Event-basiertes Zeitmodell, was bedeutet, dass sich die Eigenschaften nur zu den Zeitpunkten ändern können, zu denen ein Event auftritt, bzw. immer dann, wenn bezüglich der Eigenschaft eine Zeitschranke erreicht wird.

Die Spezifikation einer Discrete-Event Simulation besagt, dass gleichzeitig auftretende Events sequentiell abgearbeitet werden, siehe Abschnitt 2.3.6. Da die Eigenschaft auftretende Ereignisse synchronisiert beschreibt, muss also eine Synchronisation aller gleichzeitig auftretenden Ereignisse erfolgen. Für die Prüfung wird daher ein synchronisierter Signalvektor verwendet.

Da die Simulation mit Ereignissen voranschreitet, muss vor Nutzung des neuen Signalvektors geprüft werden, ob inzwischen Timeouts aufgetreten sind, vergleiche dazu Abschnitt 4.5.1. Mögliche Timeout ergeben sich aus den Zeitreferenzen der instantiierten temporalen Operatoren. Daher werden zuerst alle Timeouts abgefragt und die Zeitpunkte entsprechend der Monotonie der Zeit mit leerem Signalvektor[68] geprüft, so dass eventuell zusätzlich erforderliche Partitionie-

[68] Alle Signale sind mit false bewertet.

rungen der Zeit durch Instantiierungen von Termen realisiert werden. Anschließend erfolgt die Prüfung der Bewertung mit dem synchronisierten Signalvektor. Dem schließt sich eine Prüfung der nachfolgenden Intervallgrenze mit inaktivem Signalvekor an.

Für die Instantiierung von Formeln gelten dieselben Einschränkungen, wie in Abschnitt 6.2.1 dargestellt, also die Kopplung von Instantiierungen an Zeitpunkte relativ zu auftretenden Events. Als Formel beschrieben, kann der G-Operator mit einem zeitlich unbeschränkten Intervall entsprechend Gleichung 6.5 genutzt werden. Die Verwendung einer Implikation sorgt für eine Trennung von Trigger und Eigenschaft.

$$\psi = G_{[0,\infty]}(\varphi_{Trigger} -> \varphi_{Eigenschaft}) \tag{6.5}$$

6.2.5. Eigenschaftsbewertung

Die Bewertung einer einzelnen Eigenschaft liefert lediglich *true*, *false* oder *pending* als Resultat. Soll eine Eigenschaft φ invariant gelten, so kann sie als $G_{[0,\infty]}\varphi$ beschrieben werden. Liegt φ als Implikation vor, so sind detaillierte (paarweise) Informationen über Prämisse und Konklusion wünschenswert. Für eine sinnvolle Verwendung dieser Aussage sind daher weitere Informationen erforderlich.

Verwendet man eine Instantiierung entsprechend der Gleichung 6.5, so würde eine nicht erfüllte Instanz einer *Eigenschaft* zur Terminierung aller weiteren Eigenschaften führen. Weiterhin ist auch die Kenntnis über Instanzen von *Trigger* und *Eigenschaft* sowie deren Zusammengehörigkeit für eine Bewertung von Bedeutung. Eine Lösung dieses Problems besteht in der Verwendung eines *Assert*-Operators, der die Instantiierung von *Trigger* und *Eigenschaft* verwaltet, die Zusammengehörigkeit explizit macht, und nicht terminiert.

Zur Verwendung der Eigenschaften gehören neben den Zeitpunkten zusammengehöriger Instanzen von *Trigger* bzw. *Eigenschaft* auch die jeweils zeitliche Ausdehnung und die von den betrachteten Instanzen referenzierten Signale mit ihrem Verlauf. Aus der Struktur der konkreten Instanzen kann dann ein Fehlerbaum abgeleitet und in Relation zu den betroffenen Signalen notiert werden.

6.2.6. Schlussfolgerungen

Zunächst wurden Einschränkungen betrachtet, unter denen sich XFLTL-Eigenschaften durch Cosimulation algorithmisch prüfen lassen, indem Eventsignale zur Partitionierung der Zeit verwendet werden. Die Prüfung selbst ist durch die Instantiierung der zu prüfenden XFLTL-Formel durch ihre Operatoren realisiert. Die grundlegenden Operatoren wurden entsprechend vorgestellt. Die Bewertung der Eigenschaft hängt von den Eingangssignalen ab, die in den Blättern der Baumstruktur der XFLTL-Formel zu finden sind. Die Eingangssignale nehmen als elementare Aussagen die Rolle eines speziellen Operators ein, der während der Prüfung

dynamisch an konkrete Zeitpunkte des Eingangssignals gebunden wird. Für die Kopplung an das Simulationsmodell müssen Vorkehrungen getroffen werden, sodass das Simulationsmodell eine asynchrone Semantik und das Eigenschaftsmodell eine synchrone Semantik besitzen. Gelöst wird dies durch die Synchronisation gleichzeitig auftretender Eingangssignale zu einem synchronen Eingangsvektor.

Die getroffenen Einschränkungen, also die Bindung der zeitlichen Partitionierung an auftretende Events, sind erforderlich, da kein Wissen über zukünftige Signalverläufe vorhanden ist. Bei der Verwendung eines rein Event-basierten Modells sollten sich diese Einschränkungen jedoch kaum auswirken.

Es bleibt Spielraum für Optimierungen und Erweiterungen. Die Partitionierung (Instantiierung von Operatoren) wird derzeit durch beliebige Events getriggert, eine weitergehende Selektion würde den Prüfaufwand reduzieren, aber eine globale Betrachtung der zu analysierenden Eigenschaften erfordern. Die Prüfung der Eigenschaften erfolgt nichtdeterministisch. Daher ist insbesondere der Speicherbedarf nicht ohne weiteres abschätzbar und kann dynamisch sehr stark variieren.

Lohnenswert erscheint daher ein Ansatz, der eine symbolische Repräsentation der Eigenschaften erlaubt und eine deterministische Prüfung realisiert.

6.3. Symbolische Prüfung mittels Timed-AR-Automata

Das Grundkonzept der symbolischen Prüfung besteht darin, dass eine Eigenschaft explizit als Automat repräsentiert wird. Dadurch sind Optimierungen und Prüfungen aufgrund der Automatenstruktur möglich. Die Prüfung der Eigenschaften selbst kann dann durch die Simulation der Automaten realisiert werden.

6.3.1. Auflösung des Nichtdeterminismus

Grundsätzlich besteht das Problem einen deterministischen Automaten zu erzeugen, wobei auch bei diesem Ansatz die Partitionierung der Zeiten realisiert werden muss. Der Nichtdeterminismus besteht also einerseits in der Zeit bzw. Partitionierung und andererseits in der Struktur der XFLTL-Formel, die es ermöglicht, durch verschiedene Pfade eine Erfüllung bzw. Nichterfüllung der Eigenschaft zu realisieren.

Der Nichtdeterminismus resultierend aus der Struktur der Formel wird im originalen Ansatz dadurch gelöst, dass die einzelnen Pfade schrittweise in Erreichbarkeitsklassen zusammengefasst werden [WRKR05]. Dies ist in einem diskreten Zeitmodell mit synchronen Automaten vergleichsweise einfach möglich, da die Formel durch einen Automaten mit der Struktur eines gerichteten azyklischen Graphen beschrieben werden kann, und Zustände unterschiedlicher Pfade zeitlich eineindeutig zuordenbar sind. Dadurch kann die Zusammenfassung von Zuständen,

und damit die Bestimmung der deterministischen Zustandsübergangsbedingungen, direkt auf der Basis der zusammengehörigen Zustände erfolgen.

Betrachtet man ein kontinuierliches Zeitmodell, so muss eine andere Automatenklasse, wie z.B. Timed Automata, verwendet werden. Timed Automata nutzen Zeitvariablen und Zeitintervalle, so dass die Auswahl der zur Verknüpfung verwendeten Zustände von Belegungen der Zeitvariablen abhängig ist. Die möglichen Belegungen der Zeitvariablen hängen von den Operatoren sowie den möglicherweise auftretenden Events an Signalen ab. Um die Signalabhängigkeiten automatisch durch den Automaten auflösen zu können, dürfen die zur Partitionierung führenden Signale nicht in der Zukunft liegen. Dies kann prinzipiell durch eine Verschiebung der Operatorintervalle realisiert werden, siehe Abschnitt 6.1.2 und Abbildung 6.2. Dies kann durch eine *Normalisierung* der XFLTL-Eigenschaft bezüglich der Zeit realisiert werden, wie in Abschnitt 6.3.2 erläutert wird.

Liegen keine Signalabhängigkeiten in der Zukunft, so können die Ereignisse genutzt werden, um automatisch zusätzliche signalabhängige Partitionierungen zu realisieren, indem Zeitvariablen zurückgesetzt werden.

6.3.2. Normalisierung des Zeitmodells

Es dürfen keine zur Partitionierung benötigten Signale in der Zukunft liegen. Für eine deterministische symbolische Beschreibung muss weiterhin die Anzahl gleichzeitig zu betrachtender Intervalle bekannt sein. Die Darstellung einer XFLTL-Eigenschaft in zeitlich deterministischer Form wird als *normalisiert* bezeichnet. Zeitlich deterministisch bedeutet dabei, dass eine notwendige Partitionierung, bezogen auf die Formelstruktur, nur von aktuellen und zurückliegenden Ereignissen abhängt.

Um während der symbolischen Prüfung von XFLTL-Eigenschaften die Zeit automatisch in Intervalle partitionieren zu können, müssen die beschränkenden Zeitintervalle an temporalen Operatoren an 0 ausgerichtet sein. Im Unterschied zum algorithmischen Ansatz muss dies für alle temporalen Operatoren gelten, da sämtliche Operatoren durch einen einzelnen Automaten beschrieben werden müssen.

Dies ist allerdings nur dann erreichbar, wenn alle Ereignisse, die zu einer Partitionierung führen, relativ zum Startzeitpunkt der Eigenschaft zur selben Zeit stattfinden, so dass sich die Bedingung zur Partitionierung als kombinatorischer Ausdruck angeben lässt. Dies ist in der Regel nicht der Fall, könnte allerdings durch Verschiebung der Signale realisiert werden.

Die Verschiebung der Signale ist demzufolge der weiter verfolgte Ansatz. Da Signale aus Gründen der Kausalität nicht in die Vergangenheit geschickt werden können, ist nur eine Verzögerung der Signale möglich. Möglich ist eine Ausrichtung der Formel, so dass alle Operatoren in der Vergangenheitsform, also durch Intervalle mit negativen Intervallgrenzen, beschrieben werden. Dazu wird innerhalb der betrachteten XFLTL-Formel der längste Pfad bezüglich der

Intervalle an den temporalen Operatoren gesucht, und die Länge als Referenzzeitpunkt verwendet. Anschließend werden die Intervalle der anderen Operatoren relativ zueinander so verschoben, dass die oberen Grenzen aller Intervalle denselben Referenzzeitpunkt berühren. Die in der XFLTL-Formel enthaltenen Signale werden um den Betrag verschoben, um den das Intervall des direkt übergeordneten temporalen Operators verschoben wurde. Die Intervalle werden nun von dem letzten Zeitpunkt aus rückwärts betrachtet, so dass sie relativ gesehen in der Vergangenheit beginnen und zum Zeitpunkt 0 enden. Damit die relativen Werte korrekt sind, wird ein F-Operator mit einem Punktintervall der Verschiebung in die Formel eingefügt, die dem Betrag des Referenzzeitpunktes entspricht. Eine Sonderrolle nimmt hierbei der $U-$Operator ein. Aus seiner Definition $\varphi_1 U_{[a,b]} \varphi_2$ folgt, dass beginnend mit einem Zeitpunkt t_0 die Eigenschaft φ_1 gelten muss, und zwar bis zu einem Zeitpunkt $t_1 \in [a, b]$, an dem auch φ_2 gilt. Betrachtet man vom Zeitpunkt $t = b$ aus die Eigenschaft rückwärts, so müssen die Zeitpunkte $t = 0$ und $t = a$ vermerkt werden, also ab wann φ_1 gelten muss bzw. ab wann φ_2 gilt. Damit ergeben sich für den normalisierten U-Operator zwei Intervalle, $[-b, 0]$ und $[-(b-a), 0]$, notiert als $U_{[-c_1,0],[-c_2,0]}$ mit $c_1 = b$ und $c_2 = b - a$.

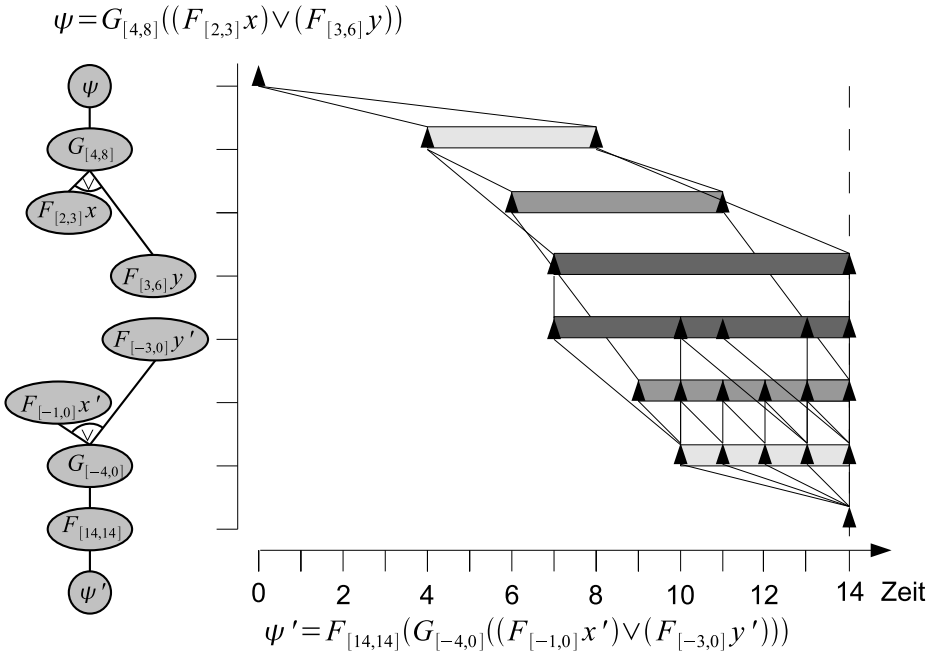


Abbildung 6.5.: Beispiel zur Normalisierung von Formeln

Abbildung 6.5 zeigt die Vorgehensweise anhand der Formel $\psi = G_{[4,8]}((F_{[2,3]}x) \vee (F_{[3,6]}y))$. Der obere Teil zeigt die ursprüngliche Formel mit der zeitlichen Zuordnung der Intervalle der Operatoren relativ zum Beginn der Eigenschaftsprüfung. Der längste Pfad ist $G_{[4,8]}(F_{[3,6]}y)$, so dass sich ein Referenzzeitpunkt $t = 14$ ergibt, an dem die Operatoren ausgerichtet werden. Im unteren Teil ist die Verschiebung angedeutet, sodass sich die normalisierte Formel $\psi' = F_{[14,14]}(G_{[-4,0]}((F_{[-1,0]}x') \vee (F_{[-3,0]}y')))$ mit $y'(t) = y(t)$ und $x'(t) = x(t-3)$ ergibt.

Diese Normalisierung erfüllt die Anforderungen an die Assertion, allerdings existieren zwei Nachteile. Die Auswertung erfolgt immer zum spätesten möglichen Zeitpunkt bezüglich der Eigenschaft. Für jeden Startzeitpunkt der Prüfung muss ein neuer Automat instantiiert werden.

Adaptive Normalisierung

Die vorgeschlagene Normalisierung beschreibt die Eigenschaft aus der Sichtweise der Prüfung durch Beobachtung der Eventsignale vom letztmöglichen Zeitpunkt aus. Abhängig von der Struktur der Formel kann prinzipiell auch ein anderer Referenzzeitpunkt gewählt werden, indem der von der Wurzel aus betrachtet erste temporale Operator, sofern er sich nicht in einer Verzweigung befindet, in die Zukunft referenziert wird. Das Intervall dieses ersten Operators wird dabei so verschoben, dass es mit 0 beginnt. Dann wird der erste Operator iterativ, also zukunftsgerichtet geprüft, während die restlichen Operatoren kombinatorisch, also vergangenheitsgerichtet geprüft werden. Vorteil ist, dass ggf. früher entschieden werden kann, ob die Formel erfüllbar oder nicht erfüllbar ist. Dieser Ansatz wird im Folgenden als *adaptive Normalisierung* bezeichnet.

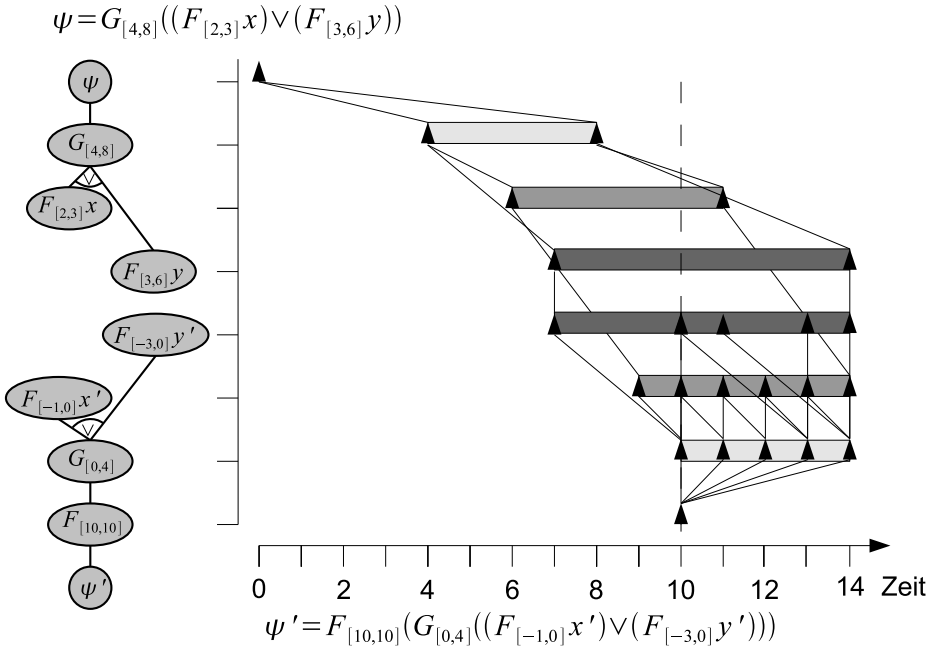


Abbildung 6.6.: Beispiel zur adaptiven Normalisierung von Formeln

Abbildung 6.6 zeigt dies an einem Beispiel. Die Formel $\psi = G_{[4,8]}((F_{[2,3]}x) \vee (F_{[3,6]}y))$ enthält als ersten den G-Operator in einem nicht verzweigten Teil, so dass der G-Operator zukunftsgerichtet bleiben kann. Demzufolge ergibt sich, nach Anpassung der Intervalle, der Zeitpunkt $t = 10$ als Referenzzeitpunkt. Es resultiert die normalisierte Formel $\psi' = F_{[10,10]}(G_{[0,4]}((F_{[-1,0]}x') \vee (F_{[-3,0]}y')))$ mit $y'(t) = y(t)$ und $x'(t) = x(t-3)$. Bereits zum Zeitpunkt $t = 10$ ist entscheid-

bar, ob die Formel noch erfüllbar ist, oder nicht. Eine Besonderheit der adaptiv normalisierten Formeln ist, dass der zukunftsgerichtete Operator auch zeitlich unbeschränkt sein darf.

6.3.3. Definition von Timed AR-Automata

Timed AR-Automata stellen eine abgewandelte Form der Timed Automata dar und können als 8-Tupel $TAR = (\Sigma, Z, z_0, C, A, E, I, L)$ beschrieben werden. Gemeinsam mit den Timed Automata sind die endliche Zustandsmenge Z, der Initialzustand z_0, Aktionen A sowie die Menge von Zeitvariablen C. Die hier verwendete Form besitzt Eingangsvariablen Σ, jedoch keine Ausgangsvariablen. Stattdessen wird jedem Zustand ein Label zugeordnet $L : Z \rightarrow l \in \{Accept, Reject, Pending\}$, wobei l den Zustand als Element der dreiwertigen Logik zurückliefert. $I : Z \rightarrow B(C)$ ordnet den Zuständen Invarianten zu. $E \subseteq Z \times \Sigma \times B(C) \times A \times Z \times P$ definiert eine Menge von Kanten zwischen Zuständen mit Signalbedingungen Σ, Bedingungen $B(C)$, Aktionen A sowie einer Priorität $p \in P \subset \mathbb{N}$. Die zusätzliche Priorität ist erforderlich, um auch dann einen deterministischen Automaten beschreiben zu können, wenn die Kantenbedingungen nicht disjunkt sind. Es muss also gelten $\forall e_1, e_2 \in E : (e_1 \neq e_2) \rightarrow (P(e_1) \neq P(e_2))$. Die Zeitvariablen nutzen einen Wertebereich $\{\mathbb{R}^+ \cup \#\}$, wobei $\#$ eine deaktivierte Zeitvariable bezeichnet. Aktionen $a : C \rightarrow \{\mathbb{Q}^+ \cup \#\}$ beschreiben Zuweisungen von Zeitvariablen auf die Werte der Menge $\{\mathbb{Q}^+ \cup \#\}$. Weiterhin können Ungleichungsbedingungen mit der Menge $\mathbb{Q}^+$ sowie Gleichungsbedingungen mit der Menge $\mathbb{Q}^+ \cup \#$ geprüft werden.

Die Semantik der Timed AR-Automaten wird im Folgenden definiert. Eine Clock-Evaluierung ist eine Funktion $u : C \rightarrow (\mathbb{R}^+ \cup \#)$, der die Menge von Zeitvariablen auf positive reelle Zahlen oder Raute abbildet. $\mathbb{R}^C$ beschreibt die Menge aller Clock-Evaluierungen. Es gilt $u_0(x) = 0$ für alle $x \in C$. Für die Bewertung von Clock-Evaluierungen durch Bedingungen und Invarianten wird abkürzend die Schreibweise $u \in I(z)$ genutzt, die besagt, dass $I(z)$ von u erfüllt wird. Für Ungleichungen gilt $x \circ \# = false$ für alle $x \in C$ und $\circ \in \{<, \leq, \geq, >\}$. Weiterhin gilt für alle $x \in C$ und $d \in \mathbb{Q}+$:

$$u(x) + d = \begin{cases} u(x) + d, & \text{falls } u(x) \neq \# \\ \#, & \text{sonst.} \end{cases}$$

Es sei $TAR = (\Sigma, Z, z_0, C, A, E, I, L)$ ein Timed AR-Automat. Die Semantik wird als Transitionssystem $\langle S, s_0, \rightarrow \rangle$ beschrieben, wobei $S \subseteq Z \times \mathbb{R}^C$ die Menge der Stellen, $s_o = (z_0, u_0)$ der Initialzustand und $\rightarrow \subseteq S \times \mathbb{R} \cup \Sigma \times S$ ein Transitionssystem ist, so dass:

$-(l, u) \xrightarrow{d} (l, u + d)$ falls $\forall d' : 0 \leq d' \leq d \implies u + d' \in I(z)$, und
$-(l, u) \xrightarrow{\sigma} (l', u')$ falls ein $e = (z, \sigma, g, a, z') \in E$ existiert, so dass
$u \in g, u' = [a]u$ und $u' \in I(z')$ gilt,

wobei für $d \in \mathbb{R}^+$, $u+d$ jeder Clock $x \in C$ den Wert $u(x)+d$ zuordnet, und [a]u die Zeitzuordnungen für jede Clock beschreibt, welche entweder in a enthalten sind oder anderenfalls aus u übernommen werden.

Die Transition können in Makroschritte und Mikroschritte unterteilt werden. Makroschritte bestehen aus Transitionen $\xrightarrow{d}$ mit $d > 0$, beschreiben also ein Voranschreiten der Zeit. Mikroschritte werden durch Transitionen $\xrightarrow{d}$ mit $d = 0$ oder Transitionen $\xrightarrow{\sigma}$ repräsentiert, laufen also ohne Voranschreiten der Zeit ab. Die Transition erfolgt dabei nach einer *Run-to-Completion* Semantik, was bedeutet, dass nach jedem Makroschritt zunächst Mikroschritte simuliert werden, bis ein stabiler Zustand erreicht ist, also keine weiteren Mikroschritte stattfinden können. Können bei einem Mikroschritt oder Makroschritt mehrere Transitionen Schalten, so wird der Konflikt durch die Prioritäten gelöst. Das Transitionssystem des Automaten ist dabei mit dem Zeitmodell der Eingangssignale verknüpft. Das bedeutet, dass die Zeit in der Simulation und im Automaten synchron laufen muss. Vergeht eine bestimmte Zeit Δt in der Simulation muss der Automat so viele Simulationsschritte $\xrightarrow{d}$ mit unbelegten Eingangssignalen absolvieren, dass $\Delta t = \sum^i d_i$ gilt. Anschließend erfolgt die Simulation mit dem neuen Eingangssignal und der synchronisierten Zeit Δt.

Aus der Run-to-Completion Semantik folgt die Bedingung, dass keine unendlichen Transitionsfolgen aus Mikroschritten existieren dürfen, da eine Simulation nicht terminieren würde. Es darf folglich der Zustand mit $u = \#^C$ nicht erreichbar sein, da sonst eine unendliche Folge von Mikroschritten $\xrightarrow{d}$ mit $d = 0$ existieren würde.

Bei der Steuerung eines Timed AR-Automaten durch Events, erfolgt bereits eine Diskretisierung der Zeit, vergleiche Abschnitt 4.5.1. Für die Bewertung von Timeouts müssen allerdings zusätzliche Zeitpunkte (Timeouts) bestimmt werden, zu denen Zustandsübergänge stattfinden müssen, um der Semantik der Timed Automata zu genügen. Das deterministische Verhalten (Reproduzierbarkeit) wird durch die Vergabe eineindeutiger Prioritäten sichergestellt.

6.3.4. Konstruktion von Timed AR-Automata

Die normalisierten XFLTL-Formeln stellen die zu prüfende XFLTL-Eigenschaft in deterministischer Form dar. In diesem Abschnitt wird die Konstruktion von Timed AR-Automata aus normalisierten XFLTL-Formeln betrachtet. Zunächst muss aus der Formel die Struktur des Automaten abgeleitet werden. Strukturell kann jedem temporalen Operator der Formel eine Menge von Zeitvariablen zugeordnet werden, die beschreiben, wie viel Zeit seit der Erfüllung der Eigenschaft vergangen ist. Die Auswertung der Eigenschaft durch den Automaten kann dann durch folgende Phasen erfolgen:

1. **Initialisierung der Zeitvariablen:** Der äußerste temporale Operator ist ein F-Operator mit dem Punktintervall zum Referenzzeitpunkt, die zugehörige Zeitvariable sei mit t_0 benannt. Daher werden initial alle Zeitvariablen mit Ausnahme von t_0 deaktiviert, so

dass initial kein Operator als erfüllt gilt.

2. **Monitoring der Operatoren:** Die Erfüllung der Operatoren wird geprüft, bis der Referenzzeitpunkt erreicht ist. Bei jeder erneuten Erfüllung eines Operators nach Voranschreiten der Zeit wird die zugehörige Zeitvariable auf 0 gesetzt.

3. **Gesamtbewertung:** Mit Erreichen des Referenzzeitpunktes wird geprüft, ob die Eigenschaft aufgrund des Zustandes der temporalen Operatoren insgesamt erfüllt ist, oder nicht.

Die Initialisierung erfolgt durch einen einzelnen Zustandsübergang. Für das Monitoring müssen die booleschen Verknüpfungen der temporalen Operatoren realisiert werden. Dazu können Zustandsübergänge als Eigenschleifen genutzt werden, die jeweils die logische Verknüpfung zwischen den Operatoren realisieren, indem eine Transition jeweils das Aktivieren oder Deaktivieren eines Operators überwacht. Wird durch die Bedingung einer Transition erkannt, dass alle erforderlichen Terme eines Operators erfüllt sind, so wird durch die Aktion dafür gesorgt, dass anschließend der Operator erfüllt ist. Umgekehrt gilt dasselbe. Wird durch die Bedingung erkannt, dass alle Terme eines Operators nicht erfüllt sind, indem beispielsweise ein Timeout erfolgt, so wird der Operator deaktiviert. Die Bewertung der Operatoren erfolgt dabei entsprechend der Abhängigkeiten der Operatoren, ausgehend von den Signalen zur Wurzel der Formel hin. Dadurch wird sichergestellt, dass die Zustände der betreffenden Zeitvariablen aktuell sind (z.B. zuerst der Timeout eines abhängigen Operators berücksichtigt wird). Zudem ergibt sich dadurch eine deterministische Reihenfolge, die zur Ermittlung der Prioritäten von Transitionen genutzt werden kann.

Variablen: Operator	t_0 : $F_{[-a,0]}\varphi$	t_0 : $G_{[-a,0]}\varphi$	t_{G_i}, t_{F_i} : $(\varphi_1)U_{[-a,0],[-b,0]}(\varphi_2)$
Bedingung erfüllt	$(0 \leq t_0 \leq a)$	$(t_0 \geq a)$	$\exists i : (t_{G_i} \geq a) \wedge (0 \leq t_{F_i} \leq b)$
Bedingung nicht erfüllt	$(t_0 = \#) \vee (t_0 > a)$	$(t_0 = \#) \vee (t_0 < a)$	$\forall i : (t_{G_i} < a) \vee (t_{F_i} > b) \vee$ $(t_{G_i} = \#) \vee (t_{F_i} = \#)$
Aktion Aktivierung	$\varphi \Rightarrow t_0 \mapsto 0$	$\varphi \Rightarrow t_0 \mapsto 0$	$\varphi_1 \Rightarrow (t_{G_i} \mapsto 0)$; $\varphi_1 \wedge \varphi_2 \Rightarrow (t_{F_i} \mapsto 0)$
Aktion Deaktivierung	$t_0 > a$ (Timeout)	$\neg\varphi \Rightarrow t_0 \mapsto \#$	$\neg\varphi_1 \Rightarrow i \mapsto i + 1$; $\forall i : (t_{F_i} > b) \Rightarrow t_{G_i} \mapsto \#, t_{F_i} \mapsto \#$

Tabelle 6.2.: Intervallbedingungen für die Verknüpfung temporaler Operatoren

Tabelle 6.2 zeigt für die temporalen Operatoren F, G und U jeweils die Bedingungen für die Auswertung, sowie die notwendigen Aktionen zur Aktivierung und Deaktivierung. Man erkennt, dass die Bewertung des *Until*-Operators komplizierter ist, da für die Terme φ_1 und φ_2 jeweils separate paarweise Zeitmessungen erforderlich sind. Zum einen ergibt sich ein Zeitintervall relativ zum Bezugszeitpunkt der Spezifikation, für das φ_1 gelten muss (Zeitschranke a). Zum anderen

muss innerhalb des ursprünglich definierten Zeitintervalls φ_2 erfüllt werden (Zeitschranke b). Die Notwendigkeit der paarweisen Verknüpfung folgt aus dem gemeinsamen Bezugszeitpunkt. Unter der Voraussetzung, dass φ_1 eine temporale Eigenschaft mit einer Intervalllänge $l > 0$ ist, ist die Anzahl der Paare von Zeitvariablen endlich.

Abbildung 6.7 zeigt die Konstruktion eines Timed AR-Automata am Beispiel der XFLTL-Formel $\psi' = F_{[14,14]}(G_{[-4,0]}((F_{[-1,0]}x')\vee(F_{[-3,0]}y')))$. Jedem Operator ist genau eine Zeitvariable (t_0, t_1, t_2, t_3) zugeordnet, wie in der Abbildung dargestellt. Zum Zeitpunkt $t = 0$ findet ein Zustandswechsel von S_0 nach S_1 statt, wobei die Zeitvariablen t_1, t_2, t_3 deaktiviert werden (Zuweisung mit dem Symbol #).

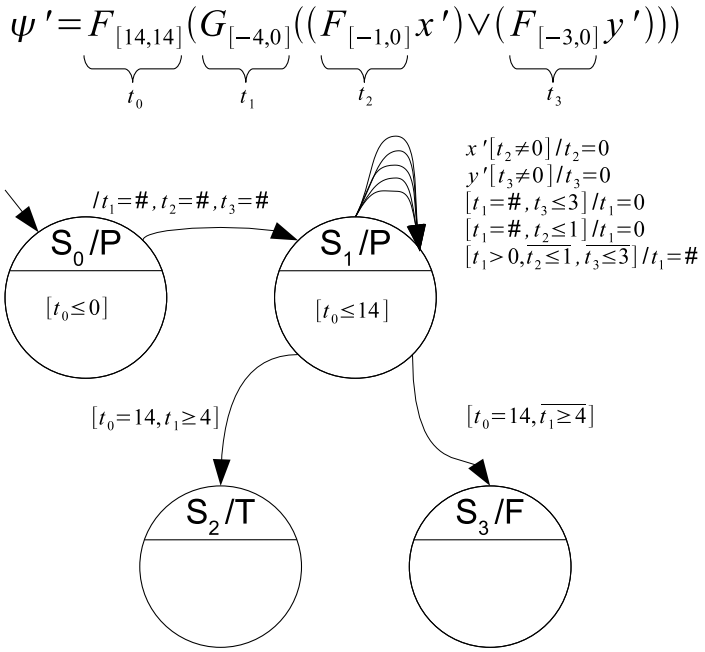


Abbildung 6.7.: Konstruktion eines Timed AR-Automata am Beispiel

Der Zustand S_1 besitzt mehrere Eigenschleifen. Von oben nach unten werden, entsprechend der Prioritäten, folgende Fälle berücksichtigt:

- Tritt ein Event x' auf, so wird die Zeitvariable t_2 auf 0 gesetzt. Damit ist $F_{[-1,0]}x'$ initial erfüllt.
- Tritt ein Event y' auf, so wird die Zeitvariable t_3 auf 0 gesetzt. Damit ist $F_{[-3,0]}y'$ initial erfüllt.
- Ist t_1 deaktiviert (der G-Operator nicht erfüllt), und der zu t_3 gehörige F-Operator aktiv, d.h. $t_3 < 3$, so wird begonnen, den G-Operator zu erfüllen, indem t_1 auf 0 gesetzt wird.
- Ist t_1 deaktiviert (der G-Operator nicht erfüllt), und der zu t_2 gehörige F-Operator aktiv, d.h. $t_2 < 3$, so wird begonnen, den G-Operator zu erfüllen, indem t_1 auf 0 gesetzt wird.

- Tritt der Fall ein, dass die beiden F-Operatoren gleichzeitig nicht erfüllt sind, sich die Zeitvariablen t_2 und t_3 also nicht im Intervall $[0, 1]$ bzw. $[0, 3]$ befinden, so ist der G-Operator nicht erfüllt, daher wird t_1 deaktiviert.

Ist der Referenzzeitpunkt $t = 14$ erreicht, so wird anhand der Zeitvariablen t_1 entschieden, ob die Eigenschaft erfüllt ist oder nicht. Gilt $t_1 > 4$, so ist die Eigenschaft erfüllt und es findet ein Zustandswechsel nach S_2 statt, ansonsten wird in den Zustand S_3 gewechselt. Der Zustand der Eigenschaft ist als Label jeweils dem Zustand des Automaten zugeordnet. Durch Invarianten bezüglich t_0 wird jeweils der Zustandswechsel erzwungen.

Timed AR-Automata für die adaptive Normalisierung

Adaptiv normalisierte Formeln beschreiben eine Eigenschaft so, dass kein fixer Zeitpunkt für die Entscheidung bereitgestellt wird. Stattdessen wird ab dem Referenzzeitpunkt geprüft, ob die Formel bereits endgültig als erfüllt oder nicht erfüllt bewertet werden kann. Dadurch ergibt sich eine leicht abgeänderte Konstruktion des Timed AR-Automata für adaptiv normalisierte XFLTL-Eigenschaften.

Verkettung von Timed AR-Automata

Die Struktur der XFLTL-Eigenschaften erlaubt eine Verkettung mittels der Suffix-Implikation strukturell zu realisieren. Bei einer Suffix-Implikation wird erst nach der Erfüllung der Prämisse begonnen die Konklusion auszuwerten. Diese Art der Notation ist vor allem für lineare Abläufe entsprechend einer SERE[Acc04] geeignet. Lassen sich die zu verknüpfenden Teilausdrücke in normalisierter oder adaptiv normalisierter Form notieren, so können jeweils Timed AR-Automata erstellt werden. Entsprechend der Verknüpfung kann dann der Initialzustand der Konklusion mit dem als *Accept* (True) bewerteten Zustand der Prämisse vereinigt werden. Das Label dieses Verknüpfungszustandes wird dabei entfernt bzw. auf *Pending* gesetzt. Sämtliche Zustandsübergänge, die in den Verknüpfungszustand führen, müssen durch geeignete Aktionen die Zeitvariablen der Konklusion auf 0 setzen.

6.3.5. Interpretation heterogener Eigenschaften

Mit der bereitgestellten Methodik können XFLTL-Eigenschaften für Eventsignale beschrieben werden. Zudem ist es möglich, analoge Signale entsprechend Abschnitt 4.5.4 zu verwenden. Für die Verwendung digitaler Signale, bezogen auf eine Clock gemäß Abschnitt 4.5.5, muss der Konstruktionsmechanismus erweitert werden. Temporale Eigenschaften digitaler Signale werden modelliert, indem die Anzahl von Events betrachtet wird. Die Referenzen bezüglich der kontinuierlichen Zeit sind dabei nebensächlich. Die Erweiterung des Verfahrens kann äquivalent zum Ansatz [WRKR05] erfolgen.

Für die Verbindung von analogen und digitalen Eigenschaften zu heterogenen Eigenschaften müssen die jeweils zugrunde liegenden Zeitmodelle verknüpft werden. Insbesondere muss die Verschiebung der Signale für die heterogene Eigenschaft koordiniert werden. Erfolgt die Verknüpfung zwischen analogen und digitalen Signalen über die Suffix-Implikation [Acc04], so lassen sich bei Verwendung einer Notation mit Eventsignalen die Verknüpfungspunkte als Events bestimmen [JLP+08].

Die Verwendung der Implikation erlaubt die Zerlegung in Vor- und Nachbedingung, wobei sich die Bedingungen in der Regel auf Intervalle beziehen. Daraus ergeben sich die Verknüpfungspunkte und Eigenschaften wie folgt.

- Bestimmung des Startzeitpunktes für die Vorbedingung: $e_{V_{start}}$.
- Bestimmung des Endzeitpunktes für die Vorbedingung: $e_{V_{ende}}$.
- Bestimmung des Startzeitpunktes für die Nachbedingung: $e_{N_{start}}$.
- Bestimmung der Intervallbedingung für die Nachbedingung: φ_N.
- Beschreibung der Eigenschaft *psi* über die Verknüpfungszeitpunkte:
  $\psi = (e_{V_{start}} \implies e_{N_{start}}) \wedge (e_{V_{start}} \implies (\varphi_N U_{[0,infty)} e_{V_{ende}}))$
  $\psi = \psi_1 \wedge \psi_2; \psi_1 = (e_{V_{start}} \implies e_{N_{start}}); \psi_2 = (e_{V_{start}} \implies (\varphi_N U_{[0,infty)} e_{V_{ende}}))$

Damit lassen sich durch Implikation verknüpfte heterogene Eigenschaften durch eventbasierte Eigenschaften notieren und verknüpfen. Zudem ist eine Zerlegung in zwei Eigenschaften ψ_1 und ψ_2, und somit die Prüfung durch zwei Automaten möglich. Eine Besonderheit ergibt sich durch den zeitlich unbeschränkten $U-$Operator, der durch das Endeevent der Vorbedingung beendet wird. Dieser Operator muss also in adaptiv Normalisierter Form vorliegen. Da $e_{V_{start}}$ vor $e_{V_{ende}}$ liegen muss, lässt sich die Eigenschaft durch Verkettung realisieren, so dass der $U-$Operator in adaptiv Normalisierter Form, also durch Verkettung von Timed AR-Automata realisiert werden kann.

6.3.6. Schlussfolgerungen

Zunächst wurde betrachtet, welche Voraussetzungen für die symbolische Beschreibung der XFLTL-Eigenschaften erfüllt sein müssen. Dies betrifft vor allem die Auflösung des Nichtdeterminismus bezüglich möglicher Folgezustände. Eine Lösung besteht in der zeitlichen Verschiebung von Eventsignalen und der Normalisierung der verwendeten Intervalle der temporalen Operatoren. Dadurch wird eine deterministische zeitliche Partitionierung erreicht. Weiterhin wurde ein Timed AR-Automat für die symbolische Beschreibung der Eigenschaften definiert. Dieser erweitert Timed Automata um Labels für die Zustände sowie um deaktivierbare Zeitvariablen. Durch das schrittweise Vorgehen können normalisierte XFLTL-Eigenschaften in Timed AR-Automata umgewandelt werden. Des Weiteren ist eine Verkettung von XFLTL-Eigenschaften

auf der Basis der Timed AR-Automata möglich. Es können heterogene Eigenschaften symbolisch repräsentiert werden.

Damit steht eine Methode zur Verfügung, mit der XFLTL-Eigenschaften in eine symbolische Darstellung in Form erweiterter Automaten transformiert werden können. Somit wird eine Eigenschaftsprüfung durch Cosimulation ermöglicht.

Die Einschränkungen der algorithmischen Prüfung werden durch die Normalisierung umgangen, allerdings erfolgt die Auswertung der Eigenschaften nicht mehr zum frühesten möglichen Zeitpunkt, sondern zeitlich versetzt.

Der Speicherbedarf für die symbolische Repräsentation ist beschränkt und nur von der Eigenschaft selbst abhängig. Allerdings ist zusätzlich Speicher für die Verzögerung der Eventsignale erforderlich.

6.4. Prototyp zur Assertion-basierten Verifikation durch algorithmische Prüfung

Der entwickelte Prototyp verwendet das Werkzeug MLDesigner und wertet die auftretenden Signale und Ereignisse aus. Dabei kommt die Eigenschaftsdefinition aus Abschnitt 4.5.2 zum Einsatz.

Für die Assertion basierte Verifikation werden die Eigenschaften jedoch nicht symbolisch, sondern algorithmisch geprüft, wie in Abschnitt 6.2 beschrieben.

Der Prototyp besteht aus einem MLDesigner-Primitiv, das die Kopplung mit dem Simulationsmodell realisiert und die notierten Eigenschaften übernimmt (*XFLTL-Primitiv*), einer C++-Klassenbibliothek zur Verwaltung und Prüfung der Formelobjekte (*XFLTL-Klassenbibliothek*), sowie aus einer Visualisierung der Ergebnisse in der Modellierungsumgebung (*Visualisierung*).

Der Nutzer muss zunächst die betroffenen Signale des zu analysierenden Modells mit dem *XFLTL-Primitiv* verbinden. Anschließend muss der Nutzer die Signale entsprechend der Verwendung in der Eigenschaft benennen. Anschließend sind eine oder mehrere Eigenschaften als XFLTL-Formeln, entsprechend einer in EBNF definierten Syntax, anzugeben. Vor Beginn der Simulation werden der Signalvektor und die Formeln analysiert, instantiiert, und der Klassenbibliothek übergeben. Syntaxfehler werden ggf. dem Nutzer gemeldet. Während der Simulation werden entsprechend dem Verlauf der Eingangssignale die Eigenschaften geprüft und visualisiert. Eine detaillierte Auswertung ist schließlich über das Logging der Analyse möglich.

6.4.1. XFLTL-Primitiv

Das XFLTL-Primitiv stellt die Nutzerschnittstelle dar und dient gleichzeitig der Anbindung des Simulationsmodells an die XFLTL-Klassenbibliothek sowie zur Anbindung an die Visualisierung.

An der Nutzerschnittstelle müssen die zu prüfenden Eigenschaften übergeben werden. Da die Modellierungsplattform MLDesigner nicht direkt erweiterbar ist, sondern nur die zu simulierenden Modelle, können die Information nur als Parameter übergeben werden. Allerdings werden die Syntax der annotierten Eigenschaften, sowie die Verwendung gültiger Signalnamen geprüft, siehe auch Anhang A.2 und Anhang C.

Die Nutzung der XFLTL-Klassenbibliothek muss über das XFLTL-Primitiv initiiert werden. Dies bedeutet, dass sämtliche Informationen über Signale und Eigenschaften ebenfalls weitergereicht werden müssen. Daher wird eine Datenstruktur für die Signale vor Beginn der Simulation erzeugt, so dass die an den Eingangsports des Primitivs auftretenden Signale über die vom Nutzer vergebenen Signalnamen ansprechbar werden. Dabei sind jeweils der Eventstatus und ein skalarer Datenwert abfragbar. Weiterhin werden die eingegebenen Formeln geparst und entsprechende Objektstrukturen vor Beginn der Simulation instantiiert und initialisiert.

Zur Laufzeit der Simulation wird die in Abschnitt 6.2.4 beschriebene Synchronisation gleichzeitig auftretender Ereignisse realisiert. Dies bedeutet, dass mit jedem eingehenden Event[69] geprüft wird, ob die Zeit seit dem letzten Event vorangeschritten ist. Wenn ja, wird der Signalvektor mit der alten Signalbelegung und dem alten Timestamp geprüft und anschließend eine neue Signalbelegung erzeugt, anderenfalls wird das Signal der Signalbelegung hinzugefügt. Nach erfolgter Prüfung wird das Ergebnis schließlich der Visualisierung zugeführt. Diese Synchronisation wird von der Klasse `XFLTL_event_handler` der XFLTL-Klassenbibliothek realisiert.

6.4.2. XFLTL-Klassenbibliothek

Die Prüfung der normalisierten Formeln erfolgt durch Instantiierung der logischen und booleschen Operatoren als Objekte. Diese Objekte werden als Baum, entsprechend der zu prüfenden Formel strukturiert und ausgehend von der Wurzel ausgewertet. Programmiertechnisch sind alle Operatoren von der abstrakten Klasse `xfltl` abgeleitet, die eine anonyme Formel repräsentiert. Wesentlich ist hierbei, dass jeder Operator die Referenz zu einer globalen Zeitbasis enthält und die Methode `check()` besitzt, die einen Rückgabewert in dreiwertiger Logik liefert, siehe Abbildung 6.8.

Die Verknüpfung zum Modell, erfolgt mittels *Signalinstanzen*, implementiert als Operatoren des Typs `xfltl_event`. Diese stellen bezüglich des Formelbaumes die Blätter dar und repräsentieren den Eventstatus zum Auswertezeitpunkt der *Signalinstanz*. Durch den erweiterten Operator `xfltl_cmpevent` besteht zusätzlich zur Eventprüfung die Möglichkeit, einfache skalare Wertprüfungen ($<, \leq, =, \neq, \geq, >$) zu realisieren. Die booleschen Operatoren erfüllen die dreiwertige Logik entsprechend Tabelle 6.1. Die temporalen Operatoren (Instanzen) implementieren die Zustandsautomaten gemäß Abbildung 6.3 und Abbildung 6.4.

Die temporale Operatoren *Finally*, *Generally*, und *Until* liegen als Implementierung vor.

[69] Aufruf der Simulationsmethode `go()` des Primitivs.

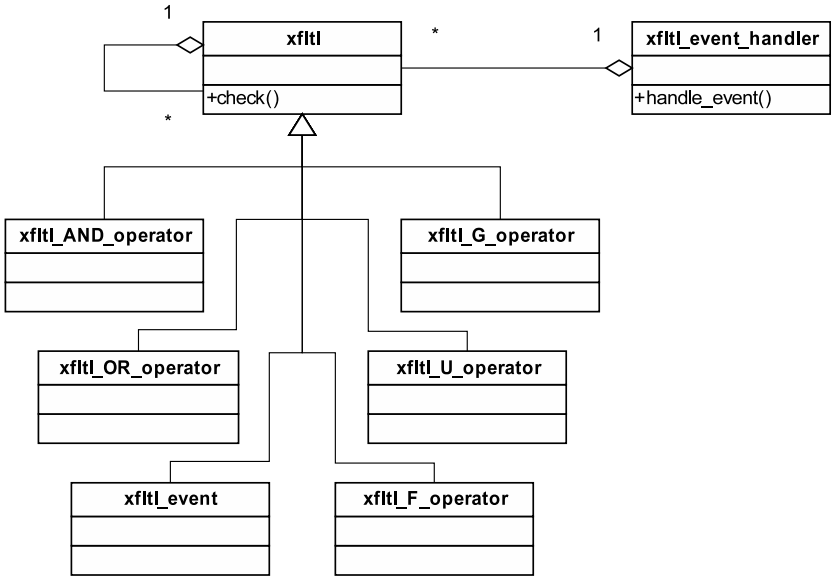


Abbildung 6.8.: Klassendiagramm der XFLTL-Bibliothek

Dazu kommt der *Assert* Operator, siehe Abschnitt 6.2.5, der eine einfachere Instantiierung und Auswertung der Eigenschaften ermöglicht. Eine Besonderheit nimmt der *Next*-Operator ein. Der *Next*-Operator beschreibt in einem diskreten Zeitmodell den nächsten Zeitpunkt, was in einem kontinuierlichen Zeitmodell nicht möglich ist. Stattdessen wird er verwendet, um von einem konkreten Zeitpunkt aus das nächste Intervall zu referenzieren, er sorgt also dafür, dass das Bezugsintervall des eingeschlossenen Terms offen ist, etwa in der Gleichung $\varphi = (e_x) \implies (X \neg e_x)$.

Neben den Operatoren, die sämtlich von der Klasse `xfltl` abgeleitet sind, existiert noch eine globale Definition für die Datentypen sowie die Klasse `xfltl_event_handler`. Diese Klasse steuert den eigentlichen Ablauf der Prüfung, verwaltet intern die Datenstrukturen, nimmt den Signalvektor vom *XFLTL-Primitiv* entgegen und liefert Ergebnisse an das *XFLTL-Primitiv* zurück.

6.4.3. Visualisierung

Die Visualisierung dient dazu, dem Nutzer die Ergebnisse der Prüfung zugänglich zu machen und Statusinformationen zu liefern. Dazu werden Statusinformationen von der *XFLTL-Klassenbibliothek* zum *XFLTL-Primitiv* übergeben und von dort aus an die Visualisierung weitergeleitet.

In Abschnitt 6.2.5 wurde skizziert, welche Informationen von Interesse sind. Bisher wird entsprechend den Eigenschaften ein Ergebnisvektor zurückgeliefert, der für jede zu prüfende Eigenschaft Zustand, Laufzeit und Speicherbedarf enthält. Dadurch ist der Zustand der Eigen-

schaften in Abhängigkeit von der Zeit darstellbar.

Abbildung 6.9 zeigt einen Screenshot der Statusanzeige der Eigenschaften. Diese Ansicht zeigt eine Tabelle mit den wesentlichen Informationen. In der ersten Spalte ist die Eigenschaft textuell als Formel dargestellt. Daneben sind Startzeitpunkt und Endzeitpunkt der Bewertung zu finden, die "−1.0" deutet an, dass die letzte Eigenschaft deaktiviert ist und die anderen Eigenschaften noch nicht zu Ende bewertet sind, also noch vom Verlauf der Simulation abhängen. Die letzten beiden Spalten zeigen die Anzahl der momentan aktiven Formelobjekte sowie den aktuellen und maximalen Speicherverbrauch in Bytes an.

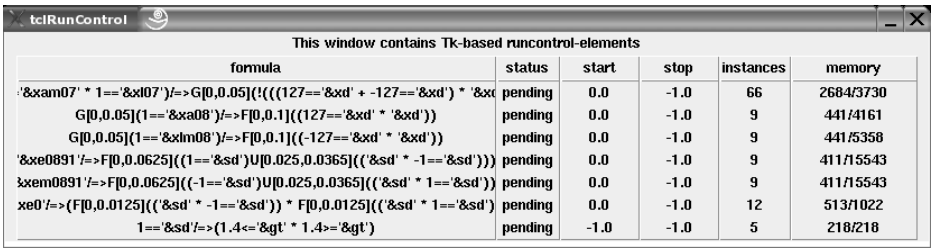


Abbildung 6.9.: Screenshot: Statusanzeige der Eigenschaften

Informationen des *Assert*-Operators, wie die paarweise Verknüpfung von Trigger und Eigenschaft, werden derzeit nicht visualisiert. Stattdessen können Logging-Informationen ausgewertet werden.

6.4.4. Erweiterungsmöglichkeiten

Im Prototyp ist die wesentliche Funktionalität umgesetzt. Allerdings bestehen noch Möglichkeiten zur Erweiterung, die vor allem die Nutzerfreundlichkeit und Integration in die Arbeitsabläufe des Benutzers verbessern.

Grundlegend ist denkbar, dass die Anbindung an das Modellierungs- und Simulationswerkzeug nicht direkt als Teilmodell realisiert ist, sondern automatisch erzeugt wird. Dadurch würde eine bessere Trennung von Modell und Analyse realisiert und die Nutzbarkeit verbessert werden. Zudem wäre dann eine direkte Zuordnung der Assertions zu Modellelementen möglich.

Im Bereich der Auswertung besteht noch Verbesserungsbedarf. So werden bestimmte Informationen derzeit nur über das Logging zugänglich. Eine systematische Zuordnung der Informationen zu den Assertions und eine übersichtliche, möglicherweise grafische Darstellung der Informationen würden den Nutzen deutlich erhöhen.

6.5. Validierung der Methode zur Assertion-basierten Verifikation am Beispiel

Die Validierung der Methode erfolgte anhand eines Modells eines Σ/Δ AD-Wandlers erster Ordnung. Das Beispiel wurde u.a. in [JLP+07] vorgestellt.

Ein Σ/Δ-Konverter ist ein repräsentatives Beispiel für ein Mixed-Signal System, das analoge Signale in digitale Signale konvertiert. Der Σ/Δ-Konverter besteht aus komplexen analogen und digitalen Komponenten, die über Schnittstellen miteinander verbunden sind. Das Modell ist mit dem Modellierungswerkzeug MLDesigner in den Domänen CTDE, DE und SDF modelliert worden, gemäß dem Modellierungsschema aus Abschnitt 4.4.

6.5.1. Σ/Δ-Konverter

In Abbildung 6.10 ist der Σ/Δ-Konverter erster Ordnung schematisch dargestellt, der als Beispiel herangezogen wurde. Der Σ/Δ-Konverter wandelt ein zeit- und wertekontinuierliches Signal $x(t)$ in ein zeit- und wertediskretes Signal $\hat{\mathbf{x}}(t_n)$ fester Bitbreite, das den Zeitpunkten t_n, abhängig von der Samplingfrequenz f_s zugeordnet ist. Der betrachtete Σ/Δ-Konverter besteht aus einem Subtrahierer, einem Integrator, einem Verstärker, einem 1-Bit Quantifizierer (A/D-Wandler), einem digitalen Tiefpass-Filter, einem 1-Bit D/A-Wandler, einem Dezimationsfilter und einer Sample&Hold Stufe. Die Verbindung zwischen analogem und digitalem Teil wird durch den 1-Bit Quantifizierer hergestellt, der das analoge Signal in einen booleschen Wert überführt, der die Werte -1 und 1 repräsentiert. Die Verbindung zwischen digitalem und analogem Teil wird durch einen einfachen 1-Bit D/A-Wandler hergestellt, der das digitale Signal $\hat{s}(t_n)$ in ein kontinuierliches Signal überführt, das für den analogen Subtrahierer aufbereitet wird, siehe Signal $g(t)$.

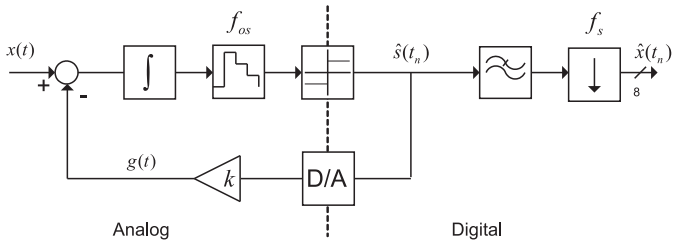


Abbildung 6.10.: Σ/Δ - Konverter.

Das Beispielmodell nutzt eine Oversamplingfrequenz von $f_{os} = 8 \cdot f_s = 160$ Hz. Die finale Samplingfrequenz beträgt $f_s = 20$ Hz und wird durch den Dezimationsfilter erreicht. Der Verstärkungsfaktor beträgt $k = 1.4$. Als Eingangssignal werden überlagerte Sinussignale mit $A_1 = 1, A_2 = \frac{1}{4}, f_1 = 1$ Hz, und $f_2 = 3$ Hz benutzt, wie Abbildung 6.13 zeigt. Das digitale Ausgangssignal $\hat{\mathbf{x}}(t_n)$ besitzt eine Auflösung von 8 Bit, beschrieben durch $\hat{x}_7(t_n), \hat{x}_6(t_n), \ldots, \hat{x}_0(t_n)$,

wobei $\hat{x}_7(t_n)$ das höherwertigste Bit darstellt. Es werden Beobachtungspunkte an wichtigen Signalen verwendet, um das Gesamtverhalten beobachten zu können. Die ersten beiden Punkte befinden sich am Eingangssignal $x(t)$ und am Ausgangssignal $\hat{\mathbf{x}}(t_n)$. Der dritte Beobachtungspunkt befindet sich hinter dem 1-Bit Quantifizierer am Bitstrom $\hat{s}(t_n)$. Der letzte Punkt beobachtet Signal $g(t)$, der sich hinter dem Verstärker befindet. Abbildung 6.13 zeigt die beobachteten Signale der ersten drei Monitorpunkte $x(t), \hat{\mathbf{x}}(t_n)$, und $\hat{s}(t_n)$.

6.5.2. Simulationsumgebung

Das Modell ist mit dem Modellierungswerkzeug MLDesigner in den Domänen CTDE, DE, und SDF modelliert worden, gemäß dem Modellierungsschema (Domänenhierarchie) aus Abschnitt 4.4. Die Domäne CTDE wird für analoge Komponenten und Mixed-Signal Komponenten verwendet. Die SDF-Domäne wird die funktionale Realisierung der digitalen Komponenten, Tiefpassfilter und Dezimationsfilter, verwendet, wobei die Verknüpfung und Synchronisation über Signale der DE-Domäne erfolgt. Abbildung 6.11 zeigt das Gesamtmodell. Zu erkennen sind die mit Piktogrammen versehenden Bestandteile des Σ/Δ-Modulators, links die Komponenten zur Generierung des Eingangssignals $x(t)$, außen Module zur Visualisierung der Signalverläufe, sowie unten das Modul zur Verifikation.

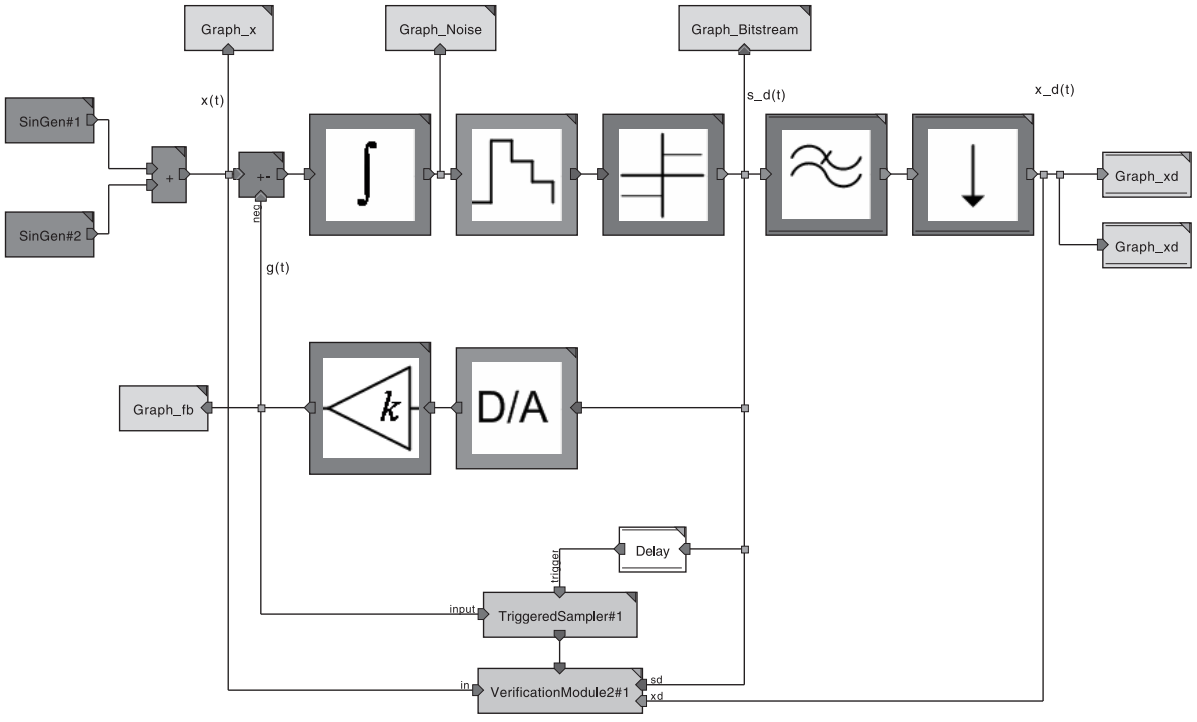


Abbildung 6.11.: MLD-Modell des Σ/Δ-Konverters

Für das Assertion-Checking wird der Prototyp entsprechend Abschnitt 6.4 verwendet. Die Anbindung erfolgt dabei über die DE-Domäne, also über das Event-basierte Signalmodell gemäß Abschnitt 4.5. Um analoge Signale beobachten zu können, sind zusätzliche Modellelemente (Komparatoren) in der CTDE-Domäne erforderlich, die ein Eventsignal erzeugen und dafür sorgen, dass durch den Solver die interessierenden Zeitpunkte korrekt berechnet werden, siehe dazu Abschnitt 4.5.4. Abbildung 6.12 zeigt, wie das Eingangssignal $x(t)$, im Bild mit *in* bezeichnet, mehreren Komparatoren zugeführt wird, die jeweils ein Eventsignal für den Vergleich ($<$, $=$, $>$) zurückliefern.

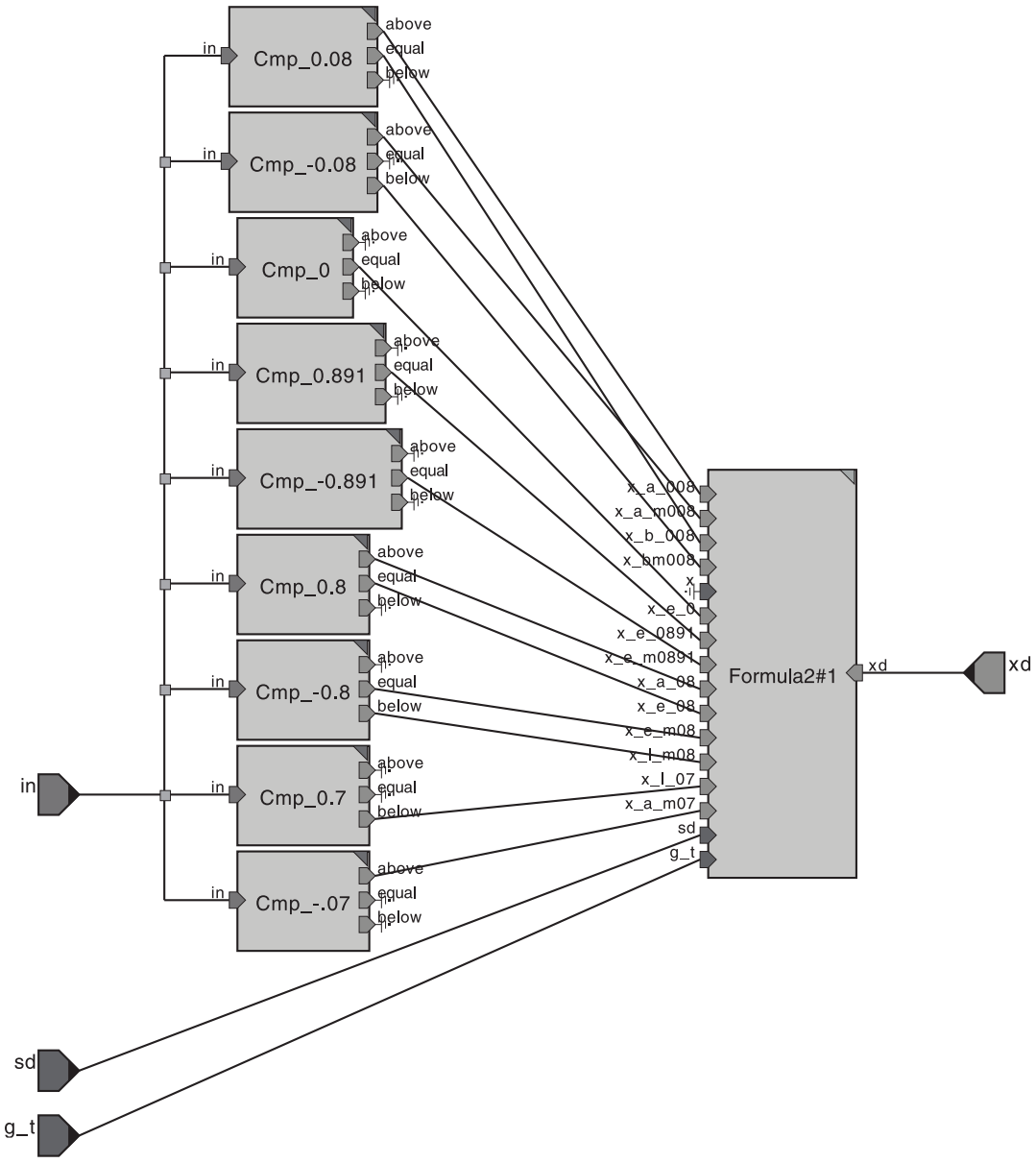


Abbildung 6.12.: Event-bezogene Diskretisierung der analogen Signale

6.5.3. Definition der Assertions

Es werden mehrere Assertions definiert, die temporale Eigenschaften des Σ/Δ-Konverters beschreiben. Die im Folgenden dargestellten Assertions in der Syntax der PSL beschrieben, wobei die SERE-Anweisungen $next\_e$ für *Finally* und $next_a$ für *Generally* verwendet werden. Zusätzlich wird jeweils die genutzte Event-basierte Assertion dargestellt. Dabei wird das Symbol $\mapsto$ für den Assert-Operator, also zur Trennung von Vor- und Nachbedingung verwendet. Für den Vergleich wird das Symbol $\cong$ verwendet.

Durch die erste Assertion werden nicht erreichbare Zustände des Ausgangssignals in Abhängigkeit vom Eingangssignal geprüft. Befindet sich das Eingangssignal im Bereich $(-0.7, 0.7)$, so erreicht das Ausgangssignal niemals ein Maximalwert innerhalb der Samplingperiode von $T_s = 0.05$ Sekunden, bezogen auf die Samplingfrequenz. Bei einer Darstellung im Zweierkomplement ergibt sich folgende Mixed-Signal Assertion:

$$
\begin{aligned}
&\{x(t) < 0.7 \& x(t) > -0.7\} \mapsto \\
&\{next\_e[1:2]!((\hat{x}_0(t_n)\&\hat{x}_1(t_n)\&\hat{x}_2(t_n)\&\hat{x}_3(t_n)\& \\
&\hat{x}_4(t_n)\&\hat{x}_5(t_n)\&\hat{x}_6(t_n)\&!\hat{x}_7(t_n))|| \\
&(!\hat{x}_0(t_n)\&!\hat{x}_1(t_n)\&!\hat{x}_2(t_n)\&!\hat{x}_3(t_n)\&!\hat{x}_4(t_n)\& \\
&!\hat{x}_5(t_n)\&!\hat{x}_6(t_n)\&\hat{x}_7(t_n)))\}
\end{aligned}
$$

Daraus kann unter Berücksichtigung der Diskretisierung die Event-basierte Assertion (6.6) abgeleitet werden. Für die Vergleiche mit dem analogen Eingangssignal werden die korrespondierenden Eventsignale der Konverter genutzt. Die Prüfung der Grenzwerte des digitalen Ausgangssignals erfolgt direkt durch Vergleich des ganzzahligen Wertes durch den Prototyp.

$$\varphi_1 = (\hat{x} \wedge (x > -0.7) \wedge (x < 0.7)) \mapsto G_{[0,0.05]}(\neg\hat{x} \wedge ((\hat{x} \cong 127) \vee (\hat{x} \cong .127))) \tag{6.6}$$

Die nächsten Assertions prüfen den Wert des Ausgangssignals bei Erreichen des Maximalen Eingangssignals $x(t) = 0.891$. Das diskrete Ausgangssignal muss dann den Maximalwert erreichen, so dass alle Bits außer dem Vorzeichenbit den Wert *1* annehmen. Selbiges gilt umgekehrt für den minimalen Eingangswert. Die Eigenschaften werden durch die Mixed-Signal Assertions 1 und 2 beschrieben.

$$
\begin{aligned}
&\text{2: } \{next\_a[0.0:0.05](x(t) > 0.8)\} \mapsto \\
&\{next\_e[2](\hat{\mathbf{x}}(t_n) = \hat{x}_0(t_n)\&\hat{x}_1(t_n)\&\hat{x}_2(t_n)\& \\
&\hat{x}_3(t_n)\&\hat{x}_4(t_n)\&\hat{x}_5(t_n)\&\hat{x}_6(t_n)\&!\hat{x}_7(t_n))\} \\
&\text{3: } \{next\_a[0.0:0.05](x(t) < -0.8)\} \mapsto \\
&\{next\_e[2](\hat{\mathbf{x}}(t_n) =!\hat{x}_0(t_n)\&!\hat{x}_1(t_n)\&!\hat{x}_2(t_n)\& \\
&!\hat{x}_3(t_n)\&!\hat{x}_4(t_n)\&!\hat{x}_5(t_n)\&!\hat{x}_6(t_n)\&\hat{x}_7(t_n))\}
\end{aligned}
$$

Die entsprechenden Event-basierten Assertions (6.7) und (6.7) ergeben sich wie folgt.

$$\varphi_2 = (G[0, 0.05]((x > 0.8))) \Rightarrow F[0, 0.1]((127 \cong \hat{x}) \wedge \hat{x}) \tag{6.7}$$
$$\varphi_3 = (G[0, 0.05]((x < -0.8))) \Rightarrow F[0, 0.2]((-127 \cong \hat{x}) \wedge \hat{x}) \tag{6.8}$$

Durch die nächsten Assertions prüfen drei Eigenschaften am Beobachtungspunkt $\hat{s}(t_n)$. Wird der maximale Eingangswert $x(t) = 0.891$ erreicht, so sollte der Bitstrom $\hat{s}(t_n)$ (Taktrate $1/f_{os}$) innerhalb der nächsten 10 Takte für 4 Takte (0.025 Sekunden) den High-Pegel annehmen und dann für einen Takt (6.25×10^{-3} Sekunden) Low-Pegel annehmen, siehe Assertion 4. Für den Fall $x(t) = -0.891$ gilt das Ganze invertiert, siehe Assertion 5. Ein weiterer Spezialfall gilt bei $x(t) = 0$, als dessen Konsequenz am Signal $\hat{s}(t_n)$ innerhalb der nächsten Zwei Takte ein Flankenwechsel beobachtbar sein muss, jeweils für einen Takt (6.25×10^{-3} Sekunden) muss also Low-Pegel bzw. High-Pegel anliegen, siehe Assertion 6. Diese Assertions sehen wie folgt aus:

4: $\{x(t) = 0.891\} \mapsto$ 1
$\{next\_e[10](next\_a[4](\hat{s}(t_n))\&$ 2
$next\_e[4:5](!\hat{s}(t_n)))\}$ 3
5: $\{x(t) = -0.891\} \mapsto$ 4
$\{next\_e[10](next\_a[4](!\hat{s}(t_n))\&$ 5
$next\_e[4:5](\hat{s}(t_n)))\}$ 6
6: $\{x(t) = 0\}|\mapsto$ 7
$\{next\_e[1:2](\hat{s}(t_n))\&next\_e[1:2](!\hat{s}(t_n))\}$ 8

Die Event-basierten Assertions ergeben sich dann wie folgt:

$$\varphi_4 = (x \cong 0.891) \Rightarrow (F[0, 0.0625]((1 = \hat{s})U[0.025, 0.0365](\hat{s} \wedge (-1 \cong \hat{s})))) \tag{6.9}$$
$$\varphi_5 = (x \cong -0.891) \Rightarrow (F[0, 0.0625]((-1 = \hat{s})U[0.025, 0.0365](\hat{s} \wedge (1 \cong \hat{s})))) \tag{6.10}$$
$$\varphi_6 = (x \cong 0) \Rightarrow ((F[0, 0.0125](\hat{s} \wedge (-1 = \hat{s}))) \wedge (F[0, 0.0125](\hat{s} \wedge (1 \cong \hat{s})))) \tag{6.11}$$

6.5.4. Ergebnisse und Bewertung

Die Abbildung 6.13 zeigt den Signalverlauf an den Beobachtungspunkten $x(t), \hat{s}(t_n)$, und $\hat{\mathbf{x}}(t_n)$. Oben ist das Eingangssignal $x(t)$ dargestellt. In der Mitte ist das digitale Signal $\hat{x}(t_n)$ aufgeschlüsselt in die 8 Bits $\hat{x}_0(t_n), \ldots, \hat{x}_7(t_n)$ zu sehen. Der unterste Teil zeigt das Signal $\hat{s}(t_n)$ welches ein 1−Bit Signal, repräsentiert durch die Werte -1 (low) und 1 (high) darstellt. Die gestrichelte Box zeigt die Funktionsweise der Assertion 1. Die Vorbedingung beschreibt einen Eingangsbereich des Signals $x(t)$ von -0.7 bis 0.7, hier ist der Übersichtlichkeit wegen nur ein Zeitpunkt dargestellt. Die Nachbedingung wird durch eine Zeitverzögerung von einer Taktperiode definiert, was als Verschiebung in der Box in der mittleren Abbildung angedeutet ist. Die Nachbedingung wird innerhalb des gesamten betrachteten Zeitfensters erfüllt.

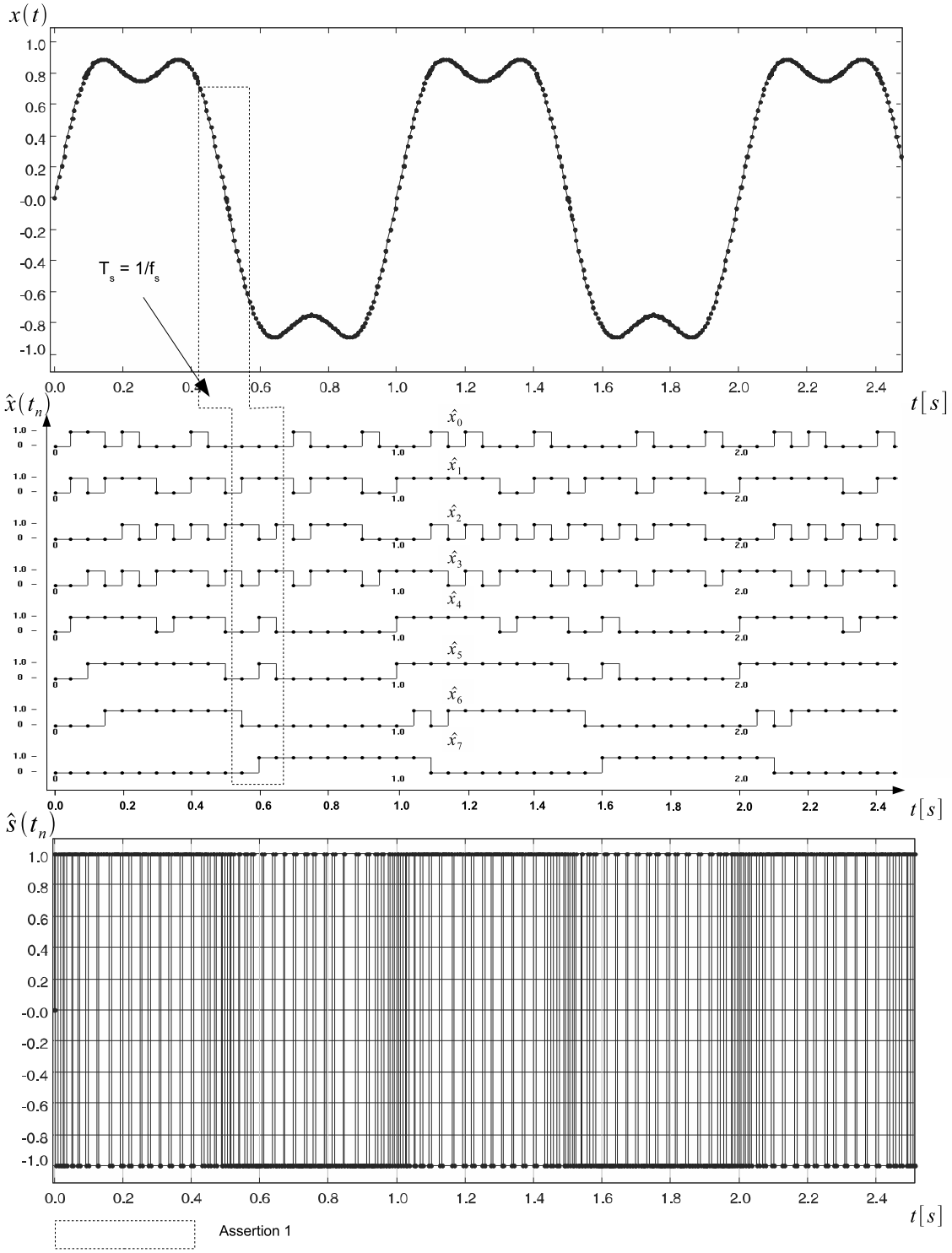


Abbildung 6.13.: Signalverlauf der genutzten Beobachtungspunkte beim Σ/Δ-Konverter und Beispiel der Assertion 1 (gestrichelte Box).

Tabelle 6.3 fasst die Ergebnisse der Assertions 1-6 zusammen. Die erste Spalte zeigt die Assertions mit der zugehörigen Taktreferenz, f_s bzw. f_{os}. Die Ergebnisse der Assertion-basierten Prüfung sind in den nachfolgenden Spalten dargestellt. Dabei steht an erster Stelle ein **A** für Accept (Eigenschaft erfüllt) und **R** für Reject (Eigenschaft nicht erfüllt), gefolgt von dem Taktzyklus, zu dem die Erfüllung oder Nichterfüllung beobachtet wurde. Die Beobachtungen wurden den Einträgen des Logfiles zum Assert-Operator entnommen.

MSA	Eigenschaft **A**ccept/**R**eject in t_{clk}				
	1	**2**	**3**	**4**	**5**
1 (f_s)	A,1	A,2	A,10	A,11	A,12
2 (f_s)	A,4	A,8	A,24	A,28	A,44
3 (f_s)	A,14	A,18	A,34	A,38	A,54
4 (f_{os})	A,26	A,31	A,69	A,186	A,191
5 (f_{os})	A,107	A,267	A,306	A,427	A,466
6 (f_{os})	A,81	A,161	A,241	A,321	A,401

Tabelle 6.3.: Ergebnisse der Verifikation

Es wurde anhand eines Beispiels gezeigt, wie Mixed-Signal Assertions durch die Event-basierte Methode geprüft werden können. Der manuelle Aufwand der Tests kann reduziert werden, indem die notierten Eigenschaften bei Bedarf mitsimuliert werden und somit eine automatisierte Bewertung der Eigenschaften realisieren. Gleichzeitig dienen die Eigenschaften als Dokumentation und könnten bei Bedarf sogar für eine formale Eigenschaftsprüfung weiterverwendet werden.

6.6. Zusammenfassung Assertion-basierte Verifikation

In diesem Kapitel wurden zwei Verfahren entwickelt, die zur Assertion-basierten Prüfung von Discrete-Event Modellen verwendet werden können. Der Ansatz der algorithmischen Prüfung verwendet direkt die Definition der Semantik von XFLTL. Daraus ergeben sich Einschränkungen hinsichtlich der prüfbaren Eigenschaften. Inwiefern durch eine Normalisierung der Eigenschaften diese Einschränkungen umgangen werden können, wurde nicht untersucht, da sich daraus grundlegende Änderungen für die Realisierung ergeben. Dieser Ansatz hat den Vorteil, dass der Ablauf und die Realisierung der Eigenschaftsprüfung, durch die direkte dynamische Verwendung der Fixpunktdefinition der temporalen Operatoren, sehr einfach sind.

Der zweite Ansatz zielt auf eine direkte Überführung der Eigenschaften in eine symbolische Darstellung ab. Dabei werden die zu prüfenden Eigenschaften zuvor normalisiert, um eine einfachere Auswertung zu ermöglichen. Die symbolische Beschreibung hat den Vorteil, dass die Eigenschaftsprüfung statisch beschrieben ist. Durch die Beschreibung als Automat können mögliche Zustände der Eigenschaftserfüllung ebenfalls statisch analysiert werden.

Beide Ansätze basieren auf der Verwendung synchronisierter Ereignisse, können also Relationen zwischen gleichzeitig auftretenden Ereignissen nicht unterscheiden. Folgende Erweiterungen und Fragestellungen sind derzeit noch ungelöst:

- **Realisierung symbolischer Ansatz:** Offen ist derzeit noch die prototypische Realisierung des symbolischen Ansatzes, sowie Vergleiche mit dem algorithmischen Ansatz.
- **Normalisierung und Konstruktion:** Freiheitsgrade bei der Vorverarbeitung und Normalisierung der Eigenschaften wurden noch nicht systematisch untersucht. Denkbar ist, dass auch andere Normalisierungen definiert werden können, die z.B. eine schnelle Falsifikation ermöglichen.
- **Automatische Modellverknüpfung:** Dem realisierten Konzept nach, erfolgt die Prüfung extern durch den Assertion Checker, getriggert durch die Simulationsumgebung. Die Verknüpfung zwischen Simulationsumgebung und dem Assertion Checker erfolgt derzeit manuell, sollte für den praktischen Einsatz jedoch automatisiert werden.
- **Modellgenerierung:** Für die Verifikation analoger (zustandsbasierter) Signale ist eine Überführung in ereignisbasierte Signale erforderlich. Dazu ist die Nutzung zusätzlicher Modellelemente erforderlich. Untersucht werden sollte daher, inwiefern eine automatische Generierung der dazu erforderlichen Teilmodelle möglich ist.
- **Automatische Eigenschaftsgenerierung:** Es wurde ein Konzept für die Beschreibung hybrider Eigenschaften mittels eventbasierter Spezifikationen entwickelt und demonstriert. Für den praktischen Einsatz sollten Methoden für die dynamische Transformation zwischen den Beschreibungsansätzen entwickelt werden.
- **Visualisierung und Fehlersuche:** Die Visualisierung und Auswertung der Eigenschaften erfolgt bisher erst rudimentär. Für die praktische Verwendung müssen Ansätze zur Darstellung der Zustände der Eigenschaften weiterentwickelt und umgesetzt werden. Dies ermöglicht eine systematische Fehlersuche bei nicht erfüllten Eigenschaften.

7. Vergleich und Bewertung der Verifikationsansätze

Im Rahmen dieser Arbeit wurden zwei verschiedene Ansätze zur Spezifikation und Analyse temporaler Eigenschaften in Discrete-Event-Modellen bzw. in hybriden Modellen entwickelt und untersucht. In diesem Kapitel werden diese Ansätze vergleichend gegenübergestellt und bewertet.

In Kapitel 4 wurde dazu definiert, wie prozessbasierte bzw. zustandsbasierte Eigenschaften verwendet werden können. Beiden Ansätzen ist gemein, dass qualitative und quantitative temporale Beziehungen zwischen eventbasierten Signalen beschrieben werden. Beide Ansätze nutzen die Idee des Model Checking, es wird also jeweils die Äquivalenz bzw. die Teilmengenbeziehung zweier Modelle auf der Basis der funktionalen Eigenschaften verglichen.

Beim entwickelte Transformation-basierte Ansatz, siehe Kapitel 5, wird dazu aus dem zu analysierenden System und entsprechenden Annotationen ein zur Analyse verwendetes Design- und Spezifikationsmodell abgeleitet, die jeweils eine Abstraktion des Systems entsprechend der unterstützten Semantik der DE-Domäne darstellen. Die resultierenden Modelle, die den symbolischen Formalismus der *Uppaal Timed Automata* nutzen, müssen zur Verifikation in geeigneter Weise verknüpft werden. Die Verknüpfung der Modelle, ergibt sich aus der hierarchischen Struktur des zu analysierenden Systems. Die Verifikation selbst erfolgt dann durch Model Checking des kombinierten Modells bezüglich lokalen Erreichbarkeitseigenschaften in den Spezifikationsmodellen, siehe Abschnitt 5.3.5.

Der entwickelte Assertion-basierte Ansatz, siehe Kapitel 6, nutzt Spezifikation aus eventbasierten temporallogischen Eigenschaften. Das durch die Spezifikationen beschriebene Eigenschaftsmodell wird entweder algorithmisch realisiert, siehe Abschnitt 6.2, oder symbolisch realisiert, siehe Abschnitt 6.3. Die Verifikation erfolgt dann durch Cosimulation der realisierten Eigenschaftsmodelle, getrieben von Signalen, die während der Modellsimulation durch die Entwicklungsumgebung erzeugt werden. Durch den Zustand eines Eigenschaftsmodells wird der Zustand der zu prüfenden Eigenschaft beschrieben.

7.1. Bewertungskriterien

Aus den unterschiedlichen Ansätzen resultieren unterschiedliche Eigenschaften, die zur Bewertung der Eignung der jeweiligen Methode für die Verifikation von ausführbaren Modellen herangezogen werden. Dazu werden die nachfolgenden Kategorien für die Bewertung aufgestellt und detaillierte Fragestellungen abgeleitet.

Ausgangspunkt für die Bewertung ist die Zweckmäßigkeit der Methode für den Einsatz zur Verifikation. Das bedeutet, dass, gemessen an dem geforderten Umfang bzw. Einsatzzweck, eine Bewertung der Leistungsfähigkeit erfolgen muss. Aus der allgemeinen Struktur des Prozesses der Verifikation - Modell, Eigenschaft, Analysetechnik - lassen sich weitere Kategorien als Gegenstand der Bewertung einer Verifikationsmethode aufstellen. Dazu werden jeweils das unterstützte Ausgangsmodell, das unterstützte Eigenschaftsmodell, sowie die Eigenschaften der Verifikationstechnik separat betrachtet.

7.1.1. Analysemodell

Für das unterstützte Ausgangsmodell, als Analysegegenstand der Verifikation, bestehen für jede Verifikationsmethode Einschränkungen und Voraussetzungen für die Anwendbarkeit. Für die Betrachtungen im Rahmen dieser Arbeit wurden in Kapitel 3 folgende prinzipiellen Annahmen bzw. Einschränkungen getroffen. Ausgangspunkt der Betrachtungen sind ausführbare Modelle, die in ihrer Anwendungsdomäne nicht beschränkt sind. Rein mathematische Modelle (Petri-Netze, Automaten, usw.), siehe Abschnitt 2.4, sind kompliziert und besitzen Einschränkungen in der Ausdrucksmächtigkeit, so dass diese (als alleiniges Beschreibungsmittel) für ausführbare Spezifikation nur bedingt geeignet sind.

Eine Charakterisierung der Ausdrucksmächtigkeit von Modellen ist anhand der zugrunde liegenden Berechnungsdomäne möglich, da diese die Einsatzmöglichkeiten für bestimmte Anwendungsdomänen definiert. Da die DE-Domäne zur Beschreibung quantitativ zeitlicher Modelle eine besondere Rolle spielt, siehe Abschnitt 2.3, werden Einschränkungen bezüglich dieser Domäne gesondert betrachtet. Für die Anwendung einer Verifikationsmethode müssen weiterhin Einschränkungen bezüglich unterstützter Daten untersucht werden.

Weiterhin ergeben sich aus der Vollständigkeit eines Modells Voraussetzungen bezüglich der Anwendbarkeit einer Methode. Vorteilhaft ist, wenn eine Verifikationsmethode die hierarchische Struktur des Modells heranzieht, um eine bessere Skalierbarkeit zu erreichen. Neben der betrachteten Realisierung sollten auch die prinzipiellen Erweiterungsmöglichkeiten bezüglich anderer Modellierungsdomänen berücksichtigt werden.

Unter den getroffenen Annahmen werden daher folgende Fragestellungen zur Bewertung des unterstützten Analysemodells abgeleitet:

- Welche Berechnungsdomänen werden unterstützt?

- Welche Einschränkungen gelten bei Verwendung der DE-Berechnungsdomäne?
- Welche Einschränkungen gelten bezüglich der Datenmodellierung?
- Welche Voraussetzungen gelten für die Anwendbarkeit der Methode bezüglich Vollständigkeit?
- Wird die hierarchische Struktur des Modells zur Verifikation genutzt?
- Lässt sich die Methode erweitern, so dass zusätzliche Ausgangsmodelle unterstützt werden?

7.1.2. Spezifikationssprache

Wesentlich für die Verifikation sind die spezifizierten Eigenschaften, die verifiziert werden können. Dazu wurden in Kapitel 4 Einschränkungen definiert, die für eine Eigenschaftsspezifikation unter dem Aspekt der abstrakten Beschreibung funktionaler Modelle sinnvoll bzw. notwendig sind.

Als wesentliches Kriterium ist die Frage nach der Entscheidbarkeit über temporale bzw. kausale Eigenschaften heranzuziehen. Als temporale Eigenschaften werden hierbei temporallogisch spezifizierbare Eigenschaften, also qualitativ und quantitativ zeitliche Zusammenhänge betrachtet. Kausale Eigenschaften sind direkte oder indirekte strukturelle Zusammenhänge oder Abhängigkeiten im Modell, die zu einer Ursache-Wirkung Beziehung führen. Die Ausdrucksmächtigkeit einer Spezifikation ist durch die verwendbaren Basiskonstrukte beschränkt. Ein Basiskonstrukt, das eine Methode im Analysemodell unterstützt, muss nicht automatisch als Teil einer zu analysierenden Eigenschaft zur Verfügung stehen. Von besonderer Bedeutung sind hierbei das Event, der Zustand eines Modells sowie Prozesse innerhalb eines Modells.

Für die Nutzung einer Spezifikation muss in geeigneter Weise eine Beziehung zum Modell hergestellt werden. Dies kann sich nach den Erfordernissen der Methode unterschieden, aber auch durch anwendungsspezifische und realisierungsspezifische Erwägungen bedingt sein. Die Wiederverwendbarkeit von Spezifikationen stellt ein weiteres Bewertungskriterium dar.

Für die Bewertung der unterstützten Eigenschaften werden folgende Fragestellungen aufgestellt:

- Welche Klassen von Eigenschaften können spezifiziert werden?
  - Werden temporale/ kausale Eigenschaften unterstützt?
  - Werden Aussagen über Events/ Zustände/ Prozesse unterstützt?
- In welcher Form werden Eigenschaften spezifiziert?
- Wie zweckmäßig ist die Spezifikation für den praktischen Einsatz?
- Sind die Spezifikationen wieder verwendbar?

7.1.3. Verifikationstechnik

Als weitere Kategorie zur Bewertung lässt sich die Verifikationstechnik selbst heranziehen. Prinzipiell stellt sich die Frage nach dem Charakter der Methode als vollständige oder unvollständige Aussage über die jeweils betrachteten Analyse- und Spezifikationsmodelle.

Um einen weitergehenden Nutzen für die Entwicklung eines Modells zu ermöglichen, sollten geeignete Auswertemöglichkeiten zur Verfügung stehen, um die Ursache auftretender Fehler im Detail ermitteln zu können.

Aus der Methode selbst ergeben sich Einschränkungen in der Komplexität der analysierbaren Modelle und Eigenschaften. Weiterhin ist der Aufwand für die Prüfung selbst ein Kriterium für die Anwendbarkeit einer Methode. Die Anwenderfreundlichkeit einer Methode ist wichtig für deren Akzeptanz.

Zur Bewertung werden folgende Fragestellungen aufgestellt:

- Ist die Verifikation vollständig?
- Welche Möglichkeiten der Auswertung und Detailanalyse stehen durch die Verifikationsmethode zur Verfügung?
- Welche Einschränkungen bestehen bezüglich der Komplexität des Modells?
- Wie aufwändig ist der Prüfvorgang selbst aus algorithmischer Sicht?
- Wie anwenderfreundlich ist die Prüfung, wie gut lässt sich die Technik nutzen?

7.2. Vergleich der Verifikationsansätze

Die untersuchten Verifikationsmethoden wurden entsprechend den aufgestellten Bewertungskriterien untersucht und bewertet. Tabelle 7.1 zeigt die Ergebnisse unterteilt in die Kategorien Analysemodell, Spezifikationsmodell und Verifikationstechnik. Als Methoden wurden die drei untersuchten Verifikationsmethoden, Transformation-basierte Verifikation (Kapitel 5), Assertion-basierte algorithmische Verifikation (Abschnitt 6.2) und Assertion-basierte symbolische Verifikation (Abschnitt 6.3) herangezogen. Zudem wurde betrachtet, welches Potential die entwickelten Methoden bei einer ausgereiften Umsetzung, als erweiterte Transformation-basierte Methode bzw. als erweiterte Assertion-basierte Methode, entwickeln könnten. Somit ergeben sich insgesamt fünf Methoden, die gegenübergestellt werden.

In den nachfolgenden Abschnitten werden die Ergebnisse der Tabelle 7.1 näher erläutert und bewertet.

Kriterium	Transformation-basiert	*Transformation-basiert erweitert*	Assertion-basiert algorithmisch	Assertion-basiert symbolisch	*Assertion-basiert erweitert*
Analysemodell					
Berechnungsdomänen	DE, FSM	DE, FSM, SDF	DE, FSM, SDF, CTDE	DE, FSM, SDF, CTDE	alle, DE-kompatibel
Einschränkungen DE-Domäne	Memory, Events,	-	-	-	-
Einschränkungen Daten	Eventinstanzen	Aufzähltypen	Schnittstelle Skalar	Schnittstelle Skalar	Schnittstelle Skalar/Struktur
Vollständiges Modell nötig	ja	ja	nein	nein	nein
Hierarchie genutzt?	nein	ja	nein	nein	nein
Erweiterbar	ja / Datenfluss	ja / Datenfluss	ja / *	ja / *	ja / *
Spezifikationsmodell					
Eigenschaften:	ja / nein	ja / ja	ja / nein	ja / nein	ja /nein
Temporal/Kausal					
Basiskonstrukte:	ja / nein / ja	ja / ja / ja	ja / ja / nein	ja / ja / nein	ja / ja / ja
Events/Zustände/Prozesse					
Notation im Modell	Annotation lokal/global	Annotation lokal/global	Modellelement	Modellelement	Annotation lokal/global
Wiederverwendbarkeit	ja	ja	nein	nein	ja
Zweckmäßigkeit/Bewertung	+	++	o	+	++
Verifikationstechnik					
Vollständigkeit	ja	ja	nein	nein	nein
Auswertemöglichkeit (Trace)	abstrakt	abstrakt / Modell	Zeitdiagramm	Zeitdiagramm	Zeitdiagramm / Signalverlauf
Modellkomplexität	gering	gering-mittel	komplex	komplex	komplex
Algorithmenkomplexität	hoch	hoch	mittel	mittel-gering	gering
Usability	gering	hoch	gering	mittel	hoch

Tabelle 7.1.: Bewertung und Gegenüberstellung der Verifikationsmethoden

7.2.1. Bewertung Analysemodell

Berechnungsdomänen

Als Berechnungsdomänen werden von allen Methoden die DE-Domäne und die FSM-Domäne unterstützt. Da für die Transformation-basierte Verifikation zusätzlich die vollständige Interpretation des zu analysierenden Modells erforderlich ist, gelten hier weiterführende Einschränkungen. Als Erweiterung des Transformation-basierten Ansatzes ist denkbar, dass zusätzlich die SDF-Domäne unterstützt wird. Für die Assertion-basierten Ansätze ist zusätzlich auch die Verwendung der SDF- und CTDE-Domäne möglich, sofern die betrachteten Signale in der DE-Domäne interpretiert werden können. Prinzipiell sind als Erweiterung sämtliche zur DE-Domäne kompatiblen[70] Modelle denkbar.

Einschränkungen DE-Domäne

Einschränkungen bei Verwendung von Modellen der DE-Domäne bestehen lediglich beim Transformation-basierten Ansatz. Es wurde zwar betrachtet, wie sich Memories und Events methodisch integrieren lassen, dies wurde jedoch nicht vollständig realisiert, ist aber als Erweiterung denkbar.

Einschränkungen Daten

Einschränkungen können bei den im Modell verwendeten Datentypen bestehen. insbesondere beim Transformation-basierten Ansatz müssen alle Daten und Operationen abgebildet werden. Derzeit werden Modelle basierend auf Signalinstanzen unterstützt, da die Modellierung komplexer Operationen im Rahmen temporaler Eigenschaften, auch in Bezug auf die genutzten Zielmodelle der Transformation sehr aufwändig ist. Es ist jedoch absehbar, dass eine Erweiterung zumindest für abzählbare Datentypen möglich ist. Bei den Assertion-basierten Ansätzen bestehen Einschränkungen nicht im gesamten Modell, sondern nur an den Stellen, die für die Verifikation beobachtet werden sollen. Derzeit werden hierbei skalare Datentypen unterstützt, die sich auf Real-Datentypen bzw. Integer-Datentypen abbilden lassen. Eine Erweiterung für strukturierte Datentypen ist hierbei denkbar.

Vollständiges Modell

Für die Transformation-basierten Ansätze ist ein vollständiges Modell zwingend erforderlich, da ansonsten keine vollständige funktionale Analyse durchgeführt werden kann. Die Assertion-basierten Ansätze fordern dies nicht, allerdings kann nur über das simulierte Verhalten eine

[70] Eine Domäne wird hier als kompatibel betrachtet, wenn sie das Zeit- und Signalmodell der DE-Domäne erbt (z.B. CTDE), oder in die DE-Domäne eingebettet werden kann (z.B. SDF).

Aussage getroffen werden, so dass sich der Anspruch an die Vollständigkeit an die Anforderungen des Simulators richtet.

Nutzung von Hierarchie

Eine Nutzung der Hierarchie ist beim Transformation-basierten Ansatz prinzipiell angedacht, aber nicht vollständig umgesetzt, dies wäre als Erweiterung denkbar. Die Assertion-basierten Ansätze nutzen nicht die vollständige Struktur des Modells, so dass durch die Betrachtung der Modellhierarchie keine Vorteile entstehen.

7.2.2. Bewertung Spezifikationssprache

Temporale und kausale Beziehungen

Techniken zur Verifikation temporaler Eigenschaften stellen die Grundlage der betrachteten Verifikationsmethoden dar. Im Transformation-basierten Ansatz werden Prozesse als Basis der Spezifikation verwendet, deren temporales Verhalten für die Verifikation genutzt wird. Dadurch besteht prinzipiell die Möglichkeit, neben temporalen Eigenschaften auch kausale Zusammenhänge im Analysemodell zu berücksichtigen, allerdings wird dies durch das Analysemodell nicht unterstützt. Durch eine erweiterte Analysemethode, siehe Anhang E, würden solche Zusammenhänge analysierbar und könnten in einem erweiterten Transformation-basierten Ansatz zur Verfügung stehen. Im Assertion-basierten Ansatz werden temporale Eigenschaften direkt spezifiziert. Wegen dem Fehlen struktureller Informationen über das Analysemodell sind kausale Zusammenhänge nicht analysierbar.

Basiskonstrukte Events, Zustände, Prozesse

Als Basiskonstrukte der Spezifikationssprachen wurden in Kapitel 4 Zustände und Prozesse betrachtet. Events stellen auf Modellebene den Zusammenhang zwischen Modellen dar. Events werden daher als Ausgangspunkt der Argumentation verwendet und stellen somit einen zentralen Bestandteil der Spezifikationsmodelle der betrachteten Verifikationsmethoden dar. Zustände in Form von Memories sind im Transformation-basierten Ansatz nicht allgemein Gegenstand der Spezifikation, auch wenn diese bei Zustandsmaschinen unterstützt werden. Als Erweiterung des Ansatzes wäre die Berücksichtigung von Memories jedoch möglich. Im Assertion-basierten Ansatz sind Zustände in Form von Signalzuständen spezifizierbar und lassen sich somit für die Analyse von Zuständen von Memories nutzen. Prozesse stehen im Assertion-basierten Ansatz als Realisierung von Funktionalität zur Verfügung. Diese werden zwar nicht direkt adressiert, sind jedoch in anonymer Form Gegenstand der Spezifikation. Im Assertion-basierten Ansatz werden Prozesse nicht verwendet. Allerdings ist denkbar, dass das Spezifikationsmodell um

eine abstrakte Sichtweise erweitert wird, so dass die temporalen Eigenschaften über Objekte spezifiziert werden, die den Charakter von Prozessen besitzen.

Notation im Modell

Die Koordination mit dem Modell beschreibt, wie die Spezifikation der Eigenschaften im Modell erfolgt. Dadurch werden u.a. der Aufwand und die Wiederverwendbarkeit bestimmt. Der Transformation-basierte Ansatz nutzt die Annotation im Modell, das heißt, dass die Eigenschaften an den betreffenden Modellelementen oder aber global im Modell definiert werden. Vorteil ist, dass die Zuordnung explizit erfolgt und über die Modellbibliothek lokale Annotationen prinzipiell wiederverwendbar sind. Im Assertion-basierten Ansatz ist derzeit für die Spezifikation ein zusätzliches Modellelement erforderlich. Als Erweiterung ist jedoch denkbar, dass spezifizierte Eigenschaften in Form von Annotationen notiert werden. Für die Verifikation selbst könnten erforderliche Modellelemente automatisch generiert werden, oder es wird ein anderer Mechanismus zur Kopplung mit der Simulation genutzt.

Wiederverwendbarkeit

Als Wiederverwendbarkeit der Spezifikation wird die automatische Nutzung einer Eigenschaft in einem neuen Modell betrachtet. Annotationen sind prinzipiell wiederverwendbar, sofern sie lokal im Kontext eines Modellelements angewendet und in die Modellbibliothek übernommen werden. Bei Verwendung von Modellelementen ist eine Wiederverwendung durch die Module zwar denkbar, aber nur bedingt zweckmäßig. Somit ist die Wiederverwendbarkeit der Spezifikationen beim Transformation-basierten Ansatz gegeben, beim Assertion-basierten Ansatz wäre dies als Erweiterung denkbar.

Zweckmäßigkeit und Bewertung

Dieses Kriterium dient der Bewertung des Spezifikationsmodells und zielt auf die Zweckmäßigkeit im Kontext der betrachteten Methode insgesamt ab. Das Spezifikationsmodell des entwickelten Transformation-basierten Ansatzes ist technisch als zweckmäßig zu bewerten, da durch die Annotation eine Verifikation des funktionalen und zeitlichen Verhaltens ermöglicht wird. Durch eine automatisierte Interpretation der Primitive als abstrakter Automat könnte der Aufwand zur Annotation reduziert werden. Durch bessere Werkzeugunterstützung bei der Spezifikation und Erweiterung der Eigenschaften könnte die Anwenderfreundlichkeit gesteigert werden. Die Spezifikation der Assertion-basierten Methode ist zwar beim algorithmischen und symbolischen Ansatz im Wesentlichen identisch, allerdings gelten bei dem Assertion-basierten Ansatz Einschränkungen für die Prüfbarkeit von Eigenschaften. Eine Abwertung muss für die Art der Notation im Modell vorgenommen werden, da diese zu einem erhöhten Aufwand und Problemen bei der Wiederverwendbarkeit führt.

7.2.3. Bewertung der Verifikationstechnik

Vollständigkeit der Analyse

Ein wichtiges Kriterium für die Bewertung der Verifikationstechnik ist die Frage nach der Vollständigkeit der Analyse. Diese ist bei dem Transformation-basierten Ansatz gegeben, beim Assertion-basierten Ansatz nicht.

Analysemöglichkeiten

Als Analysemöglichkeit wird die Art und Weise betrachtet, mit der eine Analyse der Ursache nicht erfüllter Eigenschaften erfolgen kann, bzw. inwiefern Hinweise und Hilfestellungen zur Ursachenermittlung geliefert werden können. Allgemein wird dazu ein Trace verwendet, also eine zeitliche Darstellung eines Ablaufes, der zur Nichterfüllung der Eigenschaft führt. Beim Transformation-basierten Ansatz ist die Analyse durch ein Trace im abstrakten Modell, durch das verwendete Verifikationswerkzeug möglich. Als Erweiterung wäre eine Projektion des Fehlertrace in das Simulationsmodell denkbar. Allerdings dürfte sich dieser Schritt als aufwändig erweisen, da die Steuerung des Simulators nur eingeschränkt unterstützt wird. Bei den Assertion-basierten Ansätzen kann prinzipiell der Zeitverlauf der Erfüllung der Eigenschaften zur Analyse einer Fehlerursache genutzt werden. Realisiert wurde dies in Form von Logging-Informationen. Je nach Methode bieten sich verschiedene Möglichkeiten der Darstellung an. Die Formulierung der temporalen Eigenschaften als Formel lässt sich als Baum darstellen, zu dessen Zweigen ein Zeitverlauf des Zustandes des betreffenden Operators vorgenommen werden kann. Die symbolische Darstellung ermöglicht zusätzlich die Darstellung des Zustandsverlaufes des zur Verifikation verwendeten Timed-AR Automaten. Denkbar ist auch, dass für die Darstellung eine andere, funktional äquivalente Automatenstruktur verwendet wird, die eine leichtere Auswertung ermöglicht. Die grafische Auswertung steht für die Analyse des Zeitverlaufes der Formeln noch nicht zur Verfügung, könnte aber als Erweiterung realisiert werden. Zudem wäre es denkbar, dass alternative Zeitverläufe ermittelt werden, für die, ausgehend von einem Zwischenzustand, eine Erfüllung/ Nichterfüllung der Eigenschaft möglich ist.

Modellkomplexität

Eine Bewertung der analysierbaren Komplexität des Analysemodells ist nicht trivial, da kein geeignetes Maß für die Modellkomplexität zur Verfügung steht. In der Regel werden Vergleiche daher anhand von Benchmarks durchgeführt, wobei für die Bewertung der Komplexität des konkreten Verifikationsproblems Speicherplatzbedarf und Laufzeit herangezogen werden. Problematisch ist dabei, dass sich durch bloße Änderungen von Parametern beide Maße beeinflussen lassen, ohne an der Struktur oder Größe des Modells Änderungen vorzunehmen.

Daher werden hier Abschätzungen vorgenommen, die als qualitative Bewertung verwendet

werden können. Der Transformation-basierte Ansatz ist für Modelle mit einer vergleichsweise geringen Komplexität anwendbar, da eine vollständige Analyse vorgenommen wird. Zwar werden bereits Optimierungen beim generierten Modell vorgenommen, diese ermöglichen jedoch keine Verringerung der Komplexität, sondern lediglich der Größe. Durch die Einbeziehung kausaler Erwägungen bezüglich der Modellstruktur, sowie der Optimierung des generierten Modells bezüglich der zu prüfenden Eigenschaften bzw. der Analysetechnik, ließe sich die Komplexität der verifizierbaren Modelle voraussichtlich erhöhen. Bei den Assertion-basierten Techniken ist die Modellkomplexität unerheblich, da lediglich die Signale betrachtet werden.

Algorithmenkomplexität

Das Kriterium Algorithmenkomplexität beschreibt die Laufzeit einer Verifikationsmethode relativ zur geprüften Eigenschaft. Hier wäre ein direkter Vergleich nur zwischen den Assertion-basierten Ansätzen möglich. Deswegen erfolgt hier nur eine qualitative Bewertung der betrachteten Varianten. Bei Ansätzen zur vollständigen Verifikation ist die Algorithmenkomplexität allgemein NP-vollständig. Für die Transformation-basierten Ansätze wird daher eine hohe Komplexität angenommen, auch wenn prinzipielle Verbesserungen durch weitere Optimierung der Modellstruktur sowie durch funktionale Abstraktion möglich sind. Bei den Assertion-basierten Ansätzen hängt die Komplexität allgemein von der zu prüfenden Eigenschaft sowie von den Signalverläufen ab. Die Komplexität der algorithmischen Realisierung ist als "mittel" zu bewerten, da nur jeweils die Operatoren der Teilformeln geprüft werden, die zu einer Änderung der Eigenschaft beitragen. Allerdings besteht noch das Problem der zeitlich nicht koordinierten Bezugspunkte der Operatoren, so dass potentiell gleichzeitig mehrere Instanzen von Operatoren zur Prüfung der Eigenschaften herangezogen werden. Beim symbolischen Ansatz werden die Bezugspunkte durch eine Normalisierung so koordiniert, dass jeweils nur eine Instanz des Operators geprüft werden muss. Dazu ist zwar eine zeitlich begrenzte Speicherung der Signale erforderlich, dies ist jedoch vergleichsweise effizient möglich, so dass die Algorithmenkomplexität mit "mittel-gering" eingeschätzt wird. Es ist zu erwarten, dass sich die Komplexität durch Optimierung der Automaten noch weiter verringern lässt.

Usability

Als Usability wird die Anwendbarkeit der betreffenden Verifikationsmethode insgesamt bewertet. Der Transformation-basierte Ansatz wird im derzeitigen Stand als "gering" eingeschätzt. Dies ist vor allem mit den Beschränkungen der Analysemodells zu begründen. Ferner sind eine bessere Unterstützung bei der Spezifikation, sowie Verbesserungen bezüglich der Analysemöglichkeiten wünschenswert. Durch Erweiterungen des Spezifikationsmodells würde für den Ansatz eine hohe Usability erreicht werden. Beim algorithmischen Assertion-basierten Ansatz ist die Anwendbarkeit ebenfalls mit gering zu bewerten. Dies liegt u.a. an den Einschränkun-

gen, die bezüglich der Eigenschaften bestehen. Weiterhin sind, ebenso wie bei der symbolischen Realisierung, die Analysemöglichkeiten bislang noch nicht ausgeschöpft. Es besteht keine Unterstützung bei der Spezifikation und nur eine geringe Wiederverwendbarkeit. Daher wird der symbolische Ansatz mit "mittel" bewertet. Durch eine Unterstützung bei der Spezifikation und die Erweiterung der Analysemöglichkeiten ist voraussichtlich eine hohe Anwendbarkeit des Ansatzes erreichbar.

7.3. Zusammenfassung der Verifikationsansätze

Zusammenfassend lässt sich feststellen, dass sich die erarbeiteten Ansätze prinzipiell für die Verifikation temporaler Eigenschaften in komplexen ausführbaren Spezifikationen eignen. Gleichzeitig wird deutlich, dass für eine umfassende Anwendung der Methoden noch Bedarf an Forschungs- und Entwicklungsarbeit besteht.

8. Schlussbemerkungen und Ausblick

8.1. Zusammenfassung

Ausgangspunkt der Arbeit war die Fragestellung nach einer Methode zur Verifikation von Echtzeiteigenschaften in ausführbaren Spezifikationen. Dazu wurde analysiert, welchen Charakter ausführbare Spezifikationen besitzen, siehe Kapitel 1.

Zunächst wurden, ausgehend von der allgemeinen Problemstellung, existierender Ansätze zur Modellierung ausführbarer Spezifikationen sowie zur Verifikation von zeitlich-funktionalen Eigenschaften analysiert, siehe Kapitel 2. Die Frage nach konkreten Modellierungsansätzen ist dabei schwer zu beantworten, da der konkret verwendete Modellierungsansatz von vielen Faktoren abhängt und für bestimmte Modellierungsaspekte teilweise spezialisierte Modellierungsparadigmen genutzt werden. Aus den verschiedenen Möglichkeiten wurde herausgearbeitet, dass sich die Modelle ausführbarer Spezifikationen nach den zugrunde liegenden Berechnungsmodellen charakterisieren lassen, unabhängig von der konkreten Umsetzung einer Modellbeschreibung oder Simulationsumgebung. Für die Beschreibung diskreter Systeme mit Echtzeiteigenschaften werden vor allem Diskrete-Event Modelle, oft in Verbindung mit abstrakten Zustandsmodellen eingesetzt. Die zu betrachtenden Eigenschaften sind ebenso vielfältig, wie die verwendeten Modelle. Eine Eingrenzung auf eine bestimmte Klasse von Eigenschaften ist dabei nicht zweckmäßig, da, im Hinblick auf die Analyse, sofort Eingrenzungen des Beschreibungsmodells erforderlich werden. Da für solche Fälle ohnehin spezialisierte Analysewerkzeuge zur Verfügung stehen, erfolgte eine Einschränkung lediglich auf zeitlich-logische Eigenschaften in Form quantitativ temporallogischer Eigenschaftsmodelle.

Ein weiterer Analyseschwerpunkt bestand in der Untersuchung existierender Ansätze zur Verifikation bezüglich der Eignung zur Verifikation zeitlich-funktionaler Eigenschaften. Dabei zeigte sich, dass für die Klasse der betrachteten Discrete-Event Modelle keine exakt passende Verifikationsmethode gefunden werden konnte. Allerdings existieren Verifikationsansätze für ähnliche Modelle, die zur Verifikation zeitlich-funktionaler Eigenschaften geeignet sind.

Auf Basis der Analyseergebnisse und der getroffenen Einschränkungen erfolgte die Definition konkreter Problem- und Fragestellungen, siehe Kapitel 3. Dabei wurden Konzepte entwickelt, die eine Verifikation zeitlich-funktionaler Eigenschaften in ausführbaren Spezifikationen ermöglichen. Der Transformation-basierte Ansatz besteht in der Transformation des zu analysierenden Discrete-Event Modells in eine formale Darstellung derart, dass bereits existierende Verifika-

tionstechniken und -werkzeuge verwendet werden können und eine vollständige Verifikation realisiert werden kann. Der Assertion-basierte Ansatz soll eine simulationsbegleitende Analyse und Bewertung von Eigenschaften, sogenannte Assertions ermöglichen, indem Eigenschaften cosimuliert werden. Grundlegend ist für die Entwicklung einer Verifikationsmethode die Frage nach der Art und Spezifikation von Eigenschaften sowie die Frage nach der Umsetzung der Verifikation selbst zu beantworten.

Mit Hinblick auf die zu realisierenden Konzepte wurden geeignete Ansätze zur Spezifikation von zeitlich-funktionalen Eigenschaften entwickelt, siehe Kapitel 4. Dazu wurden Spezifika der zu analysierenden Modellklassen und Konzepte untersucht und entsprechend geeignete Ansätze zur Eigenschaftsspezifikation entwickelt. Es wurde eine prozessbasierte Constraint-Language für den Transformation-basierten Ansatz und eine XFLTL-Logik zur Beschreibung eventbasierter temporaler Eigenschaften für den Assertion-basierten Ansatz entwickelt.

Kapitel 5 beschreibt die Konzeption und Realisierung des Transformation-basierten Ansatzes, sowie die Evaluierung anhand eines Beispiels. Aus den Ergebnissen werden mögliche Erweiterungen des Ansatzes vorgeschlagen.

In Kapitel 6 werden zwei Realisierungsvarianten des Assertion-basierten Verifikationskonzepts entwickelt und anhand eines Beispiels evaluiert.

Die entwickelten Ansätze werden in Kapitel 7 vergleichend gegenübergestellt. Dabei werden Abschätzungen vorgenommen, wie sich die jeweils vorgeschlagenen Erweiterungen auf die Bewertung der Methode auswirken werden.

8.2. Diskussion der Ergebnisse

Als Ergebnis der Analyse wurde festgestellt, dass zwar in zunehmendem Maße ausführbare Spezifikationsmodelle bei der systematischen Systementwicklung eingesetzt werden, diese jedoch meist nicht direkt für eine formale Verifikation eingesetzt werden können. *Zwar existieren auch spezialisierte Modellierungssprachen, wie Z, B, oder SDL, die auch eine Unterstützung für formale Verifikationsmethoden bieten, deren Anwendung ist jedoch entweder auf bestimmte Entwicklungsdomänen beschränkt, oder setzt strenge Grenzen bezüglich der unterstützten Spezifikationsmodelle.*

Bezüglich des Einsatzbereiches der zu betrachtenden Modelle und den zu verifizierenden Eigenschaften wurden allgemeine Anforderungen, basierend auf dem zu unterstützenden Berechnungsmodell, formuliert, die an Verifikationsmethoden für zeitlich-funktionale Eigenschaften in ausführbaren Spezifikationsmodellen gestellt werden müssen. Diese beinhalten die Unterstützung kontinuierlicher Zeiten und asynchroner Ereignisse in Modellen, sowie die Unterstützung quantitativer temporallogischer Eigenschaften. *Es existieren zwar weitere Modellaspekte, allerdings erscheint eine Eingrenzung bezüglich des Berechnungsmodells wegen der Unabhängigkeit von einem konkreten Modellierungsmittel als besonders geeignet. Für die Eigenschaftsspezifi-*

kation existieren neben temporallogischer Eigenschaften zahlreiche weitere Ansätze. Allerdings lässt sich zeigen, dass sich eine Vielzahl der Eigenschaften ebenso in Form temporallogischer Spezifikationen beschreiben lässt. Zudem ist es möglich, temporallogische Eigenschaften auch implizit, in Form von Automaten, auszudrücken. Verfahren zur reinen Zeitanalyse werden als nicht ausreichend angesehen, da funktional-logische Aspekte nicht modelliert werden.

Als Ergebnis der Arbeit wurden zwei Ansätze entwickelt, die mit unterschiedlichen Zielstellungen unterschiedliche Konzepte verfolgen. Beiden Ansätzen ist gemein, dass die Verifikation zeitlich-funktionaler Eigenschaften in komplexen ausführbaren Spezifikationen der Discrete-Event Berechnungsdomäne erlaubt wird.

Der entwickelte Transformation-basierte Ansatz verfolgt das Ziel, eine vollständige formale Verifikation zu realisieren. Dazu wird das zu analysierende Modell sowie die zu überprüfenden Eigenschaften in eine formale Darstellung (Zielmodell) transformiert, in der die Verifikation technisch realisiert wird. Um möglichst wenige Einschränkungen bezüglich der Modellierungsweise des zu analysierenden Modells zu treffen, wurde dazu ein Verfahren zur Annotation von Eigenschaften als prozessbasierte Spezifikationen definiert. Aus der Art der Modellierung der zu unterstützenden Modelle, speziell von MLDesigner Modellen der DE-Berechnungsdomäne, ergeben sich für die zu verifizierenden Eigenschaften Einschränkungen bezüglich der adressierbaren Modellelemente. Aus verschiedenen Zielmodellen wurden die Modelle der *Uppaal Timed Automata* ausgewählt, da sie dem Modularitätskonzept und der Semantik der DE-Domäne des MLDesigner am besten gerecht werden. Da die Verifikation selbst einen Model Checking Algorithmus nutzt, lassen sich auf dieser Ebene als Basiselemente der Eigenschaften nur Zustände verwenden. Zur Abbildung zeitlicher Aspekte und zur Verwendung modellspezifischer Basiselemente werden daher Beobachterautomaten generiert. *Ein wesentlicher Nachteil des Verfahrens besteht in der notwendigen Annotation des zu analysierenden Modells. Der dazu realisierte Sprachumfang ist noch nicht für komplexe ausführbare Spezifikationen ausreichend, Erweiterungen sind aber prinzipiell möglich. Sinnvoll ist hier die Entwicklung einer automatisierten Erzeugung der Annotationen. Das Zielmodell der* Uppaal Timed Automata *ist prinzipiell für entwickelte Methode geeignet. Allerdings zeigen sich auch Nachteile, zum Beispiel besteht die Notwendigkeit der Generierung von Beobachterautomaten und es ist keine direkte Analyse struktureller Eigenschaften möglich. Gleichwohl lassen sich strukturell bedingte Anforderungen an die Reihenfolge auftretender Events durch die prozessbasierten Spezifikationen beschreiben und prüfen. Denkbar ist hier die Weiterentwicklung und Etablierung von Analysemethoden, die sowohl temporallogische als auch strukturelle Aspekte gemeinsam analysieren können. Eine Erweiterung um weitere Berechnungsdomänen ist nur bedingt möglich. Zwar lassen sich auch SDF-Modelle in temporallogische Modelle überführen, die Semantik zeitlicher Aspekte muss dann jedoch geeignet abgebildet werden.*

Die Untersuchungen zeigen, dass der Transformation-basierte Ansatz prinzipiell anwendbar ist, allerdings zeigen sich teilweise weitreichende Einschränkungen,

vor allem bedingt durch die Ausdrucksmächtigkeit bestehen. Sinnvoll erscheint eine Transformation auf hohem Abstraktionsniveau, indem eine geeignete formale Spezifikationssprache verwendet wird. Dabei ist es erstrebenswert, Teile des ausführbaren Modells aus der formalen Spezifikation zu generieren.

Der Assertion-basierte Ansatz verfolgt das Ziel, die Erfüllung von Eigenschaften dynamisch zur Laufzeit der Simulation zu entscheiden. Dazu wird für die Spezifikation eine temporale Logik definiert, bei der auftretende Ereignisse zeitlich synchronisiert betrachtet werden. Das bedeutet, dass für die Beschreibung der Eigenschaften selbst die Diskretisierung zeitlich gleichzeitig auftretender Ereignisse nicht zur Verfügung steht. Strukturelle Eigenschaften des Modells lassen sich demnach nicht beschreiben. Im Unterschied zu herkömmlichen, zustandsbasierten temporalen Logiken werden singuläre Ereignisse als Basis der Eigenschaftsbeschreibung genutzt. *Dadurch ergibt sich eine leicht abgeänderte Semantik und es besteht prinzipiell die Möglichkeit, Eigenschaften verschiedener Berechnungsdomänen zu beschreiben. Allerdings sind dazu umfangreiche Forschungsarbeiten erforderlich, die eine systematische Verwendung innerhalb einer Multi-Domain Umgebung ermöglichen. Nachteilig ist, dass die Spezifikation der Eigenschaften selbst Spezialwissen über die Zusammenhänge und Arbeitsweise der temporalen Logik erfordert. Hier ist es sinnvoll, dem Entwickler durch Makros oder andere Hilfsmittel die Spezifikation von Eigenschaften zu erleichtern.*

Die Anwendbarkeit des Assertion-basierten Ansatzes wurde demonstriert. Insbesondere besteht hier die Möglichkeit, Zusicherungen und Annahmen eventbasiert zur Laufzeit zu prüfen. Sinnvoll erscheint die Weiterverfolgung des Ansatzes in zwei Richtungen. Die Verwendung der Assertions muss technisch und organisatorisch stärker in den Entwicklungsprozess integriert werden. Zusätzlich zur eventbasierten Prüfung sollte eine abstraktere, möglicherweise heterogene Spezifikation der Eigenschaften möglich sein.

Die gewählten Beispiele zeigen, dass die entwickelten Methoden aus Sicht der Verifikation praktisch anwendbar sind. Zu beantworten ist die Frage, ob eine Verifikation in ausführbaren Spezifikationen im Kontext eines Entwicklungsprozesses sinnvoll ist, und wenn ja, ob sich die entwickelten Methoden dafür eignen.

Die Frage nach der Aussagekraft von Ergebnissen, die aus der Verifikation von Spezifikationsmodellen resultieren, lässt sich nicht eindeutig beantworten. Natürlich können nur diejenigen Eigenschaften verifiziert werden, die durch das erstellte Modell hinreichend genau modelliert werden. Umgekehrt sind für die Bewertung der Ergebnisse die Annahmen und Ungenauigkeiten des Modells zu berücksichtigen. Auf jeden Fall ist es erstrebenswert, kritische Aspekte/Eigenschaften möglichst früh im Entwicklungsprozess zu berücksichtigen und zu verifizieren. Zwei wesentliche Aspekte sprechen für die Verwendung ausführbarer Spezifikationen zur Verifikation. Es lässt sich einfacher sicherstellen, dass die bei der Entwicklung verwendeten Artefakte konsistent und widerspruchsfrei sind. Es gibt jedoch auch Gründe die gegen die Verwendung solcher

Methoden sprechen. Der Aufwand zur Verwendung solcher Methoden ist noch relativ hoch bzw. die Anwendbarkeit bezüglich der Modellkomplexität beschränkt, so dass abgewägt werden muss, ob ggf. für eine Verifikation die Verwendung spezialisierter Modelle geeigneter ist. Zudem muss berücksichtigt werden, ob für die zu untersuchenden Eigenschaften im Einzelfall spezialisierte Verifikationsmethoden zur Verfügung stehen.

Die Ergebnisse lassen sich auf andere Beschreibungssprachen bzw. Modellierungsansätze übertragen. *Beispielsweise benutzt SystemC ein Konzept, das in den wesentlichen Punkten äquivalent zu MLDesigner ist. Es werden spezielle C++-Klassenbibliotheken verwendet, die einen syntaktischen und semantischen Rahmen für die erstellten Modelle bieten. Es werden unterschiedliche Berechnungsdomänen unterstützt. Eine hierarchische Strukturierung ist durch Komposition mittels Signalen möglich.*

8.3. Ausblick

Im Verlauf der Arbeit wurden bereits zahlreiche Andeutungen gemacht, inwiefern Erweiterungen möglich oder notwendig sind, siehe vor allem Abschnitt 5.7 und Abschnitt 6.6. In Kapitel 7 wurden die Erweiterungsmöglichkeiten diskutiert und deren Auswirkungen abgeschätzt. Wesentlich sind hierbei:

- Erweiterung der Ausdrucksmächtigkeit der Eigenschaften bzw. Modelle,
- Beseitigung bestehender Einschränkungen,
- Unterstützung für die Spezifikation,
- Verfeinerung von Verifikationsalgorithmen,
- Erweiterte Evaluierung durch Benchmarks.

Aus den Ergebnissen der Arbeit können jedoch noch weiterführende Forschungsfragen abgeleitet werden. Für die Verwendung ausführbarer Spezifikationen fehlt derzeit ein Abstraktionskonzept, bei dem die Nutzung verschiedener Abstraktionsebenen explizit beschrieben wird. Dies betrifft einerseits das Management von Metainformationen, über die der Modellierungszweck, also Aussagekraft und Genauigkeit bestimmter Systemaspekte, explizit beschrieben wird. Zudem ist es wünschenswert, Teilmodelle verschiedener Abstraktionsebenen koppeln zu können, was die Definition geeigneter Schnittstellen erfordert. Die Verfügbarkeit eines solchen Abstraktionskonzeptes würde es erlauben zu prüfen, ob ein Modell geeignet ist, Aussagen über bestimmte Modellaspekte zu treffen. Gegebenenfalls wäre es möglich, Modelle zu erstellen, die, bezüglich der Modellkomplexität, minimal den Anforderungen eines konkreten Verifikationsproblems genügen.

Für Analysen im Regionengraph fehlen derzeit leistungsfähige Algorithmen, die direkt eine Aussage über zeitbeschränkte temporallogische Aussagen erlauben. In diesem Zusammenhang sollte untersucht werden, inwiefern eine gemeinsame Analyse kausaler Zusammenhänge und

zyklischer Pfade möglich ist, siehe dazu auch Anhang E. Möglicherweise ist hierbei eine explizite Modellierung von Ressourcen erforderlich, um komplexe kausale Abhängigkeiten entscheiden zu können.

Die entwickelten Verifikationsmethoden argumentieren alleinig über Pfade im Modell. Alternativ sollten Ansätze untersucht werden, welche eine mengenbezogene Spezifikation und Bewertung erlauben, also über Teilmengen von Pfaden mit bestimmten Eigenschaften argumentieren, wie etwa ASL [SH08]. Eine solche Methode ließe sich voraussichtlich für Analysen im Regionengraph adaptieren.

Die eventbasierte Verifikation von Assertions besitzt, wie demonstriert wurde, Potential zur Verifikation temporaler Eigenschaften in hybriden, also gemischt analog-digitalen Modellen. Notwendig ist dabei eine Anpassung der zu analysierenden Signale und Eigenschaften an die Semantik der ereignisbasierten temporalen Logik. Erstrebenswert ist dabei die systematische Erforschung von Fragestellungen, die eine Automatisierung des Ansatzes ermöglichen.

A. Eigenschaftsbeschreibungssprachen

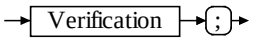
A.1. Constraint Language des Transformation-basierten Ansatzes

Die Syntax der Constraint Language ist als Erweiterte Backus-Naur-Form (EBNF) nachfolgend definiert. Dabei gelten folgende Konventionen. Produktionen beginnen mit einem Großbuchstaben. Schlüsselwörter stehen in Anführungszeichen und sind klein geschrieben. Terminalsymbole sind vollständig in Großbuchstaben geschrieben.

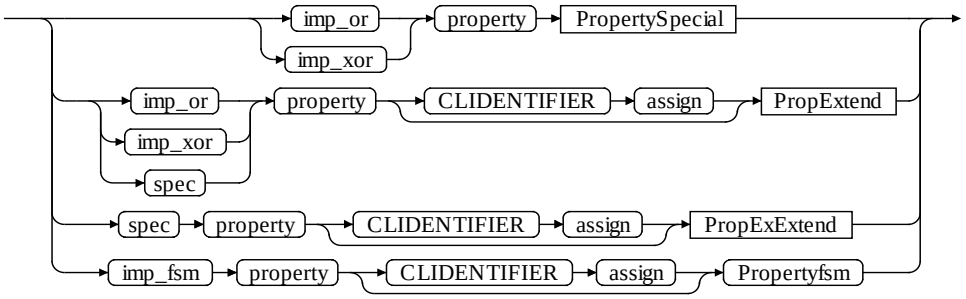
```
Start = { SingleVer }.

SingleVer = Verification ";".

Verification
  = ( "imp_or" | "imp_xor" ) "property" PropertySpecial
  | ( "imp_or" | "imp_xor" | "spec" ) "property"
    [ CLIDENTIFIER "assign" ] PropExtend
  | "spec"    "property" [ CLIDENTIFIER "assign" ] PropExExtend
  | "imp_fsm" "property" [ CLIDENTIFIER "assign" ] Propertyfsm
  .

PropExtend
  = Property
    | Property "reset" PORTEVENT
    | Property ( "enable" | "enablesingle" ) PORTEVENT "disable" PORTEVENT
    .

PropExExtend = PropertyEx
    | PropertyEx "reset" PORTEVENT
    | PropertyEx ( "enable" | "enablesingle" ) PORTEVENT "disable" PORTEVENT
    .

Property
  = ( "always"| "eventually" |"never" )  KonjEvent "in"
    TimeStruct "produce" DisjEvent
  | "every"  TimeStruct "produce" DisjEvent
  | "switch" [ "(" PORTEVENT ")" ] Case { Case } "end"
  .
```

```
PropertyEx                                                              32
  = ( "always"| "eventually" |"never" ) KonjEventEx "in"                33
    TimeStruct "produce" DisjEventEx                                    34
  | "never" konjEventEx "in" TimeStruct "produce" DisjEventEx           35
  | "switch" [ "(" PORTEVENT ")" ] CaseEx { CaseEx } "end"               36
  .                                                                     37
                                                                        38
PropertySpecial = "always" KonjEvent "in" TimeStruct "send" PORTEVENT.  39
                                                                        40
PropertyFsm                                                             41
  = "always" STATENAME "in" TimeStruct "alter" [ "with" TRANSLABEL ]    42
   | "always" STATENAME "in" TimeStruct "allow" STATENAME [ "with" TRANSLABEL ]  43
   .                                                                    44
                                                                        45
Case = "case" ":" KonjEvent "in" TimeStruct "produce" DisjEvent.        46
                                                                        47
CaseEx = "case" ":" KonjEventEx "in" TimeStruct "produce" DisjEventEx.  48
                                                                        49
KonjEvent = ( Event | CountEvent ) { "and" ( Event | CountEvent ) }.    50
                                                                        51
KonjEventEx                                                             52
  = ( Event | CountEvent | STATEEVENT )                                 53
    { "and" ( Event | CountEvent | STATEEVENT ) }                       54
  .                                                                     55
                                                                        56
DisjEvent = ( PORTEVENT | CountEvent ) { "or" ( PORTEVENT | CountEvent ) }.  57
                                                                        58
DisjEventEx                                                             59
  = ( PORTEVENT | CountEvent | STATEEVENT )                             60
    { "or" ( PORTEVENT | CountEvent | STATEEVENT ) }                    61
  .                                                                     62
                                                                        63
Event                                                                   64
  = PORTEVENT                                                           65
  | "start"                                                             66
  .                                                                     67
                                                                        68
CountEvent = "(" STATEEVENT "," NUMERIC ")".                            69
                                                                        70
TimeStruct = "[" TIMEPOINT "," TimePointEx "]".                         71
                                                                        72
TimePointEx                                                             73
  = TIMEPOINT                                                           74
  | "inf"                                                               75
  .                                                                     76
```

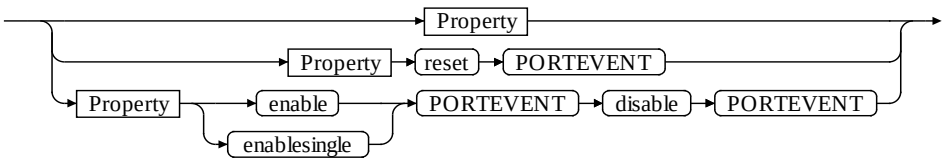
Die nachfolgende Tabelle A.1 zeigt die Produktionen der Syntax der Constraint Language zur Verdeutlichung zusätzlich als Syntaxdiagramm.

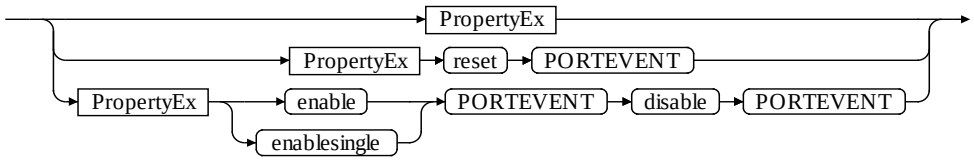
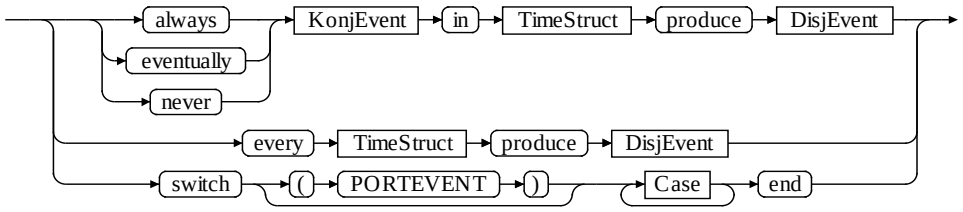
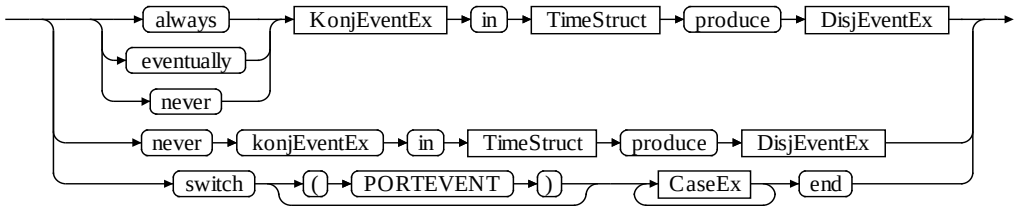
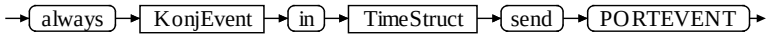
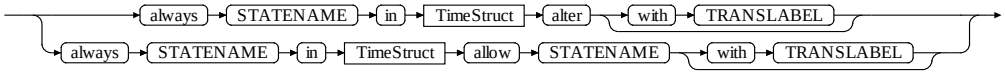
Nr.	Produktionsregel
1	`Start` `= { SingleVer }.`
	
2	`SingleVer` `= Verification ";".`
	

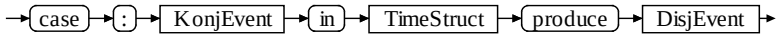
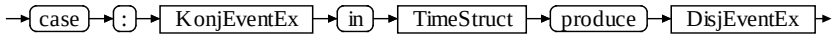
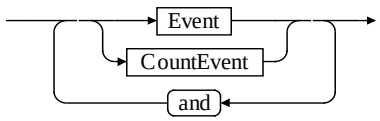
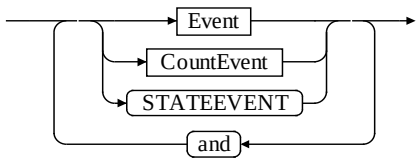
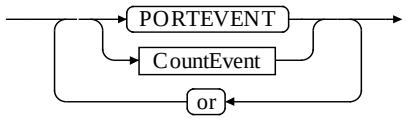
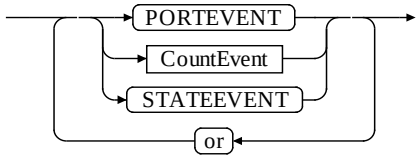
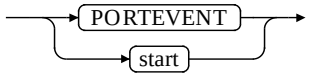
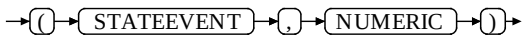
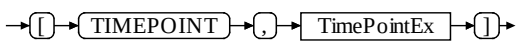
3	`Verification`
	`= ( "imp_or" | "imp_xor" ) "property" PropertySpecial`
	`| ( "imp_or" | "imp_xor" | "spec" ) "property"`
	`[ CLIDENTIFIER "assign" ] PropExtend`
	`| "spec""property"[ CLIDENTIFIER "assign"] PropExExtend`
	`| "imp_fsm""property"[ CLIDENTIFIER "assign"] Propertyfsm.`

4	`PropExtend`
	`= Property`
	`| Property "reset" PORTEVENT`
	`| Property ( "enable" | "enablesingle" ) PORTEVENT "disable"`
	`PORTEVENT.`

Nr.	Produktionsregel
5	PropExExtend = PropertyEx | PropertyEx "reset" PORTEVENT | PropertyEx ("enable" | "enablesingle") PORTEVENT "disable" PORTEVENT.
	
6	Property = ("always" | "eventually" | "never") KonjEvent "in" TimeStruct "produce" DisjEvent | "every" TimeStruct "produce" DisjEvent | "switch" ["(" PORTEVENT ")"] Case { Case } "end".
	
7	PropertyEx = ("always" | "eventually" | "never") KonjEventEx "in" TimeStruct "produce" DisjEventEx | "never" konjEventEx "in" TimeStruct "produce" DisjEventEx | "switch" ["(" PORTEVENT ")"] CaseEx { CaseEx } "end".
	
8	PropertySpecial = "always" KonjEvent "in" TimeStruct "send" PORTEVENT.
	
9	PropertyFsm = "always"STATENAME "in"TimeStruct "alter" ["with" TRANSLABEL] | "always" STATENAME "in" TimeStruct "allow" STATENAME ["with" TRANSLABEL].
	

Nr.	Produktionsregel
10	`Case = "case" ":" KonjEvent "in" TimeStruct "produce" DisjEvent.`
	
11	`CaseEx = "case" ":" KonjEventEx "in" TimeStruct "produce" DisjEventEx.`
	
12	`KonjEvent = ( Event | CountEvent ) { "and" ( Event | CountEvent ) }.`
	
13	`KonjEventEx = ( Event | CountEvent | STATEEVENT )` `{ "and" ( Event | CountEvent | STATEEVENT ) }.`
	
14	`DisjEvent =` `( PORTEVENT | CountEvent ) { "or" ( PORTEVENT | CountEvent ) }.`
	
15	`DisjEventEx = ( PORTEVENT | CountEvent | STATEEVENT )` `{ "or" ( PORTEVENT | CountEvent | STATEEVENT ) }.`
	
16	`Event = PORTEVENT` `| "start".`
	
17	`CountEvent = "(" STATEEVENT "," NUMERIC ")".`
	
18	`TimeStruct = "[" TIMEPOINT "," TimePointEx "]".`
	

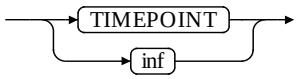
Nr.	Produktionsregel
19	`TimePointEx = TIMEPOINT` `  | "inf".`
	

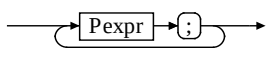
Tabelle A.1.: Struktur der Constraint Language

A.2. XFLTL Beschreibungssprache des Assertion-basierten Ansatzes

Die konkrete Syntax der XFLTL-Spezifikationen ist entsprechend der nachfolgenden Grammatik aufgebaut. Eine *ID* beginnt mit einem "&"-Zeichen (Ampersand), gefolgt von einem Buchstaben und beliebig vielen weiteren Ziffern oder Buchstaben. Als *REAL* können beliebige Gleitkomazahlen verwendet werden, die den Anforderungen des Datentyps *Double* genügen, die Eingabe muss in Dezimalschreibweise erfolgen.

```
Pxfltl                                                                    1
  =  Pexpr ";" { Pexpr ";" }                                              2
  .                                                                       3
                                                                          4
Pexpr                                                                     5
  = ( "(" | "-(" ) Pexpr { "+" Pexpr | "*" Pexpr }                       6
    ( ")"  | ")U" Pexpr | ")U[" REAL "," REAL "]" Pexpr |                 7
      "):=>" Pexpr | ")-->" Pexpr )                                       8
  | ( "F" | "F[" REAL "," REAL "]" ) Pexpr                                9
  | ( "G" | "G[" REAL "," REAL "]" ) Pexpr                                10
  | "=>" Pexpr                                                            11
  | "{" Sqexpr "}"                                                        12
  | Pfactor                                                               13
  .                                                                       14
                                                                          15
Sqexpr                                                                    16
  = Pexpr { "," Sqexpr | ">" Sqexpr }                                     17
  .                                                                       18
                                                                          19
Pfactor                                                                   20
  = "-" Pfactor                                                           21
  | ID                                                                    22
  | REAL ( ">=" ID | "<=" ID | "<" ID | ">" ID | "==" ID | "!=" ID )      23
  .                                                                       24
```

Die nachfolgende Tabelle A.2 zeigt die Produktionen der Syntax der XFLTL zur Verdeutlichung zusätzlich als Syntaxdiagramm.

Nr.	Produktionsregel
1	Pxfltl = Pexpr ";" { Pexpr ";" }. 

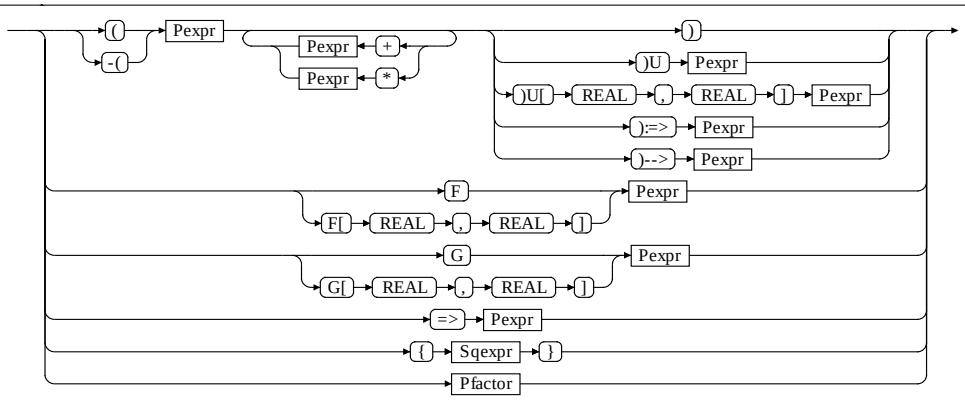
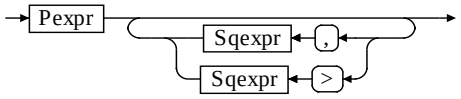
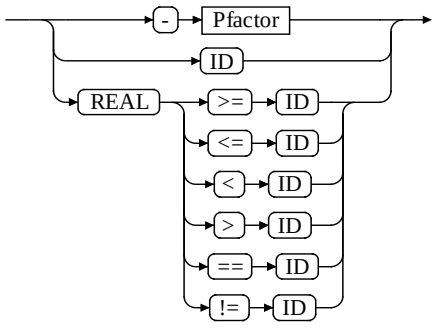
Nr.	Produktionsregel
2	Pexpr = ("(" | "-(") Pexpr { "+" Pexpr | "*" Pexpr } (")" | ")U" Pexpr | ")U[" REAL "," REAL "]" Pexpr | "):=>" Pexpr | ")->" Pexpr) | ("F" | "F[" REAL "," REAL "]") Pexpr | ("G" | "G[" REAL "," REAL "]") Pexpr | "=>" Pexpr | "{" Sqexpr "}" | Pfactor. 
3	Sqexpr = Pexpr { "," Sqexpr | ">" Sqexpr }. 
4	Pfactor = "-" Pfactor | ID | REAL (">=" ID | "<=" ID | "<" ID | ">" ID | "==" ID | "!=" ID). 

Tabelle A.2.: Assertion Specification Language (XFLTL)

B. Anleitung Prototyp Transformation

Die Bedienung des Prototyps *MLD2UPPAL* des Transformation-basierten Ansatzes erfolgt über eine grafische Benutzeroberfläche. Zur Verifikation sind für den Benutzer drei Schritte erforderlich: Eingabe, Generierung und Verifikation. Jedem dieser Schritte wurde eine eigene Eingabemaske gewidmet. Die Bedienung besteht dabei lediglich aus der Konfiguration, sowie ggf. der Behebung von Eingabefehlern. Erfolg bzw. Misserfolg der einzelnen Operationen werden jeweils im *„Console output“* angezeigt.

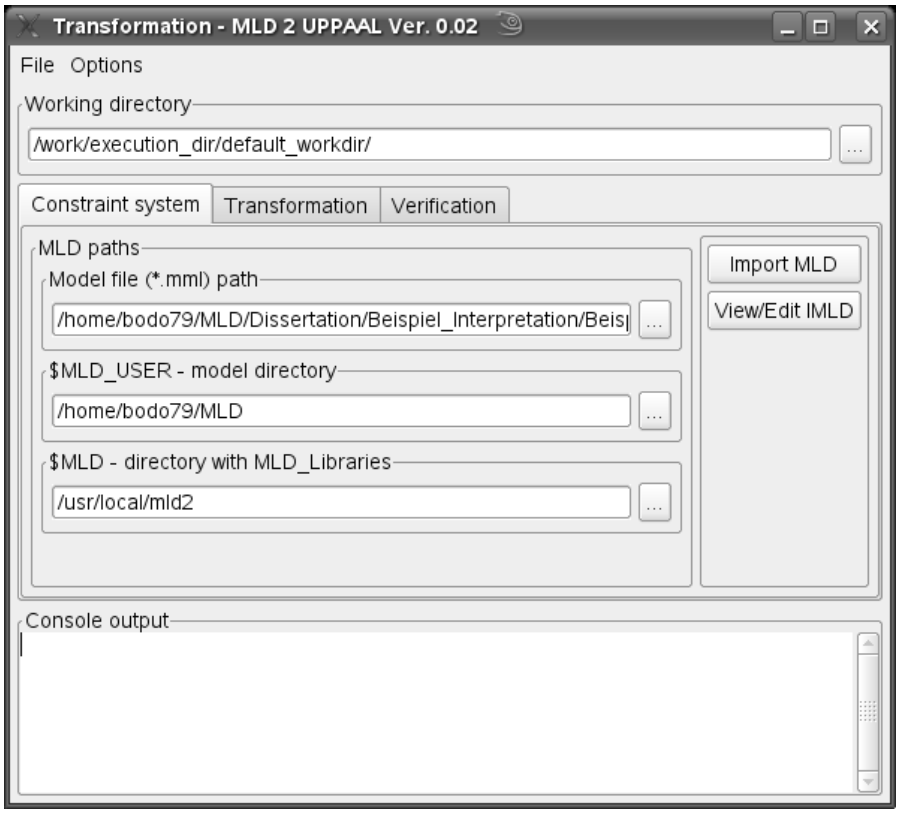


Abbildung B.1.: Screenshot MLD2Uppaal - Eingabe

Der erste Tab *„Contraint System“*, siehe Abbildung B.1, dient der Eingabe des MLDesigner Modells. Neben dem zu prüfenden System sind lediglich die Pfade für Bibliotheks- und Benutzerverzeichnis auszuwählen. Der Button *„import MLD“* dient zum Einlesen des Modells und der Button *„View/Edit IMLD“* öffnet einen Dialog zum Betrachten und Editieren der Annotationen.

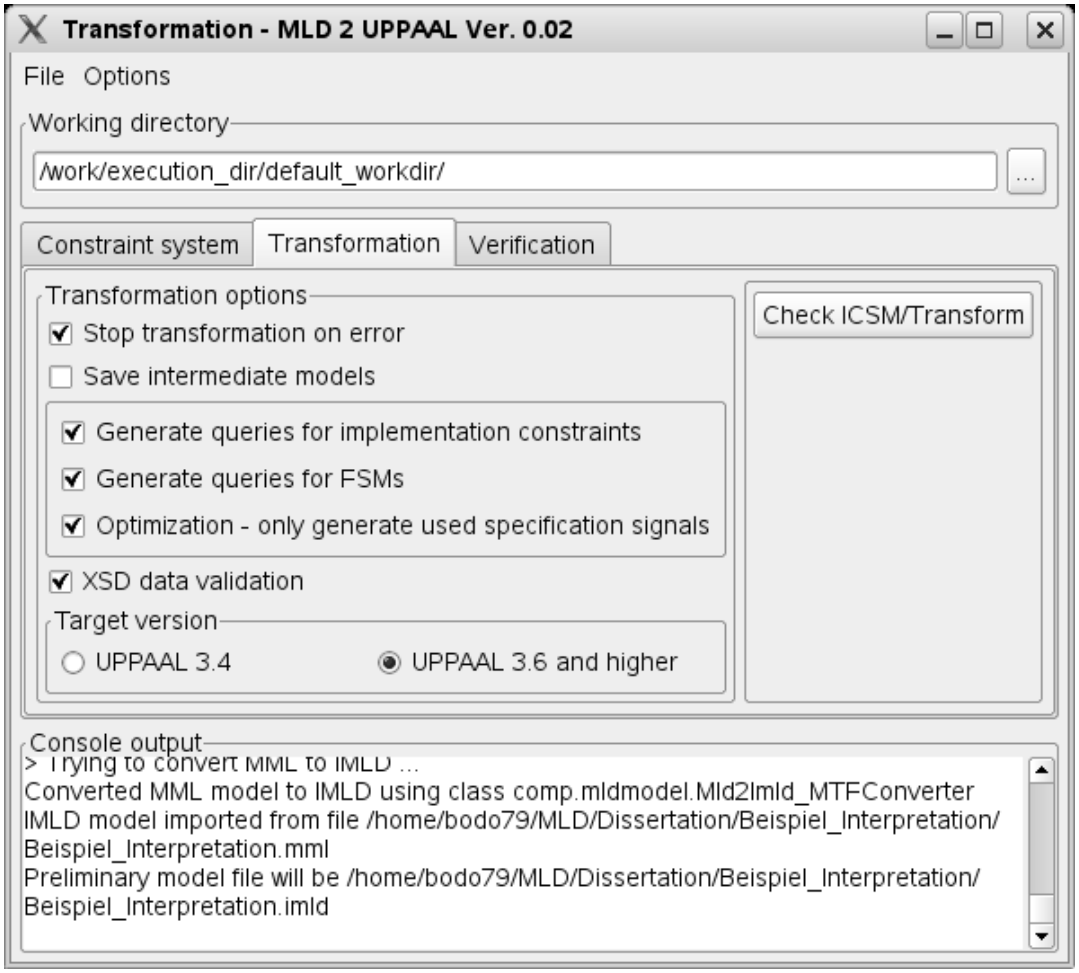


Abbildung B.2.: Screenshot MLD2Uppaal - Transformation

Der zweite Tab *„Transformation“*, siehe Abbildung B.2, dient der Konfiguration zur Generierung des Analysemodells als Uppaal Timed Automata. Die Optionen *„Stop transformation on error“* und *„Save intermediate models“* dienen dem Debugging des Prototypen. Die Optionen *„Generate queries for implementation constraints“* und *„generate queries for FSMs“* dienen der Konfiguration der generierten CTL-Anfragen in *Uppaal*. Die Option *„Optimization - only generate used specification signals“* dient der Optimierung, wobei die für die Verifikation strukturell nicht relevanten Teile nicht erzeugt werden (unabhängige Teilmodelle).

Schließlich besteht die Möglichkeit das Format der genutzten *Uppaal* Version auszuwählen und das Ausgabemodell syntaktisch gegen das XSD-Schema zu validieren. Die Generierung wird mittels des Button *„Check ICSM/Transform“* gestartet und liegt anschließend als Uppaal-Projekt vor. Dazu wird eine *.xml*-Datei mit dem Automatensystem sowie eine *.q*-Datei mit den Anfragen (Queries) erzeugt.

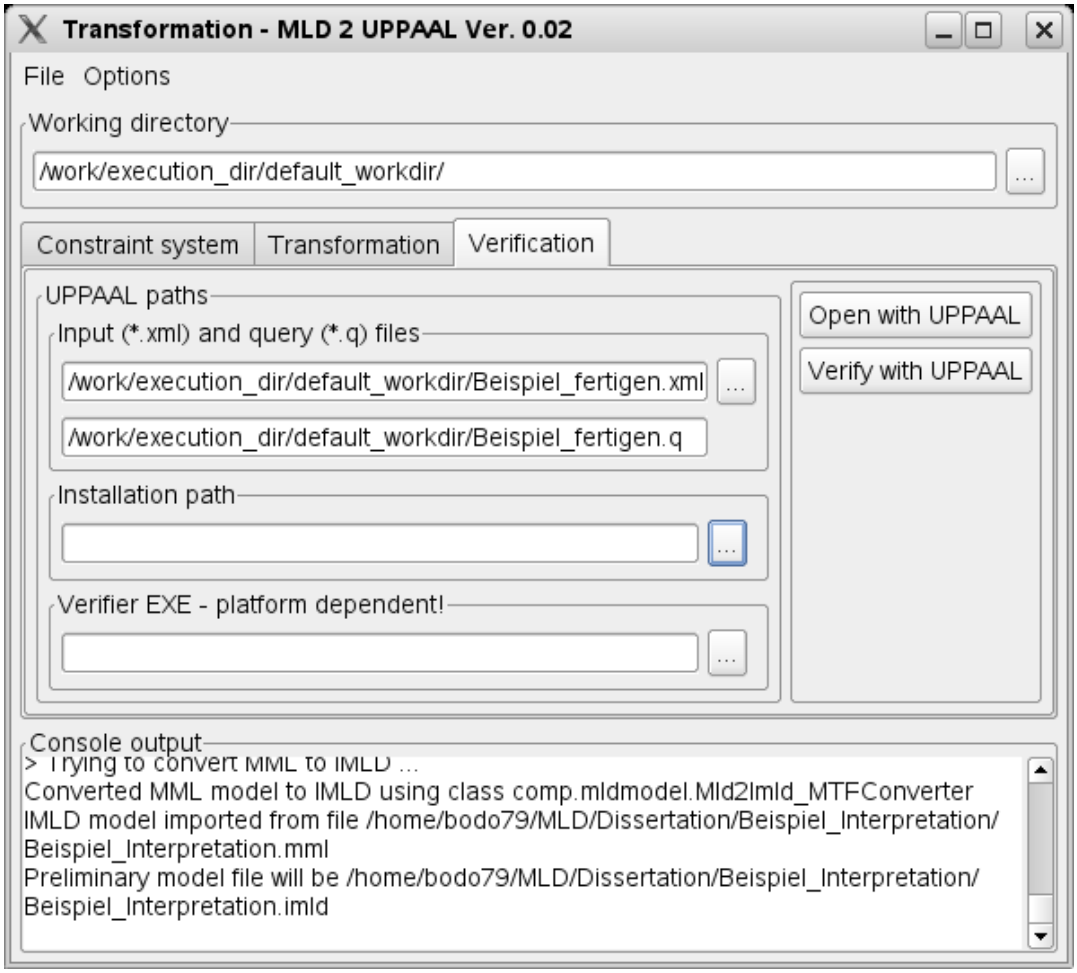


Abbildung B.3.: Screenshot MLD2Uppaal - Ausgabe

Der letzte Tab *„Verification“*, siehe Abbildung B.3 dient der Konfiguration zum Starten der Verifikation. Dazu ist die Angabe der Dateipfade und der Namen des Ausgabemodells sowie der Queries erforderlich. Weiterhin sind die Pfade des Uppaal-Editors bzw. des Verifiers anzugeben. Als Aktion kann das generierte Modell mit dem Button *„Open with UPPAAL“* geöffnet werden. Zudem ist die Prüfung durch den Verifier direkt über den Button *„Verify with UPPAAL“* möglich. Die Ergebnisse werden im *„Console output“* präsentert.

C. Anleitung Prototyp Assertion Checking

Assertion Checking verfolgt das Ziel, (formal) definierte Eigenschaften während der Laufzeit eines Systems (Simulation eines Modells) zu prüfen. Der realisierte Prototyp dient der Prüfung von *XFLTL*-Eigenschaften während der Simulation im Entwicklungs- und Simulationswerkzeug MLDesigner [MLD07]. *XFLTL*-Eigenschaften beschreiben temporallogische Eigenschaften bezüglich des Auftretens von Events (Simulationsereignissen).

Für die Bereitstellung einer solchen Funktionalität existieren prinzipiell mehrere Möglichkeiten. Einerseits könnte die Funktionalität ins Simulationswerkzeug selbst integriert werden (über die IDE oder das Simulationstarget[71]), andererseits besteht die Möglichkeit, die Funktionalität als Element einer speziellen Bibliothek in das Modell zu integrieren. Aus Gründen der Flexibilität wurde wurde letztere Variante gewählt, wobei zudem das Problem der eineindeutigen Identifikation von Signalen, siehe unten, umgangen wird.

Für die Assertion-basierte Prüfung sind zwei zusätzliche Phasen erforderlich. Zunächst muss die Spezifikation der Eigenschaften erfolgen. Anschließend erfolgt die Simulation und Auswertung der Eigenschaften.

In Abbildung C.1 sind drei Modellelemente des Typs „*PulseGen*“ zu sehen, die Signale erzeugen. Diese sind mit den Modellelementen des Typs „*SFormula1*“ verbunden, die das Assertion Checking realisieren. Die Auswertung erfolgt schließlich über die Modellelemente „*ShowFormulas*“ und „*ShowFormStat*“.

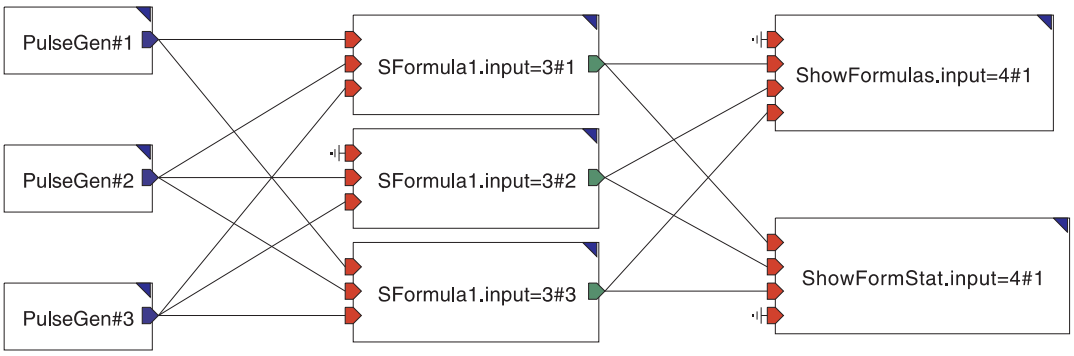


Abbildung C.1.: Beispielmodell für den Assertion Checker.

[71] Das Simulationstarget stellt eine Abstraktionsschicht zwischen Modell und Entwicklungswerkzeug dar, die angepasst und erweitert werden kann. Allerdings ist diese Funktionalität dem Nutzer nicht mehr transparent und kann ggf. nur schwer geprüft werden.

Für die Spezifikation wird das Modellelement „*SFormula1*“ genutzt, dessen Eingänge mit Signalen des Modells verbunden werden, siehe Abbildung C.1. Die Signale des zu analysierenden Modells besitzen im Gesamtmodell keine einheitliche Bezeichnung, da die Adressierung jeweils lokal im Modellelement bezüglich der verbindenden Eingabeports realisiert wird. Daher müssen den eingehenden Signalen Bezeichnungen zugeordnet werden, die auch innerhalb der Eigenschaften verwendet werden können. Dies ist über den Parameter „*Mapping*“ realisiert, siehe Abbildung C.2. Jeder Bezeichner muss mit einem Ampersand „&“ beginnen, die Trennung mehrerer Bezeichner erfolgt über Leerzeichen. Die Zuordnung zwischen Eingabeport und Bezeichner erfolgt entsprechend der Nummerierung bzw. Reihenfolge, so dass dem „`input#1`“ der Bezeichner „`&x`“, dem „`input#2`“ der Bezeichner „`&y`“ usw. zugeordnet wird.

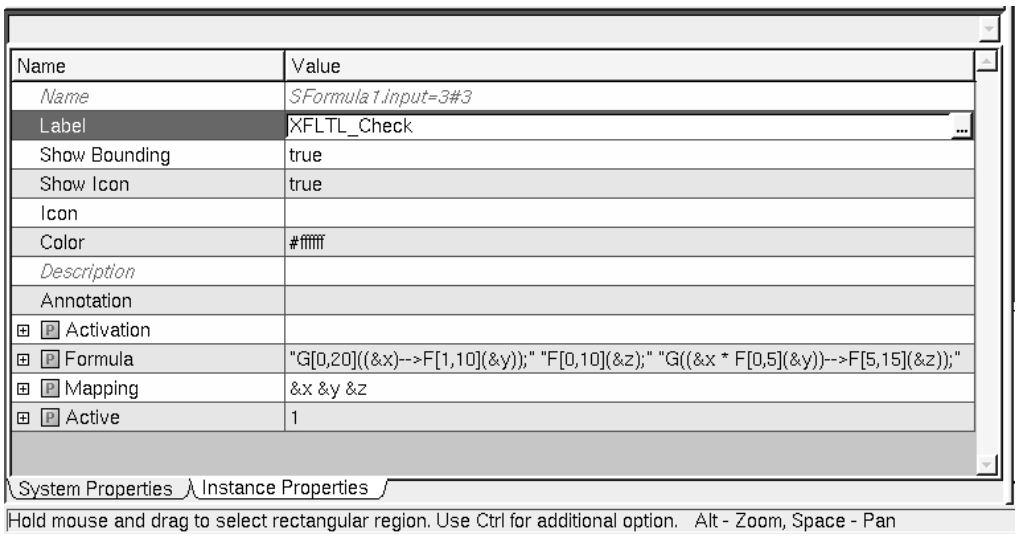


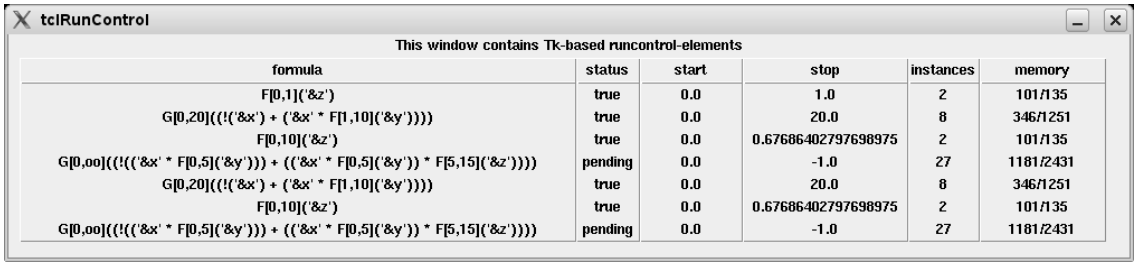
Abbildung C.2.: Parameter des „SFormula1“ Primitivs.

Des weiteren werden die zu prüfenden Eigenschaften als String-Array im Parameter „*Formula*“ notiert, die sich auf die Bezeichner beziehen, siehe Abbildung C.2. Wie in Abbildung C.2 dargestellt, sind als zu prüfende Eigenschaften die Formeln „`"G[0,20]((&x)->F[1,10](&y));"`“, „`"F[0,10](&z);"`“ und „`"G[0,20]((&x * F[0,5](&y))->F[5,15](&z));"`“ notiert. Die konkrete Syntax der Grammatik der *XFLTL* ist im Anhang A.2 zu finden. Somit können für die Eingänge beliebige Bezeichner gewählt und in den Formeln als elementare Aussagen verwendet werden.

Zu Beginn der Simulation werden Bezeichner und die definierten Formeln (Eigenschaften) bezüglich der Konformität geprüft. Über den Parameter „*Active*“ wird definiert, ob eine Prüfung stattfindet (’1’) oder nicht (’0’). Die Prüfung selbst wird an eine C++ Klassenbibliothek delegiert. Daher muss der Objektcode der Klassenbibliothek mit dem Primitiv verknüpft werden.

Während der Prüfung werden Message Objekte erzeugt, die am Ausgang „*Output1*“ des Pri-

mitivs verfügbar sind. Diese Objekte enthalten Informationen über den Zustand der Formel und können zur Auswertung verwendet werden. Abbildung C.3 zeigt eine tabellarische Übersicht der zu prüfenden Assertions, die das Modellelement „*ShowFormStat*“ erzeugt. Die erste Spalte zeigt die Formel, und die zweite den Status in dreiwertiger Logik. Die Spalte „*start*“ zeigt den Startzeitpunkt für die erste erfolgte Auswertung. Eine „`-1`“ markiert eine (noch) nicht gestartete Assertion. Die Spalte „*stop*“ zeigt den Endzeitpunkt der Prüfung. Hier deutet die „`-1`“ auf eine noch laufende Prüfung hin. Die Spalte „*instances*“ notiert die verwendete Anzahl an Formelobjekten und die Spalte „*memory*“ die Menge des benutzten Datenspeichers in Byte.



formula	status	start	stop	instances	memory
F[0,1]('&z')	true	0.0	1.0	2	101/135
G[0,20](('!('&x') + ('&x' * F[1,10]('&y'))))	true	0.0	20.0	8	346/1251
F[0,10]('&z')	true	0.0	0.67686402797698975	2	101/135
G[0,oo](('!(('&x' * F[0,5]('&y'))) + (('&x' * F[0,5]('&y')) * F[5,15]('&z'))))	pending	0.0	-1.0	27	1181/2431
G[0,20](('!('&x') + ('&x' * F[1,10]('&y'))))	true	0.0	20.0	8	346/1251
F[0,10]('&z')	true	0.0	0.67686402797698975	2	101/135
G[0,oo](('!(('&x' * F[0,5]('&y'))) + (('&x' * F[0,5]('&y')) * F[5,15]('&z'))))	pending	0.0	-1.0	27	1181/2431

Abbildung C.3.: Statistik zum Assertion Checking

Eine Analyse wird des weiteren durch die Verlaufsdarstellung (Trace) der Eigenschaftserfüllung durch das Primitiv „*ShowFormulas*“ ermöglicht, siehe Abbildung C.4. Dabei wird der Verlauf der Eigenschaftserfüllung farbig markiert. *Grün* markiert ein Zeitintervall in dem die Formel terminal erfüllt ist, *rot* beschreibt ein Zeitintervall in dem die Formel terminal nicht erfüllt ist und mit *blau* werden Intervalle markiert, in denen eine abschließende Bewertung noch nicht möglich ist. Für eine abschließende Bewertung der Eigenschaften muss darauf geachtet werden, dass die Primitive des Typs „*SFormula1*“ am Ende der Simulation ein Event erhält.

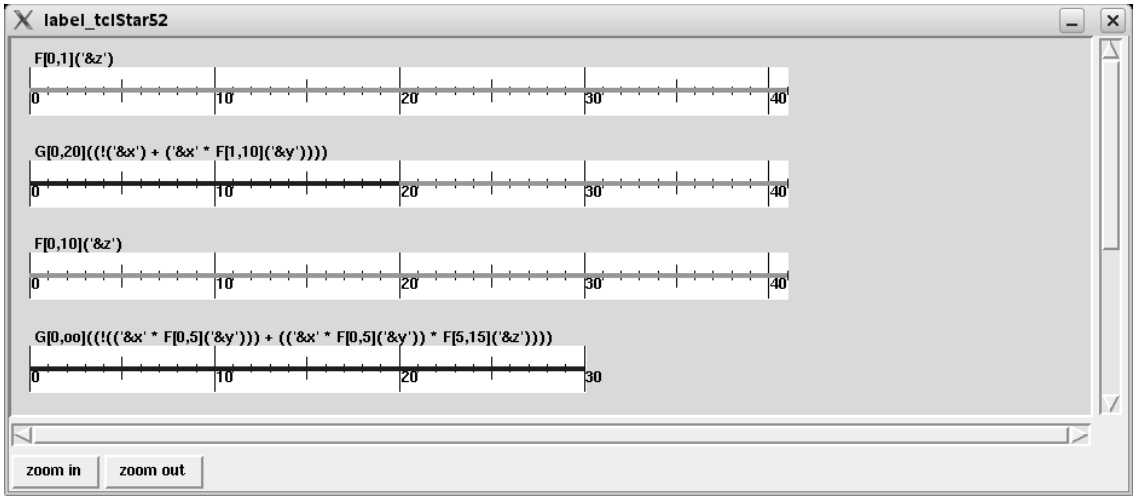


Abbildung C.4.: Trace der Eigenschaftsbewertungen

D. Abstraktion und Verfeinerung von Prozessen

Die Prüfung von Eigenschaften auf der Basis des dynamischen Verhaltens ist oft mit einem hohen (Rechen-) Aufwand verbunden. Bei Verwendung von Model Checking Algorithmen ist der Zustandsraum ganz oder zumindest teilweise zu prüfen, wobei, je nach Verfahren, Eigenschaften nach LTL, CTL oder CTL* geprüft werden können [Kro99].
In diesem Abschnitt wird die Prüfung von prozessbasierten Zeiteigenschaften auf Basis einer eigenschaftserhaltenden Strukturanalyse als alternative Herangehensweise betrachtet. Diese stützt sich auf Erkenntnisse, die auf [FGM98] und [PZS99] zurückgehen. Bei der Strukturanalyse werden Eigenschaften durch deren Expansion oder aber durch Reduktion des Modells erreicht - schließlich müssen die reduzierten Modelle strukturell und bezüglich ihres (eingeschränkten) Zustandsraumes äquivalent sein. Während bei den Verfeinerungen aus [FGM98] streng strukturierte Netze genutzt werden und die Gültigkeit der Regeln sichere Netze voraussetzen (genau eine Marke im betrachteten Modell), muss diese Methodik für die Ausweitung auf nicht sichere Netze angepasst werden.

Für die nachfolgenden Betrachtungen wird davon ausgegangen, dass es sich um einfache prozessbasierte Eigenschaften gemäß Abschnitt 4.6 handelt, die gemäß den Betrachtungen nach Abschnitt 5.3.2 als Petri-Netz dargestellt sind.

D.1. Reduktionsmechanismen

Reduktionsmechanismen [DA05] sind in der Regel derart aufgebaut, dass sich der Erreichbarkeitsgraph bzgl. bestimmter Erreichbarkeitsklassen nicht ändert. Im speziellen werden sukzessive Transitionen bzw. Plätze zusammengefasst, aus dem ursprünglichen Netz entfernt, oder durch andere Strukturen ersetzt.

D.1.1. Verfeinerungsanalyse

Die in [FGM98] in der TRIO Semantik definierten Verfeinerungen scheinen am besten zur Eventsemantik des Eigenschaftsgraphen zu passen. Zu beachten sind in jedem Fall die Modellgrenzen. So lassen sich nur Aussagen treffen, die von der Sequenz der Eingangsereignisse unabhängig sind. Insbesondere lässt sich keine Aussage über aufeinander folgende Ereignisse des

gleichen Typs treffen, da auf oberster Abstraktionsebene ein einzelnes Synchronisationssignal als Ausgangspunkt dient. Dabei sollen folgende Konventionen gelten[72]:

- Bei der Verwendung von Verfeinerungen werden nur solche Anforderungsknoten betrachtet, die sich als Petri-Netz darstellen lassen. Diese stellen jeweils eine abstrakte Eigenschaft dar.

- Eine abstrakte Eigenschaft muss mit einer Teilmenge der Eigenschaftsknoten in einer Verfeinerungsbeziehung stehen; die geforderte Eigenschaft muss sich durch eine verfeinerte Darstellung, unter Verwendung der Eigenschaftsknoten darstellen lassen. Die verfeinerte Darstellung kann durch eine beliebige, endliche Anzahl an Verfeinerungsschritten erzeugt werden.

D.1.2. Intervallarithmetik

Bei den durch Intervalle I beschriebenen quantitativen Eigenschaften der Form

$$\Phi(e, I) \vDash \exists t \in I \mid e(t) = \mathit{true}$$

von Ereignissen e besteht zwischen den abstrakten und konkreten Eigenschaften eine Teilmengenbeziehung. Dabei ist zusätzlich gefordert, dass die Mengen nicht leer sind, denn eine leere Eigenschaft ist nicht von praktischer Bedeutung.[73]

$$\Phi_{abstrakt} \supseteq \Phi_{konkret} \text{ gdw. } e_{abstrakt} \supseteq e_{konkret} \wedge I_{abstrakt} \supseteq I_{konkret} \tag{D.1}$$

$$\Phi_{abstrakt} \neq \emptyset \tag{D.2}$$

$$\Phi_{konkret} \neq \emptyset \tag{D.3}$$

Für das Aufstellen von Theoremen für die Bewertung konkreter Zeitpunktmengen lässt sich also die Erfüllbarkeit von Intervalleigenschaften aus der Teilmengenbeziehung derart ableiten, dass wenn die konkrete Intervallmenge der Teilmengenbeziehung genügt und nicht leer ist, die abstrakte Eigenschaft als erfüllt gilt.

$$\Phi_{konkret} \implies \Phi_{abstrakt} \text{ gdw. } \Phi_{abstrakt} \overset{!}{\supseteq} \Phi_{konkret} \wedge \Phi_{konkret} \neq \emptyset \tag{D.4}$$

Die Aufstellung eines zusätzlichen Axioms über leere Mengen bzw. leere Intervalle ist jedoch problematisch, da sie zusätzlich eine Information über die Art der Eigenschaft, konkret oder

[72] Statt eines Petri-Netzes kann auch eine andere Darstellung verwendet werden, bei der die Eigenschaften, insbesondere die Zeitbedingungen erhalten bleiben.

[73] Im Prinzip entzieht sich eine leere Eigenschaft der Entscheidbarkeit ihrer Erfüllung. Es mag jedoch für ein konkretes Axiomensystem sinnvoll sein, eine leere Eigenschaft als *false* zu bewerten.

abstrakt, erfordert. Für allgemeine Mengen ist die Überprüfung der Teilmengenbeziehung erforderlich. Diese kann für den Einzelfall ggf. sehr aufwendig sein, da jedes Element der Menge geprüft werden muss. Für Zeiteigenschaften wird daher im Allgemeinen eine Intervalldarstellung verwendet. Bei Realzeitautomaten etwa wird der Wertebereich von Zeitvariablen im Zustandsraum als Intervall dargestellt. Bei mehr als einer Variablen kommt dabei der Regionengraph als mathematische Repräsentation zum Einsatz, der neben den Intervalleigenschaften auch die chronologischen Relationen der Zeitvariablen zueinander berücksichtigt [Löt99]. Beschränkt man sich auf Intervallmengen, so genügt die Auswertung der Intervallgrenzen und die Anwendung einer Intervallarithmetik. Im Folgenden werden Zeitintervalle, arithmetische Operationen und Vergleichsoperationen definiert. Für die Zeitintervalle sollen folgende Konventionen gelten:

- Leeres Intervall: $\emptyset$
- Abgeschlossenes Intervall:$[a, b] = \{x \in \mathbb{R}^+ | a \leq x \leq b\}$
- Offenes Intervall:$(a, b) = \{x \in \mathbb{R}^+ | a < x < b\}$
- Halboffenes (rechtsoffenes) Intervall:$[a, b) = \{x \in \mathbb{R}^+ | a \leq x < b\}$
- Halboffenes (linksoffenes) Intervall:$(a, b] = \{x \in \mathbb{R}^+ | a < x \leq b\}$

Unendliche Intervalle sollen nicht zugelassen sein, da sie eine unzureichende Restriktion repräsentieren. Da arithmetische Operationen zwischen (halb-) offenen Intervallen problematisch sind, sollen sie ebenfalls verboten sein.[74] Die Intervalle sind auf den Bereich der positiven reellen Zahlen beschränkt, da eine negative Zeitdauer den physikalischen Gegebenheiten des Kausalitätsgrundsatzes widerspricht. Demzufolge gilt für die obigen Intervallgrenzen a, b: $|a| = |b|$. Ein leeres Intervall kann bei der Schnittmenge (Minimum) entstehen. Zum Ausführen von Operationen ist die Definition einer Intervallarithmetik erforderlich. Für arithmetische Operationen auf geschlossene, nichtleere Intervalle und reellwertige skalare Größen sollen folgende Konventionen gelten:

- Addition: $[a_1, b_1] + [a_2, b_2] = [(a_1 + a_2), (b_1 + b_2)]$
- Subtraktion: $[a_1, b_1] - [a_2, b_2] = [(a_1 - a_2), (b_1 - b_2)]$
- Multiplikation: $[a_1, b_1] \cdot [a_2, b_2] = [(a_1 \cdot a_2), (b_1 \cdot b_2)]$
- Skalare Multiplikation: $x \cdot [a, b] = [(x \cdot a), (x \cdot b)]$

[74] Bei (halb-) offenen Intervallen lässt sich die Grenze, also der frühmöglichste bzw. spätmöglichste Zeitpunkt nicht bestimmen. Numerisch gesehen, liegt er jeweils um ϵ neben der Grenze, wobei das konkrete ϵ von der gewählten Realisierung abhängig ist. Dies gilt ebenso für arithmetische Berechnungen, die Methodik wäre nicht mehr allgemein. Für symbolische Berechnungen ist zwar die Definition $\epsilon = 1/\infty$ möglich, würde aber die Operation mit unterbrochenen Intervallen erfordern, die als Vereinigung offener Intervalle entstehen können.

- Division: $[a_1, b_1]/[a_2, b_2] = [(a_1/b_2), (b_1/a_2)]$ für $b_2 \neq 0, a_2 \neq 0$
- Betrag (Dauer): $|[a, b]| = b - a$
- Vereinigung / Maximum: $\bigcup_{i=1}^{n}([a_i, b_i]) = max_{i=1}^{n}([a_i, b_i]) = [min(a_i), max(b_i)]$
- Synchronisation: $S_{i=1}^{n}([a_i, b_i]) = sync_{i=1}^{n}([a_i, b_i]) = [max(a_i), max(b_i)]$
- Schnitt / Minimum:$\bigcap_{i=1}^{n}([a_i, b_i]) = \left\{ \begin{array}{ccc} max(a_i) \leq max(b_i) & | & [max(a_i), min(b_i)] \\ max(a_i) > max(b_i) & | & \emptyset \end{array} \right\}$

Für die Bewertung von Eigenschaften sind weiterhin Vergleichsoperatoren für Intervalle erforderlich, welche die Erzeugung logischer (boolescher) Aussagen ermöglichen. Für logische Aussagen bezüglich relationalen Operationen auf Intervalle sollen folgende Konventionen gelten:

- Subintervall:$[a_1, b_1] \subseteq [a_2, b_2] \Rightarrow \left\{ \begin{array}{ccc} (a_1 \geq a_2) \wedge (b_1 \leq b_2) & | & true \\ sonst & | & false \end{array} \right\}$
- Äquivalenz:$[a_1, b_1] \equiv [a_2, b_2] \Rightarrow \left\{ \begin{array}{ccc} (a_1 = a_2) \wedge (b_1 = b_2) & | & true \\ sonst & | & false \end{array} \right\}$

D.1.3. Beschränkungen und offene Probleme

Reduktionsmechanismen können keine Änderungen vornehmen, welche die Erreichbarkeit von Teilmarkierungen verändert. Für die Abstraktion von Prozesseigenschaften wird aber gerade diese Art der Verfeinerung gewünscht, also dass die Interna komplexer Prozesse abstrahiert wird. Die Verfeinerungsanalyse bringt bezüglich ihrer Anwendbarkeit zwei Probleme mit sich:

- Die Aussagen sind bezüglich der Verfeinerung korrekt, nicht aber bezüglich der Abstraktion. Diese Einschränkung gilt für Prozesse, bei denen die Menge der Eingangsmarken nicht für die Gesamtzeit des betrachteten Teilnetzes beschränkt ist.
- Der Ausgangsprozess enthält offenbar keine nutzbare Parallelität. Er kann nur zur Modellierung einzelner, exklusiver Prozesse dienen.

Ursache dieser Beschränkungen ist die Vernachlässigung von Parallelität, die innerhalb des modellierten Prozesses enthalten sein kann. Innerhalb von Systemen der digitalen Infrastruktur werden jedoch oft nebenläufige Prozesse verwendet, da hierbei neben der Latenzzeit vor allem auch der Durchsatz entscheidend ist. Zur Analyse dieser Problemklasse ist eine Erweiterung der Modellierungstechnik und der Verfeinerungsanalyse notwendig.

D.2. Erweiterung der Modellklasse

Für die Modellierung (Beschränkung) von Parallelität bzw. Markendurchsatz in Intervall Petri-Netzen fehlen explizite Beschreibungsmöglichkeiten. Der Markendurchsatz von Prozessen bzw. Transitionen beschränkt die mögliche Eventfunktion an den Vorplätzen, also wie viele Marken (Events) pro Zeiteinheit bearbeitet werden können. Eine Nichtbeachtung dieser Einschränkung führt bezüglich der Latenz zu zusätzlichen Wartezeiten, das System wäre mit der gegebenen Methodik nicht mehr analysierbar. Die Parallelität beschreibt, wie oft ein Prozess (Transition) simultan aktiviert sein kann, also wie lang der betrachtete Ausschnitt einer Eventfunktion ist.

D.2.1. Multiple Transition

Eine multiple Transition enthält neben dem Zeitintervall zwei zusätzliche Parameter:

- Markendurchsatz K: Der Markendurchsatz K charakterisiert die minimale Wartezeit D_{in} zwischen der Konsumierung zweier aufeinander folgender Marken (Events) im Vorplatz. $K = \frac{1}{D_{in}}$
- Parallelität/ Nebenläufigkeit N: Ein zusätzlicher Parameter N kennzeichnet die Anzahl der simultan möglichen Aktivierungen einer multiplen Transition. $N \geq \frac{Max(I)}{D_{in}}$.

Das Petri-Netz Modell wurde somit um einen speziellen Transitionstyp erweitert, der die Modellierung eines Prozesses mit nebenläufigen internen Komponenten erlaubt.

D.2.2. Ersatzkonstruktion

Eine Ersatzkonstruktion dient der effizienten Abbildung des erweiterten Konzeptes in das existierende Petri-Netz Modell. Zwei offensichtliche Möglichkeiten existieren in der rein parallelen bzw. rein sequentiellen Ersetzung des parallelen Prozesses. Die sequentielle Ersetzung (siehe Abbildung D.1) besteht aus:

- einer Nulltransition zu einem Sammelplatz, deren Konzession über ein Netz, bestehend aus zwei Plätzen mit einer zusätzlichen Transition, das als Nebenbedingung die Schaltabstände der Nulltransition realisiert,
- N sequentiellen Transitionen und 2-beschränkten Plätzen. Die Summe der Intervalle muss dem Ersatzintervall entsprechen.
- Vorteilhaft sind balancierte Intervalle, bei denen versucht wird Jitter zu vermeiden. Dies führt zu einem stärkeren Determinismus im Modell und dadurch zu einem kleineren Zustandsraum. Die statistischen Eigenschaften sind allerdings von der konkreten Aufteilung der Teilintervalle abhängig.

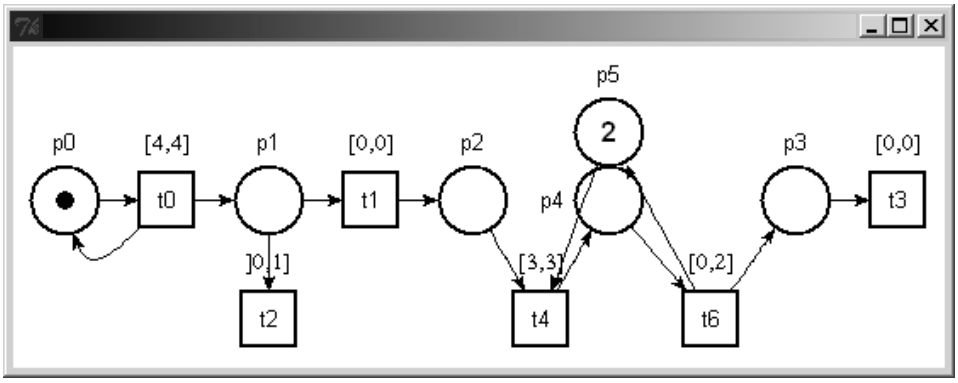


Abbildung D.1.: Sequenzielle Ersetzung

Die parallele Ersatzkonstruktion (siehe Abbildung D.2) der Transition besteht aus:

- einer Nulltransition zu einem Sammelplatz, deren Konzession über ein Netz, bestehend aus zwei Plätzen mit einer zusätzlichen Transition, das als Nebenbedingung die Schaltabstände der Nulltransition realisiert,
- N alternativen Nulltransitionen mit 1-beschränktem Platz und je einer Transition für das Prozessintervall.
- Eine Determinierung der Alternativen beschränkt den Zustandsraum für dynamische Untersuchungen. Zu empfehlen ist hier etwa die sequentielle Aktivierung der alternativen Transitionen, wie in Abbildung D.2.

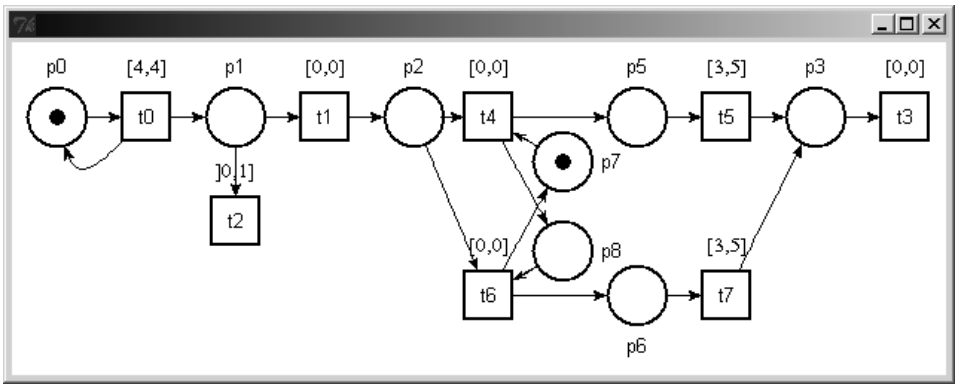


Abbildung D.2.: Parallele Ersetzung

Im parallelen Ersatzmodell ergibt sich eine weitere Möglichkeit der Reduktion für die Menge der nebenläufigen Transitionen mit dem Prozessintervall aus der Sortierung der Intervalle der Transitionen. Dadurch lässt sich die Reduktion von Symmetrie auf der Ebene des Zustandsraumes erreichen.

Die sequentielle Ersetzung bietet den Vorteil, dass kein Determinismus modelliert werden braucht und die Ausgabe der Events, bezüglich der Eingabewerte streng sequentiell erfolgt. Die Beibehaltung der Sequenz bleibt bei der parallelen Ersetzung nur gewahrt, wenn $J_{out} \leq D_{in}$, also $J_i + J_{in} \leq D_{in}$ gilt.

D.3. Erweiterte Verfeinerungsanalyse

D.3.1. Umgebungsmodell

Wie bereits erwähnt, ist das für die Verfeinerungsanalyse verwendete Umgebungsmodell besonders wichtig. Vorausgesetzt wird für die Verfeinerungsanalyse eine ratenmonotone, ggf. mit Jitter behaftete Aktivierung der Prozesse. Das Umgebungsmodell kann also durch einen ratenmonotonen Eventgenerator mit nachgeschalteter Jittererzeugung modelliert werden. Das Umgebungsmodell ist demnach gekennzeichnet durch D_{out} und J_{out}. Damit Umgebungsmodell und Prozess miteinander korrekt interagieren, (die Konsumierbarkeit der Events muss sichergestellt sein) muss $D_{out}(P_0)+J_{out}(P_0) \leq D_{in}(P_1)$ gelten. Die Fortpflanzung der Jittereffekte wird damit umgangen. Eine weitere Alternative besteht in der Resynchronisation, also im Ausgleich der Jittereffekte der vorherigen Prozesse, wodurch die maximale Gesamtlatenz nicht beeinflusst wird. $D_{out}(P_0) = D_{in}(P_1)$

D.3.2. Sequentielle Prozessketten und Jitter

Jitter führt im Allgemeinen zu zeitlichen Verschiebungen innerhalb einer Prozesskette. Im Zusammenhang mit einem nachfolgenden Prozess können sich Wartezeiten ergeben. Diese Wartezeiten treten allerdings nur auf, wenn auf eine langsame Prozesskette (geringe Aktivierungsrate) eine schnelle Prozesskette (hohe Aktivierungsrate) folgt. Allerdings wird die schnelle Prozesskette maximal soweit verzögert, dass ihre Aktivierungsrate der der langsamen Prozesskette entspricht, so dass sich aus der Latenz ein erhöhter Speicherbedarf ergibt.

Für einen Prozess muss die Bedingung $D_{i_{in}} \geq J_{i_{out}} + Max(P_{i+1})$ erfüllt sein, damit keine zusätzliche Latenz auftritt. Die zusätzliche Latenz ergibt sich zu $L_J = J(P_i)+J_{in}+Max(P_{i+1})-D_{in}$, und erfordert eine zusätzliche Zwischenspeicherung von Daten.

$$L_J = \begin{cases} 0 & | \quad J(P_i) + J_{in} + Max(P_{i+1}) - D_{in} \leq 0 \\ J(P_i) + J_{in} + Max(P_{i+1}) - D_{in} & | \quad J(P_i) + J_{in} + Max(P_{i+1}) - D_{in} > 0 \end{cases}$$

- **Zeitintervall:** $I = [\sum_{i=1..n} (I_i min), \sum_{i=1..n} (I_i max)]$
- **Parallelität:** $N = \sum_{i=1..n} (N_i)$
- **Markendurchsatz:** $K = \overset{Min}{i=1..n} (K_i)$

D.3.3. Parallele Prozesse

Bei parallelen Prozessen wird eine Synchronisation an den Anfangs- und Endpunkten vorausgesetzt, wobei die Prozesse selbst eine interne Struktur besitzen können. Soweit ergibt sich in der Abstraktion ein Bearbeitungsintervall, aus den Maxima der jeweiligen Intervallgrenze.

- **Zeitintervall:** $I = [\overset{Max}{i=1..n}(I_i min), \overset{Max}{i=1..n}(I_i max)]$
- **Parallelität:** $N = \overset{Min}{i=1..n}(N_i)$
- **Markendurchsatz:** $K = \overset{Min}{i=1..n}(K_i)$

D.3.4. Alternative Prozesse

Bei alternativen Prozessen ist die Entscheidung über den verwendeten Weg unabhängig von äußeren Einflüssen. Stattdessen existieren mehrfache Ressourcen, die das selbe Ergebnis liefern und durch die Alternative modelliert werden. Alternative Flüsse sollten Zusammenführungen besitzen. Durch die alternativen Flüsse addieren sich die Markenflüsse, während sich das Zeitintervall aus der Vereinigung der Teilintervalle ergibt. Bei alternativen Prozessen besteht die Gefahr, dass sich aufeinander folgende Eingangsevents in ihrer Wirkung überholen. Dies ist der Fall, wenn die Bedingung $J_i + J_{in} \leq D_{in}$ verletzt ist.

- **Zeitintervall:** $I = [\overset{Min}{i=1..n}(I_i min), \overset{Max}{i=1..n}(I_i max)]$
- **Parallelität:** $N = \overset{\sum}{i=1..n}(N_i)$
  bzw. $N = Max(I) * K$
- **Markendurchsatz:** $K = \overset{Max}{i=1..n}(K_i * i)$ für die sortierte Menge M mit $M = \{K_1, K_2..K_n\}$ und $K_1 \leq K_2 \leq .. \leq K_n$.

D.4. Synchronisationsbeziehungen von Prozessen

Das Verhalten der Teilsysteme kann auf Basis der Ereignisobjekte als qualitative und quantitative Relation zwischen Ereignissen beschrieben werden. Für zeitintervallbasierte Relationen lassen sich die in Tabelle D.1 aufgeführten Relationen angeben [AP96, Lic04]. Zusätzlich ist notiert, ob diese Relationen auch für Elementarereignisse relevant sind. Beschreibt man zwei Aktionen (Prozesse) jeweils durch Start- und Endereignis, so lassen sich sämtliche Zeitintervallrelationen durch Relationen dieser Ereignisse formulieren.

Voraussetzung in einer solchen paarweisen Prozessbeschreibung ist das Finden geeigneter Paare bei wiederholter Aktivierung von Prozessen. Dazu ist eine Indexierung der Ereignisse gemäß ihrem chronologischen Auftreten erforderlich. Alternativ können Ereignisse automatisch

zugeordnet werden, falls eine Überlappung verschiedener Indexklassen ausgeschlossen ist. Da bislang keine geeigneten Beschreibungsmittel bekannt sind, die eine Indexierung von Ereignissen erlauben, soll eine automatische Paarung vorausgesetzt werden.[75]

Relation	Für Elementar-ereignisse?	Beschreibung als Multiereignis [76]
before	Ja	$(Pa.Ec < PbEs)$
equal	Ja	$(Pa.Es = Pb.Es) \wedge$ (Pa.Ec=Pb.Ec)
meets	Ja	$(Pa.Ec = Pb.Es)$
operlaps	Nein	$(Pa.Es < Pb.Es) \wedge (Pa.Ec > Pb.Es) \wedge (Pa.Ec < Pb.Ec)$
During	Nein	$(Pa.Es > Pb.Es) \wedge (Pa.Ec < Pb.Ec)$
Starts	Nein	$(Pa.Es = Pb.Es)$
Finishes	Nein	$(Pa.Ec = Pb.Ec)$
Starts before	Nein	$(Pa.Es < Pb.Es)$
Finishes after	Nein	$(Pa.Ec > Pb.Ec)$
Disjunct	Ja	$(Pa.Ec < Pb.Es) \vee (Pa.Es > Pb.Ec)$

Tabelle D.1.: Synchronisationsbeziehungen von Prozessen

D.5. Bewertung der Methodik

Wie bereits festgestellt wurde, ist zur Dekomposition von intern nebenläufigen Prozessen (Transitionen) ein erweitertes Modell notwendig. Das so erhaltene Modell ist äquivalent zu erweiterten Prozessmodellen der Schedulinganalyse. Dementsprechend können Regeln zur Komposition bzw. Dekomposition von selbst nebenläufigen Prozessen äquivalent zu Prozessen definiert werden.

[75] Mittels Indexierung könnte sichergestellt werden, dass nur Events mit demselben Index verglichen werden. Lagert man die Logik zum Vergleichen der Events aus, wäre dort ein lokales Zählen der Ereignisse möglich. In jedem Fall können durch Ersatzkonstrukte weitere Einschränkungen entstehen.

[76] Es gilt folgende Leseweise: $P_A.Es$ = *Prozess A, Startevent*; P_B = *Prozess B*; Ec = *Endevent*. Für jeden Prozess x gilt $P_x.Es \leq P_x.Ec$.

E. Zeitanalyse in nebenläufigen Pfaden - Hierarchische Analyse des Zustandsraums

Bislang werden für die Bewertung von Eigenschaften zusätzliche Beobachtungsautomaten (Monitore) erzeugt, die Events konsumieren und Zeiten messen. Strukturelle Informationen aus dem analysierten Modell werden dazu nicht genutzt, sondern nur das temporale Verhalten der Automaten. Treten Nebenläufigkeiten zwischen Events auf, so dass sich Events desselben Typs überholen können, so ist die Zuordnung nicht eineindeutig. Zeitbewertete Petri-Netze (Duration Interval Petri Nets) erlauben keine nebenläufigen Aktivierungen derselben Transition [PZS99], sind also ebenfalls nicht geeignet.
Bislang genutzte Analysewerkzeuge sind nicht in der Lage strukturelle Informationen auszuwerten, da dafür eine Semantik definiert sein muss, mit der Annahmen über die Zusammenhänge zwischen unterschiedlichen Events getroffen werden können. In der betrachteten Klasse von Discrete-Event Modellen können solche Annahmen aus der Art und Weise der Informationsverarbeitung getroffen werden. Ziel dieses Abschnittes ist es, die Techniken des Model Checking für Discrete-Event Modelle so zu erweitern, dass nebenläufige Events abgebildet werden können und die Analyse zeitlicher Abstände in der Zustandsraumdarstellung ohne zusätzliche Zeitvariablen möglich ist.
Betrachtet wird eine Darstellung der Zustände eines Transitionssystems im Regionengraphen. Das Zeitverhalten eines Zustandes wird dabei als konvexe Region im Regionengraphen modelliert. Aufeinander folgende Zustände im Transitionssystem führen dabei zu benachbarten, überlappenden oder überdeckenden Regionen, sofern Zeitvariablen nicht zurückgesetzt werden. Um nun eine Aussage über den Zeitverlauf machen zu können, wird die Lage der Regionen zueinander charakterisiert [CY92]. Dabei wird der Abstand der Minima und Maxima benachbarter Regionen betrachtet. Für einen Transitionspfad lässt sich, durch eine Analyse der entsprechenden Regionen im Regionengraphen, der Zeitbedarf ermitteln.
Bei gewöhnlichen zustandsbasierenden Simulatoren wird aus dem aktuellen Zustand der Folgezustand berechnet und anschließend ein Zeitschritt durchgeführt. Die Berechnung des Folgezustandes kann dabei deterministisch oder nichtdeterministisch erfolgen und sich ggf. über mehrere interne Zyklen (Deltazyklen) erstrecken. Die Modelle sind dabei in der Regel so auf-

gebaut, dass sie den verwendeten Systemaspekt sehr exakt modellieren, die Modelle liegen in Form von RTL, HDL oder (abstrakten) Automatenbeschreibungen auf Instanzebene vor. Parallelität auf funktionaler Ebene wird - durch die Instanzen - explizit modelliert.
Bei Netzwerkmodellen sowie Transaktionsmodellen wird die Parallelität meist nicht explizit modelliert, da sie bereits im Simulationsmodell enthalten ist. Dieses DE-Systemmodell trennt funktionale Einheiten zunächst voneinander und stellt die Verknüpfung über Events und entsprechende Eventscheduler her. Funktionale Einheiten können demzufolge innerhalb eines Zeitintervalls I mit $|I| > 0$ beliebig oft genutzt (aktiviert) werden. Dadurch wird eine explizite Modellierung von Nebenläufigkeiten unnötig. Innerhalb eines Zeitschrittes werden endlich viele Simulationsschritte ausgeführt. Im Folgenden wird das DE-Systemmodell formal definiert, wobei die Datenstrukturen in den Events nicht weiter betrachtet werden[77].

E.1. Definition formales DE-Modell

Das DE-Systemmodell besteht aus einer Eventwarteschlange, zur Verwaltung von Events, und aus funktionalen Einheiten, die jeweils ein endliches Transitionssystem darstellen. Die Kopplung beider Systemteile wird durch Events realisiert, die dem Transport von Daten dienen. Die Verwaltung der Eventwarteschlange kann in Form einer Liste modelliert werden[78]. Im einfachsten Fall werden die Events in der Reihenfolge ihres Zeitstempels abgelegt und in aufsteigender Reigenfolge konsumiert. Neue Events dürfen daher nicht jünger sein als das erzeugende Event.

$$L = (p_1, p_2, .., p_n) \mid t(p_i) <= t(p_{i+1}) i = 1..(n-1)$$

Die funktionalen Einheiten können als Menge von Zustandsautomaten durch Transitionsfunktion definiert werden, die den Folgezustand in Abhängigkeit von den Eingangsereignissen und internen Zuständen beschreiben.

$$\vec{z'} = f_z(\vec{x}, \vec{z})$$

Zusätzlich liefert eine Eventfunktion die Ausgabeevents (ebenfalls abhängig von Zustand und Eingangsevents).

$$L_e = f_L(\vec{x}, \vec{z})$$

Zeitabhängige Zustandsübergänge können allgemein durch zusätzliche, rückgekoppelte Ereignisse modelliert werden. Im Spezialfall ist eine Rücksetzmöglichkeit, also die Beeinflussung der Ereigniswarteschlange notwendig. (Löschen von Ereignissen eines Typs).

[77] Die Betrachtung von Datenstrukturen stellt eine Erweiterung dar, die zusammen mit einer konkreten Realisierung betrachtet werden muss.

[78] Die Beachtung von Einschränkungen durch einen Eventscheduler sowie mögliche Implementierungsdetails werden nicht weiter betrachtet, da sich daraus ggf. Einschränkungen bezüglich der Allgemeinheit ergeben.

E.2. Simulation des formalen DE-Modells

Der Simulationsablauf kann in fünf Schritte eingeteilt werden, die sich bis zur Abbruchbedingung wiederholen.

1. Ermittlung des nächsten Events. Dabei wird das nächste Event aus der Eventliste entfernt. Ist die Eventliste leer, so wird an dieser Stelle die Simulation abgebrochen. Die Länge des Vektors $\vec{x}$ entspricht der Menge unterscheidbarer Eventtypen für die Eingabe. Ein Eventtyp wird dabei mit einem Eingabe- oder Ausgabeelement einer Funktionsinstanz eindeutig identifiziert ($\vec{x} = pop(L)$). Dabei ist nur ein Element des Vektors $\vec{x}$ besetzt, es wird also je Simulationsschritt nur ein funktionales Element angesprochen: $\vec{x}^T = (0, .., x_i, .., 0) | i = 1..n$

2. Funktionale Berechnung. Bei der funktionalen Berechnung werden aus einem Eingabevektor und dem Zustandsvektor die Liste der emittierten Events und der Vektor der Folgezustände berechnet.

$$\left[L_e, \vec{z'}\right] = f(\vec{x}, \vec{z})$$

   Der Zustandsvektor stimmt dabei mit der Menge der Instanzen funktionaler Einheiten überein. Je Simulationsschritt wird maximal ein Element des Zustandsvektors verändert. Die Länge der Liste L_e entspricht der Menge der emittierten Events. Im Spezialfall ist die Liste L_e leer. Die Berechnung lässt sich auch in zwei unabhängige Teiloperationen zerlegen:

$$\vec{z'} = f_z(\vec{x}, \vec{z})$$

$$L_e = f_L(\vec{x}, \vec{z})$$

   Um eine Verzweigung von Events erreichen zu können, müssen entsprechende Verteilungsblöcke (Fork) im Funktionsvektor ergänzt werden, die Events mit dem Intervall $[0, 0]$ emittieren. Dies ist notwendig, da Events mit einem Intervall versehen sind und weil daher das gleichzeitige Schalten dieser Events sonst nicht sichergestellt werden kann.

3. Aktualisierung der Eventliste. Bei der Aktualisierung der Eventliste werden die emittierten Events nach $t(p)$ sortiert in die Eventliste eingetragen.

$$L' = push(L, L_e)$$

4. Iterationszuweisung für Eventliste und Zustandsvektor:

$$\vec{z} := \vec{z'}$$

$$L := L'$$

Der Algorithmus lässt sich in Pseudocode wie folgt zusammenfassen:

While(|L|>0) do

1. $\vec{x} = pop(L)$
2. $\vec{z'} = f_z(\vec{x}, \vec{z})$
   $\vec{L_e} = f_L(\vec{x}, \vec{z})$
3. $L' = push(L, L_e)$
4. $\vec{z} := \vec{z'}$
   $L := L'$

End While

E.2.1. Simulationseigenschaften und Nichtdeterminismus

Die funktionale Berechnung kann, wie beschrieben, in zwei Teile untergliedert werden. Dabei zeigt sich, dass Ausgabeevents zustandsabhängig sein können, sowohl in der Erzeugung an sich als auch in den verbundenen Datenwerten. Es besteht also die Möglichkeit, zeitinvariante und zeitvariante Systeme zu beschreiben. Ebenso ist die Modellierung hochgradig variabler Modellstrukturen möglich. Somit ist die alleinige Betrachtung der äußeren Modellstruktur (Vernetzung der Funktionalen Einheiten) nicht ausreichend zur Bewertung von Abhängigkeiten.
Die Ereignisliste enthält potentiell eine Liste gleichwertiger (zeitgleicher) Ereignisse. Das Simulationsergebnis kann nun (durch die Zustandsabhängigkeit der funktionalen Berechnung) von der Reihenfolge dieser gleichwertigen Events abhängen.
Durch die Eventliste können innerhalb eines Zeitschrittes mehrere Zustandskonfigurationen (pro funktionalem Element) auftreten. Für eine Zeitbewertung ist also eine Betrachtung transienter und stabiler Zustände notwendig. Um implizite Deltazyklen zu vermeiden, wird vereinbart, dass innerhalb der Zustandsfunktion nur stabile Konfigurationen berechnet werden.

E.2.2. Modelleigenschaften

Der Zustandsraum des DE Systemmodells lässt sich in den statischen Teil, der den Funktionseinheiten zugeordnet ist, und den dynamischen Teil, der den (zukünftigen) Transaktionen (Ereignissen) zugeordnet ist, unterteilen.
Ein Vorteil ist darin zu sehen, dass der (potentielle) Grad der Parallelität innerhalb einer Funktionseinheit nicht mehr explizit festgelegt werden muss, sondern sich dynamisch anpasst. Weiterhin lassen sich nun die Auswirkungen von Ereignissen bezogen auf eine Ereignisinstanz analysieren und bewerten, wodurch auch die Analyse verschränkt auftretender Ereignisse möglich wird.

Weiterhin werden Systeme auf der Basis von Greybox-Eigenschaften beschrieben, so dass eine gerade für frühe Phasen der Systemmodellierung geeignete Darstellung erfolgt.

E.3. Abbildung anderer Systemklassen

Transformatorische Modelle lassen sich abbilden, indem der Zustandsvektor konstant bleibt, also Funktionseinheiten keine internen Systemzustände aufweisen.
Systeme von gekoppelten Zustandsautomaten (Clocked Systeme) können abgebildet werden, indem die Länge der Ereigniswarteschlange (in stabilen Zuständen) konstant ist und die Events jeweils den selben Zeitstempel aufweisen. Im Spezialfall ist die Länge der Ereigniswarteschlange gleich der Anzahl der Eingänge der Subsysteme.
Vernachlässigt man die Mappingfunktionen der Transitionen, so ergibt sich ein konventionelles Transitionsmodell. Sind nur die internen Zustände interessant, so kann weiterhin auch der dynamische Teil (Eventliste) abstrahiert werden.

E.4. Analysemethoden für das des DE-Systemmodell

Unter Berücksichtigung des Nichtdeterminismus lässt sich ein Transitionsmodell aufstellen, bei dem der globale Systemzustand Z aus dem Zustandsvektor der Funktionseinheiten $\vec{z}$ und der Ereigniswarteschlange L gebildet wird $Z = (\vec{z}, L)$. Um dabei für kontinuierlich arbeitende Systeme ein endliches Transitionssystem erreichen zu können, müssen die Zeitstempel der Ereignisse relativ abgelegt werden, so dass eine Darstellung mittels DBM(Clock Zones) [Dil90, ACD+92] möglich ist. Dabei wird eine Intervallnotation wie bei Intervall Petri-Netzen [DHFF04] vorgeschlagen, bei der die Restlaufzeit bis zur Konsumierung des Intervalls notiert wird.
Die Transition zwischen zwei Zuständen wird dann mit dem konsumierten Event sowie dem Zeitintervall zur vorherigen Transition notiert und bewirkt eine Änderung im Ereignisvektor sowie ggf. im Zustandsvektor $Z \xrightarrow{\vec{x}} Z'$). Im Ereignisvektor wird das konsumierte Event entfernt und die ggf. emittierten Events der Eventfunktion hinzugefügt. Die Änderung im Zustandsvektor wird durch die Zustandsfunktion bestimmt. Somit lässt sich der Zustandsraum für das definierte System erzeugen.
Um nun eine Bewertung der Auswirkungen durch die Transaktion vornehmen zu können, wird eine Mappingfunktion λ definiert, welche die Abbildung der Events aus der Eventliste des Vorzustandes L auf die Events in der Eventliste des Folgezustandes L' ermöglicht $\lambda : L \rightarrow (L' \cup \vec{x})$. Ein Event des Vorzustandes wird dabei auf das Transitionsevent gemappt, während das Transitionsevent auf die ggf. neu hinzugefügten Events der Eventliste des Folgezustandes gemappt werden. Die Umkehrfunktion von *lambda* ist definiert als $\lambda^{-1} : L' \rightarrow L$, wobei neu hinzugefügte Elemente L_e auf das konsumierte Event verweisen. Dadurch wird ein hierarchisches Transitionssystem erzeugt, bei dem als Verfeinerung des Zustandsraumes strukturelle Abhängigkeiten

der Events notiert werden.
Für die Untersuchung von Eigenschaften auf der Basis von Transaktionen können nun Pfade innerhalb des Transitionssystems dienen. Dazu kann die Suche in zwei Phasen eingeteilt werden. Zunächst erfolgt eine Analyse des Transitionssystems ohne strukturelle Informationen. Dabei wird untersucht werden, ob Pfade im System existieren, die die gewünschten temporalen Eigenschaften aufweisen. Etwa ist es möglich diejenigen Pfade zu suchen, bei denen nach Auftreten des Ereignisses Request (e_{req}) ein Ereignis Acknowledge (e_{ack}) folgt: $e_{req} \rightarrow Fe_{ack}$ dabei können auch die Zeiten der Transitionen berücksichtigt werden: $e_{req} \rightarrow F_{[t_{min},t_{max}]}e_{ack}$. Die gefundenen Pfade erlauben keine Aussage über strukturelle Zusammenhänge zwischen e_{req} und e_{ack}, zudem stellen die betrachteten Zeiten, aufgrund der Lokalität der Zeitvariablen, konservative Abschätzungen dar [CY92].

Im zweiten Schritt erfolgt die Detailanalyse der ermittelten Pfade. Dabei werden die durch die Mappingfunktion λ (λ^{-1}) beschriebenen Zusammenhänge wie folgt berücksichtigt:

- Strukturelle Zusammengänge: Ein struktureller Zusammenhang zwischen zwei Ereignissen eines Pfades existiert genau dann, wenn zwischen Anfangsereignis und Endereignis ein detaillierter Pfad existiert, deren Abschnitte durch Mappingfunktionen beschrieben werden.

- Das Zeitintervall eines detaillierten Pfades ergibt sich aus der Summe der Zeitintervalle neu hinzugefügter Ereignisse die auf dem betrachteten Pfad liegen. Dies entspricht der strukturellen Analyse zugrunde liegenden Beschreibung, siehe [PZH96, FGM98].

E.5. Bewertung der Analysemethodik

Das vorgeschlagene Model vermeidet Beschränkungen bezüglich der Nebenläufigkeit, wie sie bei herkömmlichen zeitbehafteten Modellen, etwa Timed Automata und zeitbehafteten Petri-Netzen, bestehen. Insbesondere ist es möglich, dass mehrere Ereignisse (Transitionen) desselben Typs nebenläufig zu sich selbst aktiv sein können. Dadurch ist es möglich Systeme, bei denen interne zeitbehaftete Prozesse durch sequentielle oder parallele Strukturen zu sich selbst nebenläufig sind, einfacher zu modellieren, indem eine implizite Nebenläufigkeit modelliert wird.

Zudem ist es möglich temporale Eigenschaften und strukturelle Eigenschaften im selben mathematischen Modell abzulegen. Dies ermöglicht die gemeinschaftliche Analyse temporaler und struktureller Eigenschaften. Zudem ist es möglich Zeiten von Pfaden exakter bestimmen zu können.

Bisher wurde das Modell in Form erweiterter zeitbehaftete Petri-Netze realisiert, die zu sich selbst nebenläufige Transitionen enthalten. Der Vektor $\vec{z}$ wird dabei durch die Markierung der Plätze repräsentiert, während die Liste der Events L durch die Menge der aktiven

Transitionen beschrieben wird.[79] Innerhalb des implementierten Modells wurde demonstriert, wie die zeitliche Analyse von Pfaden im hierarchischen Transitionssystem möglich ist. Hybride temporal-strukturelle Analysetechniken wurden dabei noch nicht realisiert. Ebenfalls offen ist bislang die Analyse von Detailpfaden, die im abstrakten Graphen Schleifen darstellen.

[79] Als Spezialfall werden durch die Änderungen des Zustandes (Markierung im Petri-Netz) Events aus der Liste entfernt (Transitionen deaktiviert), was jedoch den dargestellten Prinzipien nicht widerspricht.

Abbildungsverzeichnis

Abbildung 1.1: Kombination von Zustandsräumen nebenläufiger Komponenten . . 3
Abbildung 2.1: Entwicklungsprozess im Systems Engineering nach [Was05] 9
Abbildung 2.2: Schnittstellen-getriebene Spezifikation von Komponenten 14
Abbildung 2.3: Verhaltensmodelle, Spezifikationssprachen und Basiskonzepte 25
Abbildung 2.4: Prozesszeiten nach [Hüs94] 29
Abbildung 2.5: Diagramme der UML 2.0 nach [RHQZ05] 30
Abbildung 3.1: Transformation-basierte Verifikation 47
Abbildung 3.2: Konzept zur Assertion-basierten Prüfung in heterogenen Modellen . 49
Abbildung 4.1: Einfaches Prozessmodell 56
Abbildung 4.2: Strukturelle Komposition 58
Abbildung 4.3: Hierarchische Komposition 58
Abbildung 4.4: MLDesigner Domänen nach [MLD07] 65
Abbildung 4.5: Hierarchische Ordnung von Modellierungsdomänen nach [VGE05] . 66
Abbildung 4.6: Zeit- und Signalmodell für XFLTL 69
Abbildung 4.7: Widening und Narrowing von Intervallen 71
Abbildung 4.8: Event basiertes Assertion Checking 72
Abbildung 5.1: Struktur der Transformation-basierten Verifikation 80
Abbildung 5.2: Vorgehensweise zur Transformation-basierten Verifikation 82
Abbildung 5.3: Hierarchische Eigenschaften 83
Abbildung 5.4: Grundaufbau Timed AR-Automata 99
Abbildung 5.5: Architektur des Prototyps zur Transformation-basierten Verifikation 102
Abbildung 5.6: Templates für die Transformation 104
Abbildung 5.7: Aktivitätsdiagramm „Fertigen“ 106
Abbildung 5.8: Sequenzdiagramm „Teil übernehmen“ 107
Abbildung 5.9: Sequenzdiagramm „Rohteil bearbeiten“ 107
Abbildung 5.10: Zustandsdiagramm „Bearbeitungsgreifer“ 108
Abbildung 5.11: Teilprozess „Fertigen“ und Umgebungsmodell 109
Abbildung 5.12: DE Modelle „Teil übernehmen“ und „Rohteil bearbeiten“ 110
Abbildung 5.13: Die Funktion „Bearbeiten“ 111
Abbildung 5.14: Basistemplates für das Beispiel 112

Abbildung 5.15: Synchronisation 112
Abbildung 5.16: Prüfautomat 113
Abbildung 6.1: Signalabhängige Partitionierung von Intervallen 120
Abbildung 6.2: Verschiebung der Intervalle von temporalen Operatoren 121
Abbildung 6.3: Timed Automat für den F-Operator 124
Abbildung 6.4: Timed Automat für den G-Operator 124
Abbildung 6.5: Beispiel zur Normalisierung von Formeln 129
Abbildung 6.6: Beispiel zur adaptiven Normalisierung von Formeln 130
Abbildung 6.7: Konstruktion eines Timed AR-Automata am Beispiel 134
Abbildung 6.8: Klassendiagramm der XFLTL-Bibliothek 139
Abbildung 6.9: Screenshot: Statusanzeige der Eigenschaften 140
Abbildung 6.10: Σ/Δ - Konverter. 141
Abbildung 6.11: MLD-Modell des Σ/Δ-Konverters 142
Abbildung 6.12: Event-bezogene Diskretisierung der analogen Signale 143
Abbildung 6.13: Signalverlauf der Beobachtungspunkte am Beispiel 146
Abbildung B.1: Screenshot MLD2Uppaal - Eingabe 175
Abbildung B.2: Screenshot MLD2Uppaal - Transformation 176
Abbildung B.3: Screenshot MLD2Uppaal - Ausgabe 177
Abbildung C.1: Beispielmodell für den Assertion Checker. 179
Abbildung C.2: Parameter des „SFormula1“ Primitivs. 180
Abbildung C.3: Statistik zum Assertion Checking 181
Abbildung C.4: Trace der Eigenschaftsbewertungen 181
Abbildung D.1: Sequenzielle Ersetzung 188
Abbildung D.2: Parallele Ersetzung 188

Tabellenverzeichnis

Tab. 2.1: Klassifikationsmöglichkeiten für Temporale Logiken nach [Kro99] 32
Tab. 2.2: Templates für Temporale Logiken nach [Hol03] 34
Tab. 3.1: Umsetzung von Performancemodellen 44
Tab. 4.1: Übersicht über die Domänen 64
Tab. 4.2: Semantik von Modellierungswerkzeugen 64
Tab. 4.3: Spezifikation von Eventfluss als Ursache-Wirkung 76
Tab. 5.1: Vergleich der Strukturmerkmale von UML- und MLDesigner-Modellen . 84
Tab. 5.2: Vergleich von Annotationsmöglichkeiten 90
Tab. 5.3: Vergleich von Paradigmen zur Abbildung von DE-Modellen 93
Tab. 5.4: Mögliche Erreichbarkeitsklassen von Timed AR-Automata 100
Tab. 5.5: Fehlersuche im Modell 105
Tab. 6.1: Logische Operatoren für dreiwertige Logik nach [RHKR00] 118
Tab. 6.2: Intervallbedingungen für die Verknüpfung temporaler Operatoren 133
Tab. 6.3: Ergebnisse der Verifikation 147
Tab. 7.1: Bewertung und Gegenüberstellung der Verifikationsmethoden 153
Tab. A.1: Struktur der Constraint Language 172
Tab. A.2: Assertion Specification Language (XFLTL) 174
Tab. D.1: Synchronisationsbeziehungen von Prozessen 191

Literaturverzeichnis

[ABL96] Abrial, Jean-Raymond (Hrsg.) ; Börger, Egon (Hrsg.) ; Langmaak, Hans (Hrsg.): *Formal methods for industrial applications.* Berlin [u.a.] : Springer, 1996 (Lecture notes in computer science 1165). – ISBN 3540619291

[Abr96] Abrial, Jean-Raymond: *The B-book.* Cambridge [u.a.] : Cambridge Univ. Press, 1996. – ISBN 0521496195

[Acc04] Accellera ; Accellera (Hrsg.): *Property Specification Language (PSL), Version 1.1.* Accellera, June 2004. `http://www.eda.org/vfv`

[ACD+92] Alur, Rajeev ; Courcoubetis, Costas ; Dill, David L. ; Halbwachs, Nicolas ; Wong-Toi, Howard: An Implementation of Three Algorithms for Timing Verification Based on Automata Emptiness. In: *IEEE Real-Time Systems Symposium*, IEEE Computer Society Press, 1992, S. 157–166

[ACD93] Alur, Rajeev ; Courcoubetis, Costas ; Dill, David: Model-Checking in Dense Real-time. In: *Information and Computation* 104 (1993), S. 2–34. – ISSN 0890–5401

[AD94] Alur, Rajeev ; Dill, David L.: A theory of timed automata. In: *Theor. Comput. Sci.* 126 (1994), Nr. 2, S. 183–235. – ISSN 0304–3975

[AFH96] Alur, Rajeev ; Feder, Tomás ; Henzinger, Thomas A.: The benefits of relaxing punctuality. In: *J. ACM* 43 (1996), Nr. 1, S. 116–146. – ISSN 0004–5411

[AP96] Alferes, José J. ; Pereira, Luís M.: *Reasoning with logic programming.* Berlin [u.a.] : Springer, 1996 (Lecture notes in computer science 1111). – ISBN 3540614885

[Bau90] Baumgarten, Bernd: *Petri-Netze.* Mannheim [u.a.] : BI-Wiss.-Verl., 1990. – ISBN 341114291X

[BCNN01] Banks, Jerry ; Carson, John S. ; Nelson, Barry L. ; Nicol, David M.: *Discrete-Event System Simulation.* 3. Prentice Hall, 2001. – ISBN 0–13–088702–1

[BDL04] Behrmann, G. ; David, A. ; Larsen, Prof. K.: A Tutorial on UPPAAL. In: Bernardo, M. (Hrsg.) ; Corradini, F. (Hrsg.): *LNCS, Formal Methods for the*

Design of Real-Time Systems (revised lectures) Bd. 3185, Springer Verlag, 2004. – ISBN 3–540–23068–8, S. 200–237

[Ben05] Bender, Klaus: *Embedded Systems - qualitätsorientierte Entwicklung.* Berlin, Heidelberg : Springer-Verlag Berlin Heidelberg, 2005. – ISBN 9783540273707

[Ber06] Berthomieu, Bernard ; LAAS-CNRS, france (Hrsg.): *Official Homepage of the TINA Toolbox.* LAAS-CNRS, france, 2006. – `http://www.laas.fr/tina`

[Bey02] Beyer, Dirk: *Formale Verifikation von Realzeit-Systemen mittels Cottbus-Timed-Automata.* 2002. – PhD Thesis, TU Cottbus, in German language

[BHS07] Baumann, Tommy ; Hauguth, Maik ; Salzwedel, Horst: Overcoming the Gap between Design at Electronic System Level (ESL) and Implementation for Networked Electronics. In: *Western MultiConference on Modeling & Simulation, WMC '07, 14.-18. January 2007, San Diego, California* (2007), 1

[Bro07] Broy, Manfred: From "Formal Methods" to System Modeling. In: Jones, Cliff B. (Hrsg.) ; Liu, Zhiming (Hrsg.) ; Woodcock, Jim (Hrsg.): *Formal Methods and Hybrid Real-Time Systems* Bd. 4700, Springer, 2007 (Lecture Notes in Computer Science). – ISBN 978–3–540–75220–2, S. 24–44

[Buc91] Buchholz, Peter: *Die strukturierte Analyse Markovscher Modelle.* Berlin [u.a.] : Springer, 1991 (Informatik-Fachberichte 282). – ISBN 3540545409

[Bud03] Budinsky, Frank (Hrsg.): *Eclipse modeling framework.* 2. printing. Boston [u.a.] : Addison-Wesley, 2003 (The eclipse series). – ISBN 0131425420

[BY03] Bengtsson, Johan ; Yi, Wang: Timed Automata: Semantics, Algorithms and Tools. In: Desel, Jörg (Hrsg.) ; Reisig, Wolfgang (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Lectures on Concurrency and Petri Nets* Bd. 3098, Springer, 2003 (Lecture Notes in Computer Science). – ISBN 3–540–22261–8, S. 87–124

[CGP99] Clarke, E.M. ; Grumberg, O. ; Peled, D.A.: *Model checking.* The MIT Press, 1999. – ISBN 0–262–03270–8

[CH04] Chaochen, Zhou ; Hansen, Michael R.: *Duration calculus : a formal approach to real-time systems.* Springer, 2004. – ISBN 3–540–40823–1

[Cro07] Crouch, Greg: *Why is Autocode Generation Slow to Adoption?* Version: June 19 2007. `http://www.dspdesignline.com/showArticle.jhtml;?articleID=199905899`, Abruf: 17.06.2009. web article

[CY92] Courcoubetis, Costas ; Yannakakis, Mihalis: Minimum and Maximum Delay Problems in Real-Time Systems. In: *Formal Methods in System Design* 1 (1992), Nr. 4, S. 385–415. – ISSN 0925–9856

[DA05] Davis, R. ; Alla, H.: *Discrete, Continous, and Hybrid Petri Nets.* Springer-Verlag Berlin Heidelberg, 2005. – ISBN 3–540–22480–7

[Dep] Department of Information Technology ; Uppsala University, Sweden (Hrsg.): *Official Homepage of the Uppaal Tool.* Uppsala University, Sweden, `http://www.uppaal.com`, Abruf: 24.03.2009

[DGS05] Demathieu, Sebastien ; Griffin, Catherine ; Sendall, Shane: *Model Transformation with the IBM Model Transformation Framework.* Version: May 2005. `http://www.ibm.com/developerworks/rational/library/05/503_sebas/`, Abruf: 13.07.2009. Web article

[DHFF04] Duridanova, Vesselka ; Hummel, Thorsten ; Fengler, Olga ; Fengler, Wolfgang: Verifikation von Spezifikationsmodellen mit Intervall-Petri-Netzen. In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen 7. GI/ITG/GMM-Workshop zu Modellierung und Verifikation, Kaiserslautern 24.-25.02.2004.* (2004), S. 184–193. ISBN 3–8322–2486–6

[Dil90] Dill, D. L.: Timing assumptions and verification of finite-state concurrent systems. In: *Proceedings of the international workshop on Automatic verification methods for finite state systems.* New York, NY, USA : Springer-Verlag New York, Inc., 1990. – ISBN 0–387–52148–8, S. 197–212

[EHS97] Ellsberger, Jan ; Hogrefe, Dieter ; Sarma, Amardeo: *SDL.* London [u.a.] : Prentice Hall, 1997. – ISBN 0136213847

[Enga] Engineering, ClearSy S.: *Atelier B, the industrial tool to efficiently deploy the B Method.: Homepage of Atelier B.* `http://www.atelierb.eu`, Abruf: 15.06.2009

[Engb] Engineering, ClearSy S.: *Homepage of the B-Method.* `http://www.bmethod.com/`, Abruf: 15.06.2009. Website

[FGM98] Felder, Miguel ; Gargantini, Angelo ; Morzenti, Angelo: A Theory of Implementation and Refinement in Timed Petri Nets. In: *Theor. Comput. Sci.* 202 (1998), Nr. 1-2, S. 127–161. – ISSN 0304–3975

[Fit05] Fitzgerald, John (Hrsg.): *Validated designs for object-oriented systems.* London : Springer, 2005. – ISBN 1852338814

[FKL04] Foster, H.D. ; Krolnik, A.C. ; Lacey, D.J.: *Assertion-Based Design 2nd Edition*. 2. Kluwer Academic Publishers, 2004. – ISBN 1–4020–8027–1

[GL06] Gamerman, Dani ; Lopes, Hedibert F.: *Markov chain Monte Carlo*. 2. ed. Boca Raton, Fla. [u.a.] : Chapman & Hall/CRC, 2006 (Texts in statistical science series 68). – ISBN 9781584885870

[Har87] Harel, David: Statecharts: A visual formalism for complex systems. In: *Sci. Comput. Program.* 8 (1987), Nr. 3, S. 231–274. – ISSN 0167–6423

[Hey79] Heyer, Herbert: *Einführung in die Theorie Markoffscher Prozesse*. Mannheim [u.a.] : BI-Wiss.-Verl., 1979. – ISBN 3411015640

[Hoa78] Hoare, C. A. R.: Communicating sequential processes. In: *Commun. ACM* 21 (1978), Nr. 8, S. 666–677. – ISSN 0001–0782

[Hol03] Holzmann, Gerard J.: *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003. – ISBN 0321228626

[HRS99] Henzinger, Thomas A. ; Raskin, Jean-Francois ; Schobbens, Pierre-Yves: Axioms for Real-Time Logics / Max-Planck-Institut für Informatik. Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, August 1999 (MPI-I-1999-3-005). – Research Report. – ISSN 0946–011X

[Hüs94] Hüsener, Thomas: *Entwurf komplexer Echtzeitsysteme: state of the art*. 1. BI-Wiss.-Verl.,Mannheim [u.a.], 1994. – ISBN 3–411–16441–7

[IBM07] IBM alphaworks: *alphaWorks : Model Transformation Framework : Overview:*. Version: 2007. `http://www.alphaworks.ibm.com/tech/mtf`, Abruf: 13.07.2009. Website

[Jer05] Jersák, Marek: *Compositional Performance Analysisfor Complex Embedded Applications*. 2005. – PhD Thesis, Technical University of Braunschweig

[JH08] Jesser, A. ; Hedrich., L.: A Symbolic Approach for Mixed-Signal Model Checking. In: *The 13th Asia and South Pacific Design Automation Conference (ASP-DAC'08), COEX, Seoul, Korea* (2008), January, S. 404–409

[JLP+07] Jesser, A. ; Lämmermann, S. ; Pacholik, A. ; Weiss, R. ; Ruf, J. ; Fengler, W. ; Hedrich, L. ; Kropf, T. ; Rosenstiel, W.: Analog Simulation Meets Digital Verification - A Formal Assertion Approach for Mixed-Signal Verification. In: *The 14th Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI'07)*, 2007, S. 507–514

[JLP+08] Jesser, A. ; Laemmermann, S. ; Pacholik, A. ; Weiss, R. ; Ruf, J. ; Fengler, W. ; Hedrich, L. ; Kropf, T. ; Rosenstiel, W.: Advanced Assertion-Based Design for Mixed-Signal Verification. In: *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, Special Section on VLSI Design and CAD Algorithms* E91-A (2008), December, Nr. 12

[JS05a] Jantsch, A. ; Sander, I.: Models of computation and languages for embedded system design. In: *IEE Proceedings on Computers and Digital Techniques* 152 (2005), March, S. 114–129

[JS05b] Jantsch, A. ; Sander, I.: Models of computation in the design process. In: *System On Chip: Next Generation Electronics* (2005), S. 161–183. ISBN 0–86341–552–0

[KNP07] Kwiatkowska, M. ; Norman, G. ; Parker, D.: Stochastic Model Checking. In: Bernardo, M. (Hrsg.) ; Hillston, J. (Hrsg.): *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07)* Bd. 4486, Springer, 2007 (LNCS (Tutorial Volume)). – ISSN 0302–9743, S. 220–270

[Kro99] Kropf, Thomas: *Introduction to Formal Hardware Verification.* Springer Verlag, 1999. – ISBN 3–540–65445–3

[Lam00] Lamsweerde, Axel v.: Formal specification: a roadmap. In: *ICSE '00: Proceedings of the Conference on The Future of Software Engineering.* New York, NY, USA : ACM, 2000. – ISBN 1–58113–253–0, S. 147–159

[Lan96] Lano, Kevin ; Haughton, Howard (Hrsg.): *Specification in B.* London : Imperial College Press, 1996. – ISBN 1860940080

[Lic04] Licht, T.: *Ein Verfahren zur zeitlichen Analyse von UML-Modellen beim Entwurf von Automatisierungssystemen.* Isle Verlag, 2004. – ISBN 3–932633–91–1. – in German language

[LK00] Law, Averill M. ; Kelton, W. D.: *Simulation Modeling and Analysis* . 3. McGraw-Hill, 2000. – ISBN 0–07–059292–6

[Löt99] Lötzbeyer, Annette: *Temporale Realzeitverifikation.* PhD Thesis, Universität Karlsruhe, 1999. – ISBN 3–89722–289–2. – in German language

[Mül06] Müller, Marcus: *Transformation von modellbasierten Systembeschreibungen - Entwurf eines Java-Frameworks*, Tu Ilmenau, Diplomarbeit, 07 2006

[MLD07] MLDesign Technologies Inc. ; MLDesign Technologies Inc. (Hrsg.): *ML-Designer Documentation, Version 2.7.* MLDesign Technologies Inc., 2007. – `http://www.mldesigner.com/mldesigner/documents.html`

[MNP08] Maler, Oded ; Nickovic, Dejan ; Pnueli, Amir: Checking Temporal Properties of Discrete, Timed and Continuous Behaviors. In: *Pillars of Computer Science*, 2008. – ISBN 978–3–540–78126–4, 475–505

[MNPB06] Maler, Oded ; Nickovic, Dejan ; Pnueli, Amir ; Bohr, Niels: From MITL to timed automata. In: *In Proceedings of the 4th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS'06), volume 4202 of Lecture Notes in Computer Science*, 2006, S. 274–289

[MP91] Manna, Zohar ; Pnueli, Amir: *The Temporal Logic of Reactive and Concurrent Systems - Specification.* 1. Springer Verlag, 1991. – ISBN 0–387–97664–7

[MPF07] Müller, Marcus ; Pacholik, Alexander ; Fengler, Wolfgang: Tool Support for Formal System Verification. In: Scharff, Peter (Hrsg.) ; Ilmenau (Veranst.): *Computer science meets automation* Bd. II Ilmenau, Univ.-Verl., September 2007 (52. IWK, Internationales Wissenschaftliches Kolloquium, 10 - 13 September 2007.). – ISBN 978–3–939473–17–6, S. 137–142

[Nat04] National Instuments ; National Instuments (Hrsg.): *LabVIEW Benutzerhandbuch, Version Juni 2003, Artikel 321200E-01.* National Instuments, June 2004. – `http://www.ni.com/pdf/manuals/321200e.pdf`

[NMP+06] Nickovic, Dejan ; Maler, Oded ; Pnueli, Amir ; Caspi, Paul ; Girard, Antoine: Final Proposal fpr PSL Analog Extensions - Deliverable 1.3/2 of the PROSYD Project / Verimag, PROSYD. 2006 (1.3/2). – Techreport/Delivarable

[NPW02] Nipkow, Tobias ; Paulson, Lawrence C. ; Wenzel, Markus: *Isabelle/HOL.* Berlin [u.a.] : Springer, 2002 (Lecture notes in computer science 2283). – ISBN 3540433767

[Nüt99] Nützel, Jürgen: *Objektorientierter Entwurf verteilter eingebetteter Echtzeitsysteme auf Basis höherer Petrinetze.* PhD Thesis, TU Ilmenau, 1999. – in German language

[OH04] Oh, Hyunok ; Ha, Soonhoi: Fractional Rate Dataflow Model for Efficient Code Synthesis. In: *J. VLSI Signal Process. Syst.* 37 (2004), Nr. 1, S. 41–51. – ISSN 0922–5773

[OMG04] OMG ; Object Management Group (OMG) (Hrsg.): *UML 2.0 Superstructure. OMG Final Adopted Specification. Ptc/04-10-02.* Object Management Group (OMG), June 2004. – `http://www.omg.org`

[OS06] Ouaknine, Joël ; Schneider, Steve: Timed CSP: A Retrospective. In: *Electr. Notes Theor. Comput. Sci.* 162 (2006), S. 273–276

[OSC07] OSCI ; Open SystemC Initiative (OSCI) (Hrsg.): *SystemC 2.2 - Core SystemC Language and Examples, Release 2.2.* V2.2. Open SystemC Initiative (OSCI), March 2007. – `http://www.systemc.org/downloads/standards`

[OSC08] OSCI: *SystemC AMS Extensions Draft 1.* December 2008. – `http://www.systemc.org`

[Oua02] Ouaknine, Joël: Digitisation and Full Abstraction for Dense-Time Model Checking. In: Katoen, Joost-Pieter (Hrsg.) ; Stevens, Perdita (Hrsg.): *TACAS* Bd. 2280, Springer, 2002 (Lecture Notes in Computer Science). – ISBN 3–540–43419–4, S. 37–51

[Oxf] Oxford University Computing Laboratory: *PRISM - Probabilistic Symbolic Model Checker.* `http://www.prismmodelchecker.org/`, Abruf: 21. 06. 2009. – Website of the PRISM Tool.

[Par05] Partners, SysML ; SysML Partners (Hrsg.): *SysML Partners: SysML Specification v. 1.0a.* SysML Partners, November 2005. – `http://www.sysml.org`

[PF07] Pacholik, Alexander ; Fengler, Wolfgang: A System Model for Formal Verification of TLM based Transaction Properties. In: Riley, George (Hrsg.) ; Rajaei, Hassan (Hrsg.) ; Ahmad, Aftab (Hrsg.): *Proceedings of the tenth Communications and Networking Simulation Symposium (CNS'07)* Bd. 39, 2007. – ISBN 1–56555–312–8, 93-99

[PFSV05] Pacholik, A. ; Fengler, W. ; Salzwedel, H. ; Vinogradov, O.: Real Time Constraints in System Level Specifications Improving the Verification Flow of Complex Systems. In: *Net.ObjectDays 2005 - NODe '05, Object Oriented Software Design for Real Time and Embedded Computer Systems - OORE '05, Erfurt, 19.-22. September 2005*, 2005. – ISBN 3–9808628–4–4, 283-294

[PRO] PROSYD ; EU Project Property-Based System Design (Hrsg.): *Official Homepage of the PROSYD Project.* EU Project Property-Based System Design, `http://www.prosyd.org/`, Abruf: 23.03.2009

[PZ89] Popova-Zeugmann, Louchka G.: Zeit-Petri-Netze. (1989). `http://worldcat.org/oclc/75120616`

[PZH96] Popova-Zeugmann, Louchka ; Heiner, Monika: Worst-case Analysis of Concurrent Systems with Duration Interval Petri Nets. In: *BTU Cottbus*, 1996, S. 162–179

[PZS99] Popova-Zeugmann, Louchka ; Schlatter, Dirk: Analyzing Paths in Time Petri Nets. Amsterdam, The Netherlands, The Netherlands : IOS Press, 1999. – ISSN 0169–2968, S. 311–327

[RHKR00] Ruf, J. ; Hoffmann, M. ; Kropf, T. ; Rosenstiel, W.: Simulation Based Validation of FLTL Formulas in Executable System Descriptions. In: *Forum on Design Languages (FDL)*, 2000

[RHQZ05] Rupp, C. ; Hahn, J. ; Queins, S. ; Zengler, B.: *UML 2 glasklar - Praxiswissen für die UML-Modellierung und -Zertifizierung*. 2. Carl Hanser Verlag, 2005. – ISBN 3–446–22952–3

[RK00] Ruf, Jürgen ; Kropf, Thomas: Analyzing real-time systems. In: *DATE '00: Proceedings of the conference on Design, automation and test in Europe*. New York, NY, USA : ACM, 2000. – ISBN 1–58113–244–1, S. 243–249

[RKNP04] Rutten, J. ; Kwiatkowska, M. ; Norman, G. ; Parker, D.: *CRM Monograph Series*. Bd. 23: *Mathematical Techniques for Analyzing Concurrent and Probabilistic Systems,* P. Panangaden and F. van Breugel (eds.). American Mathematical Society, 2004

[RP03] Ruf, J. ; Peranandam, P.: Bounded property checking with symbolic simulation. In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, GI/ITG/GMM Workshop*, Shaker Verlag, 2003, S. 209–218

[RS96] Raskin, Jean-François ; Schobbens, Pierre yves: State Clock Logic: a Decidable Real-Time Logic. In: *HART'97, LNCS 1201*, Springer-Verlag, 1996, S. 33–47

[RS03] Reichardt, Jürgen ; Schwarz, Bernd: *VHDL-Synthese : Entwurf digitaler Schaltungen und Systeme*. 3. München [u.a.] : Oldenbourg, 2003, 2003. – ISBN 3–486–27384–1

[RS04] Rath, Holger ; Salzwedel, Horst: ANSI C code synthesis for MLDesigner finite state machines. In: Sawodny, Oliver (Hrsg.): *Synergies between information processing and automation : 49. Internationales Wissenschaftliches Kolloquium, 27.-30.9.2004* Bd. 2. Shaker Verlag, 2004. – ISBN 3–8322–2824–1, S. 107–112

[Sal04] Salzwedel, H.: Design Technology Development Towards Mission Level Design. In: *49. Internationales Wissenschaftliches Kolloquium IWK'2004, 27.-30. September 2004, Ilmenau* (2004), 9. ISBN 3–8322–2824–1

[Sch01] Schöning, Uwe: *Theoretische Informatik - kurzgefasst.* 4. Aufl. Heidelberg [u.a.] : Spektrum, Akad. Verl., 2001 (Spektrum-Hochschultaschenbuch). – ISBN 3827410991

[Sch03] Schmidt, Volker: Markov-Ketten und Monte-Carlo-Simulation / Universität Ulm Abteilung Stochastik. Ulm, September 2003. – Vorlesungsmanuskript

[SH08] Steinhorst, Sebastian ; Hedrich, Lars: Model checking of analog systems using an analog specification language. In: *DATE '08: Proceedings of the conference on Design, automation and test in Europe.* New York, NY, USA : ACM, 2008. – ISBN 978–3–9810801–3–1, S. 324–329

[Spi89] Spivey, John M.: *The Z notation.* New York u.a. : Prentice Hall, 1989 (Prentice Hall International Series in computer science). – ISBN 013983768X

[Stö04] Störrle, Harald: *Semantics of UML 2.0 Activities with Data-Flow.* 2004

[Sta90] Starke, Peter H.: *Analyse von Petri-Netz-Modellen.* Stuttgart : Teubner, 1990 (@Leitfäden und Monographien der Informatik). – ISBN 3519022443

[Tei97] Teich, Jürgen: *Digitale Hardware/Software-Systeme.* Berlin [u.a.] : Springer, 1997 (Springer-Lehrbuch). – ISBN 3540624333

[The07] The MathWorks: *Matlab 7 - Desktop Tools and Development Environment*, 2007. – `http://www.mathworks.com/`

[Tur04] Turau, Volker: *Algorithmische Graphentheorie.* 2. Oldenbourg, 2004. – ISBN 3–486–20038–0

[VGE05] Vachoux, A. ; Grimm, C. ; Einwich, K.: Extending SystemC to support mixed discrete-continuous system modeling and simulation. In: *Proc. IEEE International Symposium on Circuits and Systems ISCAS 2005*, 2005, S. 5166–5169 Vol. 5

[Was05] Wasson, Charles S.: *System Analysis, Design, and Development - Concepts, Principles, and Practices.* Wiley & Sons, Inc., 2005. – ISBN 0–471–39333–9

[Wei06] Weilkiens, Tim: *Systems Engineering mit SysML/UML.* dpunkt.verlag, 2006. – ISBN 3–89864–409–X

[Wika] Wikimedia Foundation Inc.: *Wikipedia Artikel Markow-Prozess.* `http://de.wikipedia.org/wiki/Markow-Prozess`, Abruf: 1. 02. 2009

[Wikb] Wikimedia Foundation Inc.: *Wikipedia Artikel Maschinengestütztes Beweisen.* `http://de.wikipedia.org/wiki/Maschinengest%C3%BCtztes_Beweisen`, Abruf: 1. August 2009

[Wikc] Wikimedia Foundation Inc.: *Wikipedia Artikel Test-Driven Development*. `http://en.wikipedia.org/wiki/Test-driven_development`, Abruf: 24. März 2009

[Wikd] Wikimedia Foundation Inc.: *Wikipedia Artikel Testgetriebene Entwicklung*. `http://de.wikipedia.org/wiki/Testgetriebene_Entwicklung`, Abruf: 24. März 2009

[WK04] Warmer, J. ; Kleppe, A.: *Object Constraint Language 2.0*. mitp-Verlag Bonn, 2004. – ISBN 3–8266–1445–3. – in german language

[WM91] Ward, Paul T. ; Mellor, Stephen J.: *Strukturierte Systemanalyse von Echtzeit-Systemen*. 1. Hanser, 1991. – ISBN 3–446–16198–8

[WRKR05] Weiss, R.J. ; Ruf, J. ; Kropf, T. ; Rosenstiel, W.: Efficient and Customizable Integration of Temporal Properties into SystemC. In: *Forum on Specification and Design Languages (FDL'05)* (2005), September

[ZJ97] Zave, Pamela ; Jackson, Michael: Four dark corners of requirements engineering. In: *ACM Trans. Softw. Eng. Methodol.* 6 (1997), Nr. 1, S. 1–30. – ISSN 1049–331X