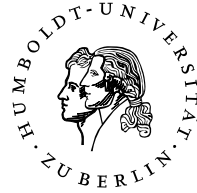


HUMBOLDT-UNIVERSITÄT ZU BERLIN



# SimML – eine SysML-Erweiterung für ausführbare Simulationsmodelle

Diplomarbeit

Humboldt-Universität zu Berlin  
Mathematische-Naturwissenschaftliche Fakultät II  
Institut für Informatik

Lehr- und Forschungseinheit  
Systemanalyse

eingereicht von:  
Peer Hausding

Betreuer:  
Prof. Dr. Joachim Fischer  
Dipl.-Inf. Andreas Blunk

Berlin, den 30. September 2011

1. Gutachter: Prof. Dr. Joachim Fischer
2. Gutachter: Prof. Dr. Jens-Peter Redlich



# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>7</b>
1.1	Problemstellung . . . . .	8
1.2	Zielsetzung . . . . .	8
1.3	Vorgehensweise . . . . .	9
1.4	Notation . . . . .	10
<b>2</b>	<b>Systemmodellierung mit SysML</b>	<b>11</b>
2.1	Systembegriffe . . . . .	11
2.2	Systemdefinition . . . . .	12
2.3	Beschreibung von Systemstruktur . . . . .	13
2.3.1	Wertetyp . . . . .	14
2.3.2	Block . . . . .	14
2.3.3	Werte Verteilung . . . . .	16
2.3.4	Notation . . . . .	16
2.4	Beschreibung von Systemverhalten . . . . .	16
2.4.1	Beschreibung von zeitdiskreten Zustandsänderungen . . .	17
2.4.2	Beschreibung von zeitkontinuierlichen Zustandsänderungen	18
2.4.3	Notation . . . . .	19
2.5	Systemkonfiguration . . . . .	20
2.5.1	Notation . . . . .	20
<b>3</b>	<b>Die Sprache SimML</b>	<b>21</b>
3.1	Namensgebung . . . . .	21
3.2	Strukturbeschreibung . . . . .	22
3.2.1	Paket . . . . .	22
3.2.2	Wertetyp . . . . .	22
3.2.3	Block . . . . .	23
3.2.4	Blockeigenschaft . . . . .	23
3.2.5	Operation . . . . .	25
3.3	Verhaltensbeschreibung . . . . .	25
3.3.1	Zustandsmaschine . . . . .	25
3.3.2	Pseudozustand . . . . .	26
3.3.3	Zustand . . . . .	26
3.3.4	Transition . . . . .	27
3.3.5	Ereignis . . . . .	27
3.3.6	Bedingung . . . . .	28

3.3.7	Effekt . . . . .	28
3.4	Aktionssprache . . . . .	29
3.4.1	Blockteil/-referenz . . . . .	29
3.4.2	Blockwert . . . . .	31
3.4.3	Mehrwertige Blockeigenschaft . . . . .	31
3.5	Konfigurationsbeschreibung . . . . .	33
3.6	Erläuterungen zu SimML . . . . .	34
3.6.1	Namensgebung . . . . .	34
3.6.2	Strukturbeschreibung . . . . .	34
3.6.3	Verhaltensbeschreibung . . . . .	34
3.6.4	Port-Kommunikation . . . . .	35
3.6.5	Zustandsereignis . . . . .	35
<b>4</b>	<b>ODEMx-Erweiterungen</b>	<b>37</b>
4.1	Prozessorientierte Simulationsmodelle in ODEMx . . . . .	37
4.1.1	Warten auf Ereignisse . . . . .	37
4.1.2	Zeitereignisse . . . . .	38
4.1.3	Zustandsereignisse . . . . .	38
4.2	Überblick der ODEMx-Erweiterungen . . . . .	39
4.3	ODEMx-Erweiterungen im Modul: Synchronization . . . . .	39
4.3.1	Zeitdiskrete Zustandsänderung . . . . .	40
4.3.2	Monitor-Konzept . . . . .	43
4.3.3	Erweitertes Monitor-Konzept . . . . .	44
4.4	ODEMx-Erweiterungen im Modul: SimML . . . . .	45
4.4.1	Klasse Block . . . . .	45
4.4.2	Klasse Property . . . . .	46
4.4.3	Klasse ValueType . . . . .	47
4.4.4	Klasse StateMachine . . . . .	47
<b>5</b>	<b>Abbildung eines SimML-Modells nach ODEMx</b>	<b>51</b>
5.1	Abbildungsregeln . . . . .	51
5.2	Validierung . . . . .	51
5.3	Strukturabbildung . . . . .	52
5.3.1	Block-Abbildungsregeln . . . . .	52
5.3.2	Beispiel für die Abbildung von Blöcken . . . . .	57
5.4	Verhaltensabbildung . . . . .	61
5.4.1	Zustandsmaschinen-Abbildungsregeln . . . . .	61
5.4.2	Beispiel für die Abbildung von Zustandsmaschinen . . . . .	65
5.5	Simulationsbeschreibung . . . . .	70
5.5.1	Simulationsbeschreibungsabbildung . . . . .	70
<b>6</b>	<b>Zusammenfassung</b>	<b>73</b>
6.1	Ausblick . . . . .	74
6.1.1	Abbildung kombinierter Systeme . . . . .	74

6.1.2	Zufallszahlen . . . . .	75
6.1.3	Aktionssprache . . . . .	75
6.1.4	Konvertierung von Werten . . . . .	75
6.2	Resultat . . . . .	76
<b>A</b>	<b>MTL-Transformationsregeln</b>	<b>77</b>
<b>B</b>	<b>Anhang</b>	<b>93</b>
B.1	Fachbegriffe . . . . .	94
<b>C</b>	<b>Erklärungen</b>	<b>103</b>



# 1 Einleitung

*„Modelle und Simulationen jeder Art sind Hilfsmittel für den Umgang mit der Realität; sie sind so alt wie die Menschheit selber.“*

(„Systeme, Dynamik, Simulation“, Hartmut Bossel [5])

Für eine Untersuchung eines Systems wird eine Nachbildung benötigt, um Experimente daran durchführen zu können. Im Folgenden werden diese Nachbildungen ausschließlich Modelle sein, die mit einem Computer verarbeitet werden können. Die Experimente erfolgen ebenfalls am Computer in Form einer Simulation (ausschließlich Digitalrechner). Ein Modell ist eine Repräsentation eines realen oder gedachten Systems und beschreibt dieses in Hinblick auf ein konkretes Untersuchungsziel. Es abstrahiert das System und blendet die für das Untersuchungsziel unwesentlichen Elemente, d. h. solche die keine Relevanz auf die Untersuchung haben, aus.

Diese Arbeit beschäftigt sich mit der Systemanalyse, die mit Hilfe des modellbasierten Systems Engineering (MBSE) (siehe „Systems Engineering Vision 2020“ von der International Council on Systems Engineering (INCOSE) [7]) Modelle auf Computern zur Ausführung bringen soll. MBSE wird als die Modellierung eines Systems im Systementwicklungsprozess verstanden und soll die Analyse, die Spezifikation, das Design und die Verifikation und Validierung anhand eines Systemmodells durchgängig unterstützen. Daneben gibt es das dokumentenbasierte Systems Engineering, welches als Basis für die verschiedenen Prozesse eine Menge von textuellen Spezifikationen besitzt.

Die modellbasierte Entwicklung ermöglicht die automatisierte Weiterverarbeitung der erstellten Modelle. Die Modelle des Systems, welche mit Hilfe eines Modell-Repositorys versioniert und verwaltet werden, können in den Entwicklungsphasen auf verschiedene Aspekte untersucht werden. Ziel ist es, solche Modelle in Modelle zu überführen, die mit Hilfe eines Laufzeitsystems auf dem Computer ausgeführt werden können (Simulation).

Eine Systemanalyse ist für einige Systeme bereits auf Grundlage der Modelle durchführbar. Systeme, die sich analytisch nicht lösen lassen, können erst durch eine Simulation hinsichtlich ihrer Untersuchungskriterien bewertet werden. Ausgehend von einem Systemmodell wird mit Hilfe von Konfigurationseinstellungen und der Übergabe von Werten an die äußere Schnittstelle des Systems die Veränderung des Systemzustandes im Verlauf der Zeit einer Simulation beobachtet. Die Zustandsgrößen können sich dabei grundsätzlich zeitkontinuierlich oder -diskret verändern. Werden die Konfigurationseinstellungen und Werte an den äußeren

Schnittstellen verändert, so kann es zu einem anderen Verlauf der Veränderung des Systemzustandes kommen. Der Vergleich der Simulationsexperimente ermöglicht Rückschlüsse auf die Untersuchungskriterien.

## 1.1 Problemstellung

Die Systems Modeling Language (SysML) [13] ist eine von der Object Management Group (OMG) standardisierte Modellierungssprache für die Spezifikation, die Analyse, das Design, und die Verifikation und Validierung von Systemen und deren Systemelemente wie beispielsweise Software, Hardware, Informationen, Prozesse, Personen und Gegenstände. Die Sprache ist eine Erweiterung der Unified Modeling Language (UML) [14], die für die modellgetriebene Softwareentwicklung definiert wurde.

Die mit UML bewährten Eigenschaften wie die objektorientierte Modellierung, die Beschreibung von Verhalten mit UML-Zustandsmaschinen und die grafische Modellrepräsentation, die sich bei der Beschreibung von komplexen Modellen als hilfreich erweisen sollen, sind ebenso Bestandteil der Sprache SysML. Die Sprache bietet darüber hinaus eine an die Domäne des Systems Engineering angepasste Bezeichnung der Modellierungselemente und eine angepasste Semantik für Konzepte, die aus UML übernommen wurden. Zusätzlich besitzt SysML Erweiterungen mit denen sich kombinierte Systeme, die auch von stochastischen Größen abhängen können, beschreiben lassen.

Ausgehend von diesen Systemmodellen ist bisher nicht vollständig untersucht, ob sich diese für die Simulation eines Systems eignen. Die Ausführung von Systemmodellen erfordert eine detaillierte Beschreibung des Systems, wobei untersucht werden muss, ob eine Generierung eines Simulationsprogrammes anhand eines Systemmodells in SysML möglich ist. Prinzipiell lassen sich dabei prozess- und ereignisorientierte und kombinierte Simulationsmodelle unterscheiden.

## 1.2 Zielsetzung

Diese Arbeit soll zeigen, dass ein Modell in SysML sich als Grundlage für die Erweiterung in ein Simulationsmodell verwenden lässt, wenn zusätzlich Informationen, die für die Simulation nötig sind, ergänzt werden. Aufbauend auf dem SysML-Profil wird ein UML-Profil für die Domäne Simulation (Simulations Modeling Language (SimML)) benötigt, welches Konzepte von Simulationssprachen definiert. Modelle in SimML sollen dann in Programme in einer Simulationssprache abbildbar sein.

SysML-Modelle bestehen aus zeitdiskreten und zeitkontinuierlichen Zustandsänderungen. Die Ausführbarkeit von Modellen von zeitkontinuierlichen Systemen wurde am Georgia Institute of Technology untersucht. Die Systemmodelle werden dabei in die Simulationssprache Modelica abgebildet, um die Verwendung der Modelle für die Simulation nachzuweisen (siehe Masterarbeit „Integrating



Models and Simulations of continuous dynamic System Behavior into SysML“ von Thomas A. Johnson [18]). Des Weiteren gab es eine Arbeit, die sich mit ersten Gedanken zu einer Abbildung von zeitdiskreten Modellen in SysML in eine ausführbare Simulationssprache beschäftigt hat (siehe „A simple Example of SysML-driven Simulation“ von Leon McGinnis und Volkan Ustun [9]).

Bislang ist die Ausführbarkeit von kombinierten Systemmodellen in SysML nicht untersucht und die vorliegende Arbeit soll einen ersten Schritt in diese Richtung gehen. Konkret soll in dieser Arbeit herausgefunden werden, welche Einschränkungen und Erweiterungen an SysML vorgenommen werden müssen, damit zeitdiskrete Modelle unter Verwendung des Simulations-Frameworks ODEmx ausgeführt werden können. Die Simulationsbibliothek ODEmx [8] entstand an der Lehr- und Forschungseinheit Systemanalyse am Institut für Informatik der Humboldt-Universität zu Berlin. ODEmx ermöglicht es einen Simulator in der Sprache C++ für eine ereignis- oder prozessorientierte Simulation von zeitdiskreten, zeitkontinuierlichen und kombinierten Systemen zu erstellen. Die Bibliothek und ihre Erweiterungen sollen genutzt werden, um die Abbildung in ein C++-Programm einer prozessorientierten Simulation umzusetzen.

Diese Arbeit soll auf Diagramme zur Modellierung von Systemstrukturen und Zustandsautomaten für die Beschreibung von Systemverhalten beschränkt bleiben. Zusätzlich wird eine Aktionssprache benötigt, um Aktionen zur Modelländerung anzugeben. Es wird vermutet, dass sich C++ auf Grund der Abbildung in die C++-Simulationsbibliothek ODEmx eignet und keine neue Sprache definiert werden muss. Dies soll mit der Arbeit gezeigt werden. Darüber hinaus soll die Arbeit einen Vorschlag skizzieren, mit dem eine Abbildung in Bezug auf zeitkontinuierliche und kombinierte Systeme realisiert werden kann.

### 1.3 Vorgehensweise

In den folgenden Kapiteln wird SysML und die Erweiterung SimML hinsichtlich ihrer Konzepte und Anwendbarkeit untersucht, um zeitdiskrete Systeme zu modellieren und in ein Programm für eine prozessorientierte Simulation einer Erweiterung der ODEmx-Bibliothek abzubilden.

Das zweite Kapitel definiert zunächst einige grundsätzliche Begriffe der Systemmodellierung. Anschließend werden die Konzepte und die Modellierung von Systemen in SysML beschrieben.

Kapitel drei erklärt die Erweiterung SimML, mit Hilfe derer ausführbare Modelle entstehen können. Zusätzlich wird eine Bibliothek erläutert, die wiederverwendbare Simulationskonzepte enthält. Zuletzt wird eine Aktionssprache eingeführt (siehe Abschnitt 3.4), die für Aktionen in Verhaltensbeschreibungen von SimML-Modellen genutzt werden kann.

Das Kapitel vier gibt zunächst eine Einführung in die prozessorientierte Simulation und die vorhandenen Konzepte zur Beschreibung von Simulationsprogrammen mit ODEmx. Für die Abbildung von SimML-Modellen in ODEmx-

Simulationsprogramme wird eine Erweiterung der ODEMX-Bibliothek beschrieben.

Die in Kapitel vier definierten Erweiterungen werden im fünften Kapitel verwendet, um eine prinzipielle Abbildung eines SimML-Modells in ein ODEMX-Simulationsprogramm zu zeigen. Es sind zeitdiskrete Systeme mit Hilfe der Abbildung in ein ausführbares Programm überführbar.

In dem letzten Kapitel wird ein Resümee der gesamten Arbeit mit dem Schwerpunkt der Abbildung des SimML-Modells in ein ODEMX-Simulationsprogramm gegeben. Es folgt ein Ausblick für zukünftige Arbeiten in diesem Bereich.

### 1.4 Notation

Für Bezeichner von Modellelementen wird im Text der **Schreibmaschinenstil** verwendet.

Stereotypen werden ebenfalls in diesem Stil dargestellt und zusätzlich mit den französischen Anführungszeichen ( << . . . >>; Guillemets) mit Spitzen nach außen versehen.

Fachbegriffe aus UML, SysML oder SimML werden *kursiv* dargestellt und bei der ersten Verwendung erfolgt in Klammern die Angabe über die Sprachdefinition und der englischen Originalbezeichnung. Im Anhang B.1 sind alle Fachbegriffe zusammenfassend aufgeführt.

## 2 Systemmodellierung mit SysML

Die OMG hat in Zusammenarbeit mit dem International Council on Systems Engineering (INCOSE) im September 2001 Anforderungen für die Definition einer standardisierten Erweiterung der UML als Modellierungssprache zur Spezifikation, zum Design und zur Verifikation von komplexen Systemen formuliert. SysML entstand aus diesen Anforderungen als ein Profil für UML und führt Konzepte für die Modellierung von Systemen ein. Der UML-Profilmechanismus ermöglicht es neue Konzepte und Modellbibliotheken hinzuzufügen und Einschränkungen zu definieren. SysML erweitert nicht nur UML, sondern verbietet auch die Verwendung einiger Konzepte aus UML. Die Sprache SysML setzt sich somit aus den im SysML-Profil definierten Konzepten und Konzepten der Sprache UML, die in SysML wiederverwendet werden (UML4SysML) zusammen (weiterführende Informationen in „Systemmodellierung mit SysML“ von Peer Hausding [17]).

Nachfolgend soll die Sprache SysML für die Systemmodellierung betrachtet werden. Es werden ausschließlich die Konzepte in SysML erklärt, die später für eine Abbildung in eine ausführbare Simulation benötigt werden (Anforderungs- und Aktivitätsmodellierung entfallen beispielsweise komplett). Zunächst erfolgen Definitionen aus dem Bereich der Systemmodellierung allgemein und anschließend sollen diese Begriffe der Erläuterung der Konzepte der Sprache SysML dienen.

### 2.1 Systembegriffe

Der Begriff „System“ und Dinge, die damit in direkter Beziehung stehen, werden für viele Objekte in unserer Erfahrungswelt verwendet. In dieser Arbeit gelten die nachfolgenden Definitionen aus „Objektorientierte Prozeßsimulation in C++“ von Joachim Fischer und Klaus Ahrens [8].

Ein Phänomen wird als System bezeichnet, wenn die folgenden Merkmale erfüllt sind.

1. Das Phänomen erfüllt eine bestimmte Funktion (Systemzweck).
2. Das Phänomen besteht aus einer bestimmten Zusammensetzung von Objekten (Systemelementen), die untereinander in Relation (Wirkungsbeziehung) stehen.
3. Das Phänomen verliert seine Identität (Systemidentität), wenn seine Integrität (Systemintegrität) zerstört wird, d. h. das System ist nicht teilbar.

Ein System lässt sich von seiner Umwelt trennen und die Grenze wird als Systemgrenze bezeichnet. Die Umwelt eines Systems nennt man auch Systemumgebung,



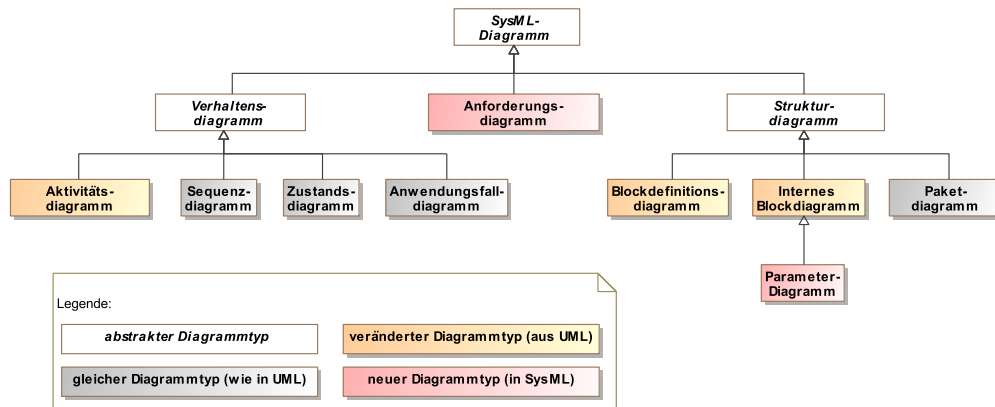


Abbildung 2.2: SysML-Diagrammtypen

stemmodell (Gesamtheit aller Modelle des Systems) in ein Modell einer Simulationssprache zu überführen (siehe Kapitel 5).

SysML ist genau wie UML eine objektorientierte Sprache und es werden Typen modelliert, deren Instanzen in einer Systemkonfiguration das reale System widerspiegeln sollen. Zunächst werden die Möglichkeiten der Systemstrukturdefinition und anschließend die der Verhaltensbeschreibungen für Typen von Systemelementen in SysML vorgestellt. Im letzten Abschnitt wird die Modellierung der Systemkonfiguration gezeigt. Für die Modellelemente werden deutsche Begriffe verwendet ähnlich wie in „Systems Engineering mit SysML/UML“ von Tim Weilkiens [19]. Eine Auflistung der Begriffe und deren englische Bezeichnung in der SysML-Sprachdefinition wird in Anhang B.1 gegeben. Die deutschen Begriffe orientieren sich an den englischen Begriffen, damit auf der Modellebene die englische Bezeichnungen der Stereotypen und die Verwendung der deutschen Begriffe nicht zu einer Verwechslung führt.

## 2.3 Beschreibung von Systemstruktur

UML erlaubt bereits die Typdefinition von *Datentypen* (UML: data type) und *Klassen* (UML: class) und deren Verwendung als Strukturierungselemente für Typbeschreibungen von Softwaresystemen und -elementen. Diese können untereinander *Assoziationen* (UML: Association) besitzen oder mit Hilfe von *Eigenschaften* (UML: property) weiter verschachtelt sein. Die Instanzen dieser Typdefinitionen spiegeln dann ein konkretes Softwaresystem wieder. Zusätzlich führt UML das Konzept der *Komponente*<sup>1</sup> (UML: component) ein und ermöglicht somit die Definition von Typen, die zusätzliche *Eigenschaften* gegenüber einer Klassendefinition besitzen können. Ausgehend von den erlaubten Strukturierungsmög-

<sup>1</sup> *Komponenten* sind spezielle *Klassen* und können beispielsweise *Ports* (UML: ports) besitzen, die mit Hilfe von *Konnektoren* (UML: connector) mit anderen *Komponenten* verbunden sind.

lichkeiten in UML wurden in SysML Konzepte erweitert oder neu eingeführt. SysML gestattet die Typdefinitionen von *Werttypen* (SysML: value type) und *Blöcken* (SysML: block) und deren Verwendung als Strukturierungselemente für Typbeschreibungen von Systemen und Systemelementen. *Instanz-Spezifikationen* (UML: instance specification) können in SysML genau wie in UML genutzt werden, um Instanzen von Typbeschreibungen zu repräsentieren. *Werttyp* erweitert die Definition des *Datentyps* und *Block* die der *Klasse* aus UML. Ein *Block* kann *Operationen* (UML: operation) besitzen oder mit einer Verhaltensbeschreibung verknüpft werden, so dass die Instanz eines *Blockes* ein Verhalten ausführen kann (siehe Kapitel 2.4). Genau wie in UML können diese Typdefinitionen in *Paketen* (UML: package) verwaltet werden.

### 2.3.1 Werttyp

*Werttypen* sind Typen, deren Instanzen keine Identität besitzen, d. h. Instanzen von *Werttypen*, deren Werte gleich sind, können nicht von einander unterschieden werden. Außerdem können Instanzen von *Werttypen* kein Verhalten ausführen und dienen somit nur der Speicherung von Werten des Systems. SysML definiert die *Werttypen*: *Numbers*(abstrakt), *Integer*, *Real*, *Complex*, *String* und *Boolean*. Ein *Werttyp* erweitert das Konzept des *Datentyps* aus UML hinsichtlich der Möglichkeit der Angabe von *Einheit* (SysML: unit) und *physikalischer Größe* (SysML: quantity kind).

### 2.3.2 Block

Ein *Block* ist eine Typbeschreibung eines Systems- oder Systemelements sein. Er besitzt eine Menge von *Eigenschaften* und lässt sich somit weiter strukturieren. Instanzen von *Blöcken* besitzen eine Identität, d. h. zwei Instanzen eines *Blockes* lassen sich unterscheiden, selbst wenn sie vom gleichen Typ sind und die Instanzen ihrer *Eigenschaften* gleich sind. Außerdem kann ein *Block* als *gekapselt* (SysML: isEncapsulated) modelliert werden, so dass er eine Black-Box<sup>2</sup> darstellt und nur über seine *Ports* oder die äußere Grenze *Assoziationen* oder *Konnektoren* festlegen kann.

*Blockeigenschaften* (SysML: block property) entsprechen den *Klasseneigenschaften* (UML: class property) angewandt auf das in SysML neu eingeführte Konzept *Block*. Eine *Blockeigenschaft* lässt sich in eine der folgenden Kategorien einordnen und SysML erlaubt im *Blockdefinitionsdiagramm* (SysML: block definition diagram) die Darstellung der *Blockeigenschaften* eines *Blockes* nach diesen Kategorien in eigenen *Abteilen* (UML: compartment). Im *Internen Blockdiagramm* (SysML: internal block diagram) können die Beziehungen zwischen den *Eigenschaften* eines *Blockes* ausgedrückt werden.

---

<sup>2</sup>Als Black-Box bezeichnet man in der Systemtheorie ein System, von welchem im gegebenen Zusammenhang nur das äußere Verhalten betrachtet werden soll. Die innere Struktur darf nicht benutzt werden.

### Blockteil

Eine *Blockeigenschaft*, deren Typ ein *Block* ist und eine *Kompositionsbeziehung* (UML: composite aggregation) zum *Block* besitzt, wird als *Blockteil* (SysML: block part property) kategorisiert. Dies ermöglicht eine hierarchische *Block*-Komposition.

### Blockreferenz

Eine *Blockeigenschaft*, deren Typ ein *Block* ist und eine *Aggregationsbeziehung* (UML: shared aggregation) zum *Block* besitzt, wird als *Blockreferenz* (SysML: block reference property) kategorisiert.

### Blockwert

Eine *Blockeigenschaft*, deren Typ ein *Werttyp* ist, wird der Kategorie *Blockwert* (SysML: block value property) zugeordnet, unabhängig davon, ob eine *Kompositions-* oder *Aggregationsbeziehung* verwendet wird.

### Blockbedingung

Die Kategorie *Blockbedingung* (SysML: block constraint property) trifft dann auf die *Blockeigenschaft* zu, falls der Typ ein *Bedingungsblock* (SysML: constraint block) ist (siehe Kapitel 2.4), unabhängig davon, ob eine *Kompositions-* oder *Aggregationsbeziehung* verwendet wird.

### Flussport

Der *Port* ist eine besondere *Eigenschaft*. UML führt *Schnittstellen* (UML: interface) als *Eigenschaft* eines *Ports* ein, um eine vereinbarte, synchrone Kommunikationsschnittstelle definieren zu können. In SysML wird dies um das Konzept des *Flussports* (SysML: flow port) erweitert. Ein *Flussport* kann mit Hilfe einer *Flussspezifikation* (SysML: flow specification) (Erweiterung von *Schnittstelle*) angeben auf welche Weise *Elementen* (SysML: item), die beliebigen Instanzen zur Laufzeit entsprechen können, ausgetauscht werden dürfen. Die *Flusseigenschaften* (SysML: flow property) einer *Flussspezifikation* legen hierfür einen Typ und eine Richtung fest. *Elemente* können dann nur entsprechend der *Flussspezifikation* ausgetauscht werden. *Flussports* können über eine Verbindung, dem *Elementfluss* (SysML: item flow), mit anderen *Flussports* verbunden werden. Eine *Blockeigenschaft* des umgebenen *Blockes* (der sogenannte Blockkontext) kann mit diesem *Elementfluss* verknüpft werden, so dass über die Modellierung der *Blockeigenschaften* der konkrete Austausch der *Elemente* spezifiziert werden kann (siehe Kapitel 2.4).

### 2.3.3 Werteverteilung

Für *Eigenschaften* von *Werttypen* oder *Blöcken* führt SysML das Konzept *Werteverteilung* (SysML: distributed property) ein. Das Konzept kann durch den Anwender verfeinert werden, indem speziellere *Werteverteilungen* mit konkreten Verteilungsfunktionen definiert werden. Wird eine *Werteverteilung* dann auf eine *Eigenschaft* angewandt, so müssen die Werte dieser *Eigenschaft* der Verteilungsfunktionen der *Werteverteilung* entsprechen. Die genaue Semantik der Anwendbarkeit auf eine *Eigenschaft* und Berechnung der Verteilungsfunktionen muss durch den Anwender definiert werden.

### 2.3.4 Notation

Das *Blockdefinitionsdiagramm* (BDD) und das *Interne Blockdiagramm* (IBD) dienen der Definition von *Blöcken* und *Werttypen*. Das *Blockdefinitionsdiagramm* entspricht dem UML-Klassendiagramm und das *Interne Blockdiagramm* dem UML-Kompositionsstrukturdiagramm (UML: composite structure diagram). Beide wurden im Hinblick auf die Definition von *Block* und *Werttyp* angepasst.

#### Blockdefinitionsdiagramm (BDD)

Die Abbildung 2.3 zeigt das *Blockdefinitionsdiagramm*. Typdefinitionen, welches im *Paket* *Systembegriffe* erstellt wurde. Es enthält die *Block*-Definitionen *Systemblock*, *Block1*, *Block2*, *Block3* und *Block4*. *Systemblock* soll, wie der Name bereits andeutet, den äußersten *Block* des Systemtyps und damit die Systemgrenze festlegen. Er enthält zwei *Blockteile*, die wiederum weitere *Blockteile*, *Blockreferenzen* und *-werte* festlegen.

Der *Blockwert* *systemWert1* der *Block*-Definition *Block3* ist vom *Werttyp* *Werttyp1* und besitzt eine *Einheit* *Einheit1*. Die Definition der *Einheit* erfolgte ebenfalls in diesem Diagramm und verwendet die *physikalische Größe* *physikalischeGröße1* (auch diese wurde erst in diesem Diagramm modelliert).

#### Internes Blockdiagramm (IBD)

Das *Interne Blockdiagramm* in Abbildung 2.4 zeigt die interne Struktur des *Blockes* *Systemblock*. Zu sehen ist seine komplette *Block*-Hierarchie. *Blockteile* und *Blockwerte* werden als Rechtecke mit einer durchgezogenen und *Blockreferenzen* mit einer gestrichelten Linie dargestellt.

## 2.4 Beschreibung von Systemverhalten

Die Verhaltensbeschreibung für Systeme kann grundsätzlich zeitdiskrete oder zeitkontinuierliche Änderungen der Zustandsgrößen des Systems beschreiben.



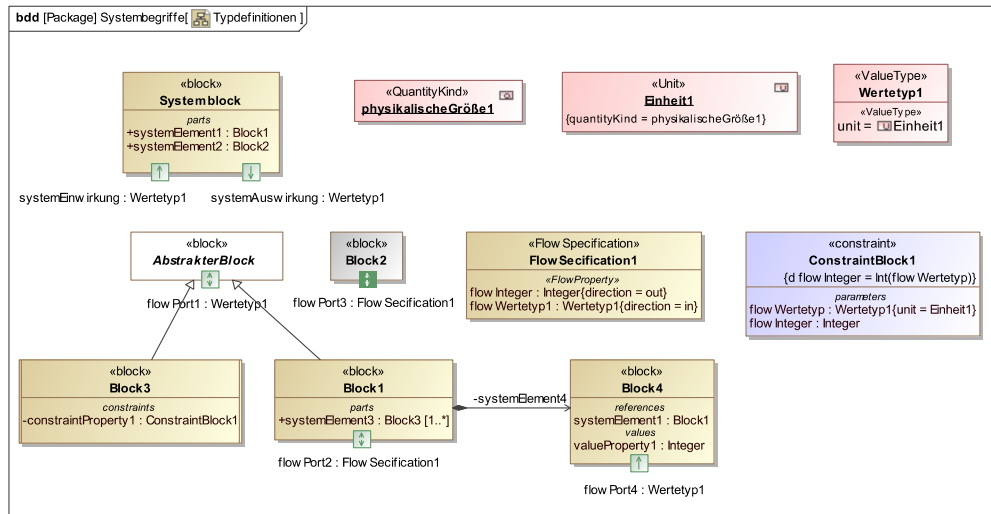


Abbildung 2.3: Block- und Werttyp-Definitionen

UML bietet die Möglichkeit diskretes, ereignisbasiertes Verhalten zu beschreiben. Es können verschiedene Formalismen genutzt werden: *Zustandsdiagramm* (Automat), *Aktivitätsdiagramm* (Petri-Netz ähnlicher Graph), *Anwendungsfall-diagramm* (informale Beschreibung) und *Interaktionsdiagramm* (partiell geordnete Sequenzen von Ereignissen).

Mit SysML können diese Möglichkeiten ebenfalls genutzt werden und es werden zusätzliche Konzepte, die die Modellierung von zeitkontinuierlichen Änderungen von Zustandsgrößen ermöglichen, definiert. Die Erweiterungen betreffen den *Objektfluss* in *Aktivitätsdiagrammen* aus UML, der in SysML nicht nur zu einem Zeitpunkt stattfinden kann, sondern kontinuierlich über einen Zeitraum. Neben dieser Erweiterung ermöglicht SysML in der Strukturbeschreibung (siehe Abbildung 2.2) die Definition und Verwendung von *Bedingungsblöcken*, die *Bedingungen* modellieren, welche beispielsweise mit Hilfe von Differentialgleichungen zeitkontinuierliche Änderungen von *Blockwerten* festlegen können.

Nachfolgend erfolgt ein Überblick über die Modellierung von diskreten, ereignisbasierten Verhaltensbeschreibungen mit Hilfe von *Zustandsdiagrammen* (UML: state machine diagram) aus UML und anschließend eine ausführliche Erklärung zur Möglichkeit der Modellierung von zeitkontinuierlichen Zustandsänderungen in der neuen SysML-Diagrammart *Parameterdiagramm*.

### 2.4.1 Beschreibung von zeitdiskreten Zustandsänderungen

Ein *Block* kann mit einer *Verhaltensbeschreibung* verbunden sein, welches dem Verhalten der Instanz eines *Blockes* entsprechen soll. Die Verhaltensbeschreibung besitzt damit einen Kontext und einen Zugriff auf die Strukturbeschreibung.

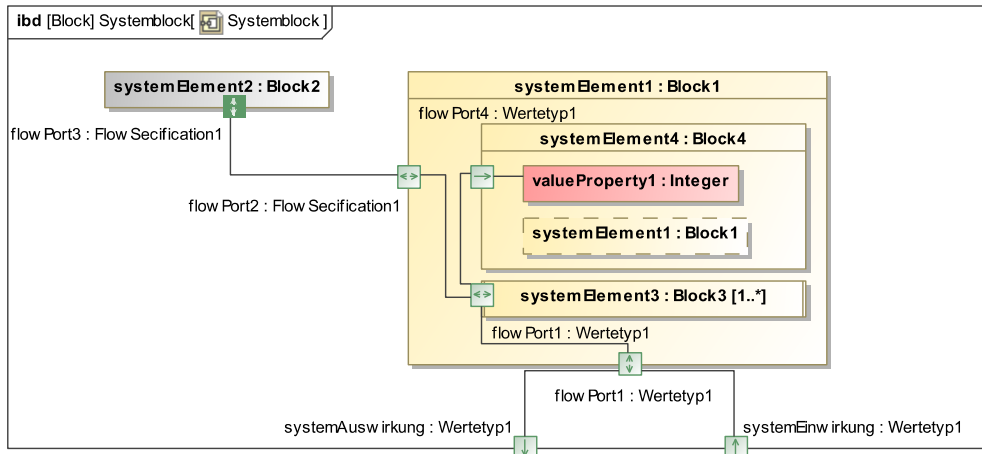


Abbildung 2.4: Block Systemblock

### Zustandsautomat

Mit Hilfe des *Zustandsdiagramms* können ähnlich wie mit dem Formalismus des Endlichen Automaten von Harel [2] diskrete Zustandsübergänge des Systems modelliert werden. Der *Zustandsautomat* besitzt jedoch keine grafische Notation und setzt sich aus *Pseudozuständen*, *Zuständen* und *Transitionen* zusammen. Die *Trigger* der *Transitionen* legen fest, wie das Auftreten von *Ereignissen* behandelt wird und zu Zustandsübergängen führen kann. In *Zuständen* und an *Transitionen* können weitere *Verhaltensbeschreibungen* festgelegt werden. Der *Zustandsautomat* verhält sich nach der Run-to-Completion-Semantik, d. h. das die Behandlung eines auftretenden *Ereignisses* erst nach der vollständigen Behandlung des vorherigen *Ereignisses* erfolgt.

#### 2.4.2 Beschreibung von zeitkontinuierlichen Zustandsänderungen

Das *Parameterdiagramm* dient der Beschreibung der internen Struktur eines Blockes genau wie das *Interne Blockdiagramm* mit dem Zusatz, dass *Blockbedingungen* in Verbindung mit anderen *Blockeigenschaften* dargestellt werden können. *Blockbedingungen* besitzen die Möglichkeit über eine *Bedingung* Differentialgleichungen zu verwenden und somit Verhalten zu beschreiben.

### Bedingungsblock

Die *Blockbedingung* ist eine *Eigenschaft* eines Blockes, deren Typ ein *Bedingungsblock* ist. Der *Bedingungsblock* ist ein spezieller *Block*, welcher eine *Bedingung* über seine *Eigenschaften* formuliert. Die *Eigenschaft* in Form eines *Ports* wird in diesem Zusammenhang in einem *Abteil Bedingungsparameter* geführt. Er dient

dem Austausch von Instanzen, deren *Eigenschaften* in den Bedingungen verwendet werden können. Diese *Bedingungen* können beispielsweise Differentialgleichungen über *Blockwerte* sein und ermöglichen somit die Typbeschreibung von Systemelementen, deren Zustandsgrößen sich zeitkontinuierlich ändern.

### Bindungskonnektor

Die Verbindung zwischen *Eigenschaften* von *Blöcken* und *Bedingungsparametern* legt der *Bindungskonnektor* fest. Sind die *Eigenschaften*, die über den *Bindungskonnektor* mit dem *Bedingungsparametern* verbunden sind *Wertetypen*, so müssen die vorliegenden Werte gleich sein. Falls es sich um Instanzen von *Blöcken* handelt, so referenzieren beide die gleiche Blockinstanz.

### 2.4.3 Notation

Das *Zustandsdiagramm* dient der Beschreibung von zeitdiskretem, ereignisbasiertem Verhalten. SysML führt das *Parameterdiagramm* ein, um in der Strukturbeschreibung zusätzlich auch die zeitkontinuierliche Veränderung von Zustandsgrößen beschreiben zu können. Die *Bedingungen* können in einer beliebigen Sprache formuliert werden und so können beispielsweise Differentialgleichungen die Änderungsrate von *Blockwerten* beschreiben.

### Parameterdiagramm

Das *Parameterdiagramm* in Abbildung 2.5 zeigt die *Blockbedingung* des Blockes **Block3**. Diese ist vom Typ **ConstraintBlock1** und besitzt zwei *Parameter*, die mit dem *Flussport* **flowPort1** über *Bindungskonnektoren* verbunden sind. Instanzen vom Typ **Wertetyp1** müssen somit die *Bedingung* **constraintProperty1** erfüllen, falls sie mit dem *Flussport* des Blockes **Block3** ausgetauscht werden.

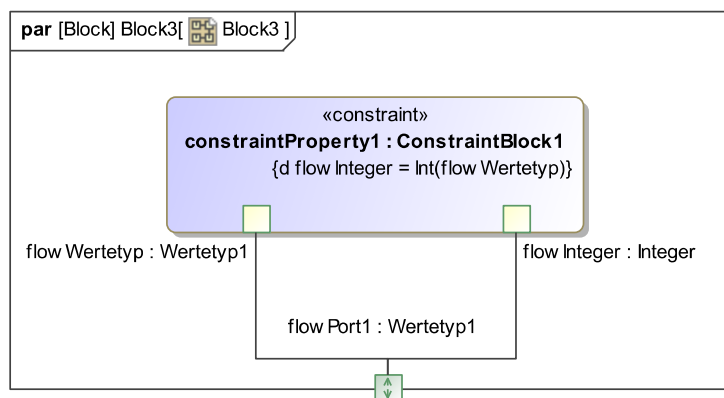


Abbildung 2.5: Blockbedingung des Blockes Block3

## 2.5 Systemkonfiguration

Ein *Schnappschuss* bezeichnet in SysML die Angabe der vorhandenen Instanzen und Wertebelegungen zu einem Zeitpunkt. Dieser kann genutzt werden, um eine initiale Systemstruktur zu beschreiben. Mit Hilfe der *Instanz-Spezifikation* wird beispielsweise eine Instanz eines *Blockes* oder eines *Wertetyps* spezifiziert. Es können *Blockwerten* initiale Werte zugewiesen werden. Außerdem können mit *Links* vorhandene Verbindungen zwischen *Instanz-Spezifikationen* beschrieben werden (Instanzen von *Assoziationen*).

### 2.5.1 Notation

Das *Blockdefinitionsdiagramm* kann verwendet werden, um *Instanz-Spezifikationen* und initiale Werte anzugeben.

#### Objektdiagramm

Die Abbildung 2.6 zeigt einen *Schnappschuss* der Systemtypdefinition aus Abbildung 2.3.

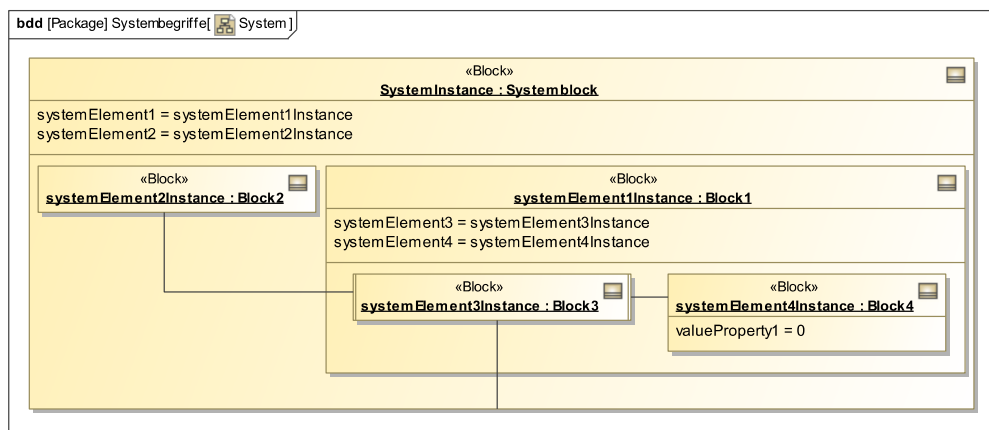


Abbildung 2.6: Instanz des Blockes Systemblock

Die abgebildeten Instanzen stehen in Relation zueinander, was durch die Angabe des Namens einer Instanz nach dem Gleichheitszeichen einer *Eigenschaft* eines *Blockes* zu erkennen ist (z. B. Verbindung der *Blockeigenschaft* `systemElement1` der Blockinstanz `SystemInstance` mit der Instanz `systemElement1Instance`. Die Belegung eines *Blockwertes* ist im Modell beispielsweise durch `valueProperty1` dargestellt.

## 3 Die Sprache SimML

Die Simulation Modeling Language (SimML) ist eine objektorientierte Modellierungssprache, die sich aus Teilen der Sprache UML und SysML zusammensetzt und Erweiterungen definiert, um Systemtypen, deren Instanzen und Verhalten beschreiben zu können, mit dem Ziel Modelle der Sprache ausführen zu können. Nachfolgend wird beschrieben, welche Modellierungselemente ein gültiges SimML-Modell besitzen kann, wie sie in Beziehung stehen dürfen (Syntax) und welche Bedeutung sie haben (Semantik).

Die Sprache setzt sich ausschließlich durch die in diesem Kapitel aufgeführten Modellelemente zusammen. Die Grundlage bildet die Sprache SysML bzw. UML und die Abweichungen der Syntax oder Semantik gleicher Modellelemente wird mit Hilfe von Bedingungen in der Object Constraint Language (OCL) [12] und/oder in Text festgehalten. Um dies klar herauszustellen werden diese Abweichungen als SimML-Regeln gekennzeichnet.

### 3.1 Namensgebung

Für die nachfolgend aufgelisteten Modellierungselemente ist ein eindeutiger und korrekter Name notwendig.

- Paket
- Block
  - Blockteil
  - Blockreferenz
  - Blockwert
  - Operation
- Zustandsmaschine
  - Zustand
  - Entscheidungsknoten
- Instanzspezifikation

Die Eingerückten müssen nur innerhalb der Definition des übergeordneten eindeutig sein, da innerhalb derer ein eigener Namensraum eröffnet wird. Ein korrekter Name hat als erstes Zeichen einen Buchstaben oder einen Unterstrich und

besitzt eine beliebige Folge von Buchstaben, Zahlen oder Unterstrichen. Das Listing 3.1 zeigt diese Regel in Backus-Naur-Form<sup>1</sup>.

```
1 < identifier > ::= < letter > | < underline > { < letter > | < underline > | < digit > }*
```

Listing 3.1: Gültige Namen in SimML

## 3.2 Strukturbeschreibung

Die Strukturbeschreibung bezieht sich auf die Definition von Typen und deren Verwendung bei der Definition neuer strukturierter Typen. Daneben gibt es vordefinierte primitive *Werttypen*.

### 3.2.1 Paket

Eine hierarchische Paketdefinition ist für ein SimML-Modell nicht möglich. Es muss genau ein *Paket* geben, innerhalb dessen Typdefinitionen erfolgen.

#### SimML-Regel: Paket-Validierung

```
1 context Model inv validatePackages :
2 nestedPackage -> reject(name.matches('UML Standard Profile')) -> select(getAppliedStereotype('
Kernel::Simulation')).oclIsUndefined() -> size() = 1
```

Listing 3.2: SimML-Regel: Paket-Validierung

Die Benennung der definierten Typen muss eindeutig geschehen. Alle Definitionen haben die Sichtbarkeit *public*, jede Typdefinition liegt also im Sichtbarkeitsbereich der anderen Typdefinitionen.

#### SimML-Regel: Sichtbarkeit einer Typdefinition

```
1 context Type inv typeVisibility :
2 visibility = VisibilityKind :: public
```

Listing 3.3: SimML-Regel: Sichtbarkeit einer Typdefinition

### 3.2.2 Wertetyp

Es existieren die vordefinierten *Werttypen*: *Integer*, *Real*, *Complex*, *String* und *Boolean*. Nur *Complex* besitzt *Werttypeigenschaften*, die anderen hingegen nicht. *Integer* besitzt den Wertebereich einer natürlichen Zahl, *Real* den einer reellen

---

<sup>1</sup>Die Backus-Naur-Form (BNF) ist eine formale Metasprache zur Darstellung kontextfreier Grammatiken.

Zahl, *Boolean* die Wahrheitswerte `true` und `false`, *String* eine beliebige Zeichenkette und *Complex* den Wertebereich einer komplexen Zahl mit Hilfe der *Werte-typeeigenschaften* `realPart` für den Realteil und `imaginaryPart` für den Imaginärteil, beide vom *Wertetyp* *Real*. Die vordefinierten *Wertetypen* besitzen die nachfolgend aufgeführten initialen Belegungen.

Wertetyp	Belegung
Integer	0
Real	0.0
String	„“
Boolean	false

Tabelle 3.1: Initiale Belegung vordefinierter Wertetypen

### 3.2.3 Block

Die Definition eines *Blockes* ist eine Typdefinition, die als Kontext einer Verhaltensbeschreibung dienen kann. Ein *Block* kann die *Eigenschaft* aktiv besitzen, so dass die Definition von Verhalten in Form einer *Zustandsmaschine* angegeben werden muss. *Blöcke* ohne eigene Verhaltensbeschreibung sind hingegen passiv.

#### SimML-Regel: Aktiver Block

```

1 context Class inv activeClass :
2 if isActive then not ( classifierBehavior .oclAsType(StateMachine).oclIsUndefined() ) else
   classifierBehavior .oclIsUndefined() endif

```

Listing 3.4: SimML-Regel: Aktiver Block

Ein *Block* kann von einem oder mehreren *Blöcken* erben. Hierbei werden alle *Blockeigenschaften* und *Operationen* geerbt und sind in allen abgeleiteten Klasse verfügbar.

### 3.2.4 Blockeigenschaft

Ein *Block* kann *Eigenschaften* (*Blockeigenschaft*) und *Operationen* besitzen. Eine *Blockeigenschaft* muss einen Typ und immer die Sichtbarkeit *public* besitzen. Der Typ ist entweder ein *Wertetyp* oder ein *Block*.

#### SimML-Regel: Blockeigenschaft

Eine *Blockeigenschaft* kann von einem Mengentyp sein (siehe UML 7.3.32 [14]). Es können eine obere und/oder eine untere Grenze für die Anzahl der Instanzen angegeben werden.

```

1 context Class inv classProperty :
2 attribute ->forall( visibility = VisibilityKind :: public and (type.ocllsKindOf(Class) or type.
   occllKindOf(DataType)))

```

Listing 3.5: SimML-Regel: Blockeigenschaft

### SimML-Regel: Änderung einer mehrwertigen Blockeigenschaft

```

1 Falls beim Ändern der mehrwertigen Blockeigenschaft eine der Grenzen verletzt werden würde,
  so wird die Änderung nicht vollzogen.

```

Listing 3.6: SimML-Regel: Änderungen mehrwertigen Blockeigenschaften

Mehrwertige *Blockeigenschaften* können zusätzlich die Kennzeichnung geordnet und eindeutig besitzen. Ist nichts anderes angegeben, so gilt, dass die Elemente der Menge nicht geordnet und eindeutig sind. Es ergeben sich über das Kreuzprodukt der Eigenschaften (geordnet und eindeutig) vier verschiedene Mengentypen. Die Tabelle 3.2 gibt die Bezeichnung des Mengentyps anhand der Kombination der Eigenschaften an (siehe UML 7.3.44 [14]). In SysML bzw. UML ist nicht

geordnet	eindeutig	Mengentyp
ja	nein	Sequence
ja	ja	OrderedSet
nein	nein	Bag
nein	ja	Set

Tabelle 3.2: Mengentypen mehrwertiger Blockeigenschaften

konkret spezifiziert, wie die Elemente geordnet werden sollen, daher gilt folgende Ordnungsregel.

### SimML-Regel: Ordnung einer mehrwertigen Blockeigenschaft

```

1 Die Ordnung der Elemente erfolgt nach der Reihenfolge ihrer Einordnung, falls es sich um
  eine geordnete Menge handelt.

```

Listing 3.7: SimML-Regel: Ordnung einer mehrwertigen Blockeigenschaft

Eindeutig bedeutet, dass ein Element nur einmal in der Menge existieren darf. Das Hinzufügen von bereits vorhandenen Elementen in eine eindeutige Menge ist nicht festgelegt (UML 7.3.32 [14]). Die nachfolgende Regel bestimmt die Semantik.

### SimML-Regel: Änderung einer mehrwertigen, eindeutigen Blockeigenschaft



- 
- ```

1 Das Hinzufügen eines Elementes zu einer eindeutigen Menge, welches bereits in der Menge
  vorhanden ist, bewirkt keine Änderung an der Menge.

```
- 

Listing 3.8: SimML-Regel: Änderung einer mehrwertigen, eindeutigen Blockeigenschaft

### 3.2.5 Operation

Für *Blöcke* können *Operationen* angelegt werden. Diese können einen *Rückgabewert* und beliebig viele *Parameter* besitzen. Ein *Parameter* oder *Rückgabewert* kann vom Typ eines *Blockes* oder eines *Wertetyps* sein. Als Richtung für einen *Parameter* ist nur `inout` erlaubt.

#### SimML-Regel: Operationsparameter

- ```

1 context Operation inv operationParameter:
2 ownedParameter -> forAll(direction = ParameterDirectionKind::inout or direction =
  ParameterDirectionKind::return)

```
- 

Listing 3.9: SimML-Regel: Operationsparameter

## 3.3 Verhaltensbeschreibung

Nachfolgend werden ausschließlich zeitdiskrete Verhaltensbeschreibungen, die mit *Zustandsmaschinen* formuliert werden können, erklärt.

### 3.3.1 Zustandsmaschine

*Zustandsmaschinen* werden im Kontext eines *Blockes* definiert. Auf die im Blockkontext angelegten *Blockeigenschaften* und *Operationen* kann direkt zugegriffen werden.

Eine *Zustandsmaschine* muss genau eine *Region* besitzen, innerhalb derer *Knoten* (*Zustände* oder *Pseudozustände*) und *Transitionen* existieren.

#### SimML-Regel: Zustandsmaschinen-Region

- ```

1 context StateMachine inv stateMachineRegion:
2 stateMachine.region -> size() = 1

```
- 

Listing 3.10: SimML-Regel: Zustandsmaschinen-Region

Ein *Knoten* kann mit einer *Transition* verbunden sein, die wiederum mit genau einem Nachfolgeknoten verbunden sein muss. Ist ein *Knoten* mit keiner *Transition* verbunden, wird in diesem *Zustand* verharret.

### 3.3.2 Pseudozustand

Es gibt die *Pseudozustände*: *Initialknoten*, *Terminierungsknoten* und *Entscheidungsknoten*. Der *Initialknoten* ist der Startpunkt der *Zustandsmaschine*, muss genau einmal definiert sein und kann höchstens eine ausgehende *Transition* besitzen. An dieser *Transition* darf es keinen *Trigger* oder *Wächter* geben.

#### SimML-Regel: Region-Initialknoten

```
1 context Region inv regionInitial :  
2 subvertex ->select(ocllsKindOf(Pseudostate)).oclAsType(Pseudostate)->select(kind =  
   PseudostateKind::initial)->size() = 1
```

Listing 3.11: SimML-Regel: Region-Initialknoten

*Terminierungsknoten* kann es keinen oder beliebig viele geben. Er kann keine ausgehenden *Transitionen* haben.

#### SimML-Regel: Region-Terminierungsknoten

```
1 context Region inv regionTerminate:  
2 subvertex ->select(ocllsKindOf(Pseudostate)).oclAsType(Pseudostate)->select(kind =  
   PseudostateKind::terminate).outgoing->size() = 1
```

Listing 3.12: SimML-Regel: Region-Terminierungsknoten

Der *Entscheidungsknoten* kann beliebig viele ein- und ausgehende *Transitionen* besitzen, jedoch mindestens eine ein- und eine ausgehende. Die ausgehenden *Transitionen* können *Wächter* besitzen. Es muss sichergestellt sein, dass die *Bedingungen* der *Wächter* zur eindeutigen Auswahl genau eines Folgezustandes führen (siehe Abschnitt 3.3.6).

### 3.3.3 Zustand

Ein *Zustand* einer *Zustandsmaschine* muss mit einem eindeutigen Namen innerhalb der *Zustandsmaschine* versehen werden. Er besitzt mindestens eine eingehende und eine ausgehende *Transition*. *Zustände* dürfen nur **simple** sein, d. h. ein *Zustand* darf sich nicht aus anderen *Zuständen* zusammensetzen.

#### SimML-Regel: Einfachzustand

```
1 context State inv simpleState:  
2 isSimple
```

Listing 3.13: SimML-Regel: Einfachzustand

### 3.3.4 Transition

*Transitionen* verbinden *Knoten* miteinander und beschreiben mögliche Zustandsübergänge. Eine *Transition* hat genau einen Quell- und einen Zielknoten. Der Übergang von Quell- zu Zielknoten wird vollzogen, wenn ein *Ereignis* eintritt, dass den *Trigger* der *Transition* erfüllt. Eine *Transition* kann keinen oder beliebig viele *Trigger* besitzen. Der Übergang zwischen zwei *Knoten* mit einer *Transition* ohne *Trigger* wird direkt vollzogen (das *Abschlussereignis*, ist das *Ereignis* der vollständigen Realisierung des Quellzustandes). Besitzt ein Quellzustand mehrere *Transitionen*, deren *Trigger* durch die gleichen *Ereignisse* ausgelöst werden können oder keinen *Trigger*, dann ist die Auswahl der *Transition* nicht determiniert. Ebenfalls nichtdeterministisch ist die Auswahl einer *Transition*, falls mehrere durchlaufen werden könnten (gleichzeitige *Ereignisse*).

### SimML-Regel: Transitionsauswahl

- 1 Ist der Trigger von mehr als einer Transition eines Zustandes erfüllt, so wird die zu durchlaufende Transition anhand der Stellung in der internen Verwaltungsliste für ausführbare Transitionen gewählt. Gewählt wird die zuerst der Liste hinzugefügte Transition.

Listing 3.14: SimML-Regel: Transitionsauswahl

An einer *Transition* kann ein *Effekt* definiert werden, der ausgeführt wird, wenn die *Transition* durchlaufen wird (siehe 3.3.7).

### 3.3.5 Ereignis

Ein *Ereignis* kann ein *AnnotatedChangeEvent* (*Zustandsereignis*) oder ein *TimeEvent* (*Zeitereignis*) sein.

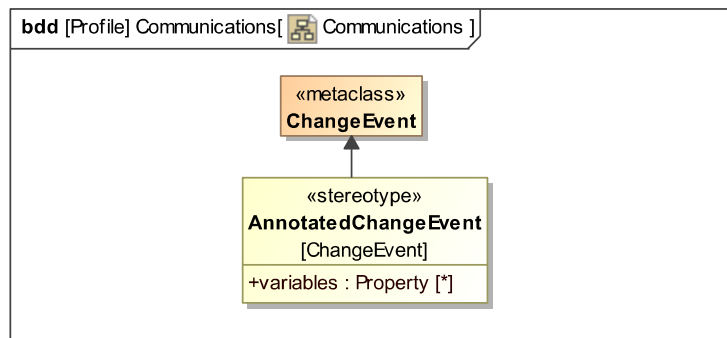


Abbildung 3.1: Stereotyp für SimML-Zustandsereignis

**SimML-Regel: Ereignistyp**

```

1 context Event inv eventType:
2 oclIsTypeOf(TimeEvent) or not (getAppliedStereotype('Communications::AnnotatedChangeEvent')
  ).oclIsUndefined())

```

Listing 3.15: SimML-Regel: Transitionsauswahl

Ein `AnnotatedChangeEvent` besitzt genau eine *Bedingung*, die innerhalb des Blockkontextes der *Zustandsmaschine* ausgewertet werden kann.

**SimML-Regel: Zustandseignis**

```

1 Die Formulierung der Bedingung erfolgt in der Programmiersprache C++ und muss einen
  booleschen Wert als Antwort liefern.

```

Listing 3.16: SimML-Regel: Zustandseignis

Genaueres zur Formulierung der *Bedingung* findet sich in Abschnitt 3.3.6. Über das Tag `variables` müssen die *Eigenschaften* hinterlegt sein, die in der *Bedingung* vorkommen.

Das Warten auf ein *Zeitereignis* kann relativ zur Modellzeit (relatives *Zeitereignis*) oder für eine konkrete Modellzeit (absolutes *Zeitereignis*) festgelegt werden (*Eigenschaft isRelative*). Es muss genau eine `TimeExpression` beschrieben sein, die zu einem natürlichen oder reellen Zahlenwert ausgewertet werden kann.

**3.3.6 Bedingung**

Eine *Bedingung* muss zu einem booleschen Wert ausgewertet werden können.

**SimML-Regel: Bedingung**

```

1 Eine Bedingung muss in C++ formuliert sein und darf keine Seiteneffekte hervorrufen. Die
  Bedingung darf keine Anweisungen enthalten (auch keine seiteneffektfreien ) und muss im
  Kontext des Blockes auswertbar sein.

```

Listing 3.17: SimML-Regel: Bedingung

**3.3.7 Effekt**

Der *Effekt* einer *Transition* ist eine Menge von Anweisungen, die Werte oder Instanzen des *Blockes* betreffen. Das Ändern der Instanzen an *Blockeigenschaften* erfolgt abhängig davon, ob es sich um ein *Blockteil*, *-referenz* oder *-wert* handelt. Eine Besonderheit stellen mehrwertige *Blockeigenschaften* und die *Blockeigenschaft* vom Typ *String* dar, da diese Typen über besondere Funktionen verfügen (siehe Abschnitt 3.4). Enthalten die Anweisungen lokale Variablen, so müssen deren Typen bekannt sein. Ist der Typ (*Block* oder *Werttyp*) einer lokalen Variablen nicht einer der *Blockeigenschaften* des Blockkontextes, so müssen diese

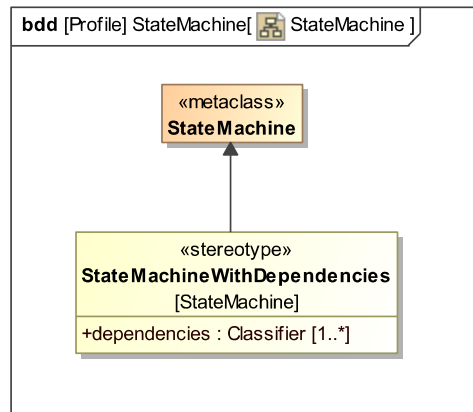


Abbildung 3.2: Zustandsmaschine mit zusätzlichen Abhängigkeiten

Typen explizit angegeben werden. Hierfür erhält die *Zustandsmaschine* den Stereotypen `<<StateMachineWithDependencies>>` (siehe Abbildung 3.2) und über das Tag `dependencies` müssen die verwendeten Typen angegeben werden.

### SimML-Regel: Effekt

- 1 Die Anweisungen eines Effekts müssen in C++ beschrieben sein und im Blockkontext der Zustandsmaschine ausgeführt werden können. Werden zusätzliche Typen benötigt, müssen diese über den Stereotyp `\element{<<StateMachineWithDependencies>>}` der Zustandsmaschine angegeben werden.

Listing 3.18: SimML-Regel: Effekt

## 3.4 Aktionssprache

Nachfolgend ist anhand des Beispiels aus der Abbildung 3.3 abhängig vom Typ der *Blockeigenschaft* erklärt, wie Instanzen neu angelegt, gelöscht, verändert, geprüft oder als *Parameter* für Operationsaufrufe angegeben werden können. Der Zugriff wird ausschließlich über den Kontext des *Blockes A* dargestellt (Beschreibung von *Effekten* in der *Zustandsmaschine* einer Instanz von *A*).

### 3.4.1 Blockteil/-referenz

*Blockteile* oder *-referenzen* sind, falls keine *Vorgabewerte* und in der Konfiguration für das System nicht anders angegeben (siehe Abschnitt 3.5), an sich mit keinen Instanzen belegt. Es sind Aktionen möglich, um diese zu ändern und um Instanzen über *Blockteile* oder *-referenzen* nicht mehr erreichbar zu machen. Falls über das Modell kein weiterer Zugriff auf eine Instanz möglich ist, weil diese nicht über ein *Blockteil* oder eine *-referenz* erreichbar ist und auch kein eigenes

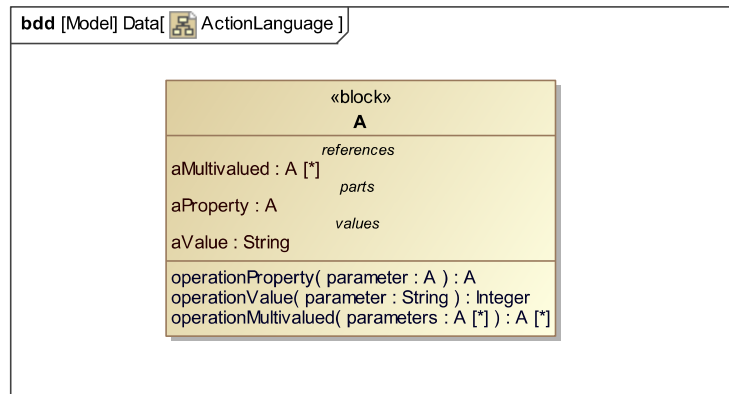


Abbildung 3.3: Beispiel Aktionssprache: Block A

Verhalten besitzt, so muss vorher die Instanz gelöscht werden, um ein mögliches Speicherloch zu verhindern (C++ Semantik für Objekte auf dem Heap). Falls die Instanz eigenes Verhalten besitzt und der *Terminierungsknoten* erreicht wird, so wird die Instanz nach dem Erreichen dieses Endzustandes entfernt. Wenn eine Instanz den *Terminierungsknoten* erreicht oder explizit gelöscht wurde, so können die *Blockeigenschaften*, die auf diese Instanz verweisen, diese nicht mehr erreichen (siehe UML 15.3.8 [14]). Das Erreichen einer Instanz über eine *Blockeigenschaft* kann abgefragt werden (siehe weiter unten „Instanz prüfen“).

- Instanz erstellen: Die *Blockeigenschaft* **aProperty** mit einer neuen Instanz von **A** belegen.

```
1  aProperty = new A();
```

- Instanz löschen: Die Instanz der *Blockeigenschaft* **aProperty** löschen.

```
1  delete aProperty;
```

- Instanz ändern: Eine lokale Variable mit einer neuen Instanz anlegen und diese der *Blockeigenschaft* **aProperty** von **aProperty** zuweisen.

```
1  A* a = new A();
2  aProperty->aProperty = a;
```

- Instanz prüfen: Prüfen, ob eine Instanz der *Blockeigenschaft* **aProperty** von **aProperty** existiert.

```
1  // if-then-else
2  if (aProperty->aProperty)
3      true;
4  else
5      false;
```

```

6 // short if-then-else
7 aProperty->aProperty ? true : false ;

```

- Operationsaufruf: Die Instanz von `aProperty` als Operationsparameter für `operationProperty` verwenden und den *Rückgabewert* in `aProperty` speichern.

```

1 aProperty = aProperty->operationProperty(aProperty);

```

### 3.4.2 Blockwert

Werden die bereits in SysML definierten *Werttypen* als Typ eines *Blockwertes* verwendet, so besitzen diese initiale Belegungen (siehe Tabelle 3.1), falls es in der Konfiguration für das System nicht anders angegeben ist. Eine Besonderheit ist der *Werttyp Complex*, dessen Wert sich aus den Werten seiner *Eigenschaften* (`realPart` und `imaginaryPart` vom Typ *Real*) zusammensetzt. Des Weiteren kann der *Werttyp String* genau wie ein `std::string` aus C++ verwendet werden.

- Wert ändern: Einen neuen Wert `aValue` zuweisen.

```

1 aValue = "aValue";

```

- Wert verwenden: Den Wert von `aValue` an die Standardausgabe leiten.

```

1 std::cout << aValue << std::endl;

```

- Werte vergleichen: Den Wert von `aValue` mit einem *String* vergleichen.

```

1 // if-then-else
2 if (aValue == "aValue")
3     true;
4 else
5     false ;
6 // short if-then-else
7 aValue == "aValue" ? true : false ;

```

- Operationsaufruf: Den Wert von `aValue` als Operationsparameter für `operationValue` verwenden und den *Rückgabewert* in `aValue` speichern.

```

1 aValue = aValue->operationValue(aValue);

```

### 3.4.3 Mehrwertige Blockeigenschaft

- Instanz erstellen: Eine Instanz von `A` erstellen und der mehrwertigen Blockeigenschaft `aMultivalued` zuordnen.

```
1  A* a = new A();  
2  aMultivalued->insert(a);
```

- Instanz löschen: Eine Instanz von `A` aus der *Blockeigenschaft* `aMultivalued` entfernen und diese löschen.

```
1  aMultivalued->erase(a);
```

- Instanz ändern: Eine Instanz von `A` aus der *Blockeigenschaft* `aMultivalued` ermitteln und den Wert von dessen *Blockeigenschaft* `aValue` verändern.

```
1  aMultivalued->find(a)->aValue = "aValue";
```

- Operationsaufruf: Die Instanzen von `aMultivalued` als Operationsparameter für `operationMultivalued` verwenden und den *Rückgabewert* in `aMultivalued` speichern.

```
1  aMultivalued = aMultivalued->operationMultivalued(&aMultivalued);
```

### Operatoren für Mengentypen

Die Tabelle 3.3 zeigt die erlaubten *Operationen* zum Modifizieren von mehrwertige *Blockeigenschaften* abhängig vom Mengentyp (siehe Tabelle 3.2). Anhand des angegebenen Containers aus der C++ Standard Template Library (siehe „The C++ Programming Language: Special Edition“ von Bjarne Stroustrup [1]) können weitere Operatoren ermittelt und verwendet werden, solange diese ausschließlich lesend auf den Container zugreifen.

| Mengentyp            | Operator  | Container   |
|----------------------|-----------|-------------|
| OrderedSet, Sequence | insert    | std::vector |
| Set, Bag             | insert    | std::set    |
| OrderedSet, Sequence | erase     | std::vector |
| Set, Bag             | erase     | std::set    |
| OrderedSet, Sequence | swap      | std::vector |
| Set, Bag             | swap      | std::set    |
| OrderedSet, Sequence | clear     | std::vector |
| Set, Bag             | clear     | std::set    |
| OrderedSet, Sequence | push_back | std::vector |
| OrderedSet, Sequence | pop_back  | std::vector |

Tabelle 3.3: Mengentypen mehrwertiger Blockeigenschaften



### 3.5 Konfigurationsbeschreibung

Die Konfigurationsbeschreibung muss in einem separaten Paket erfolgen, welches den Stereotypen `<<Simulation>>` zugeordnet bekommt (siehe Abbildung 3.4). Für die Ausführung des Systems (Simulation) kann eine Start- und/oder eine

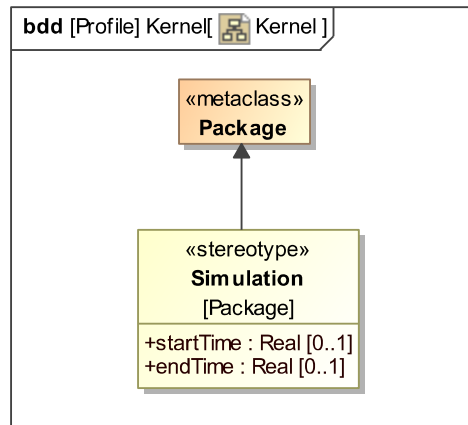


Abbildung 3.4: Initiale Systemkonfigurationsdefinition

Endzeit angegeben werden. Die Startzeit legt die Zeit fest mit der die Simulation startet. Die Endzeit bestimmt die Zeit zu der die Simulation beendet werden soll (*Zeitereignis*). Das Beenden der Simulation kann auch direkt in einer Verhaltensbeschreibung mit dem Aufruf `odemx::simml::Block::exitSimulation()` erfolgen. In der *Zustandsmaschine* eines aktiven *Blockes* kann ein *Zeit-* oder *Zustandsereignis* diesen Aufruf bewirken.

Zusätzlich zu den Angaben der Simulationskonfiguration müssen in dem Paket mit dem Stereotyp `<<Simulation>>` die initialen Blockinstanzen und -werte des System angegeben werden.

- **Blockteil-/referenz**  
Die Instanzen können als Vorbelegung mit *Blockeigenschaften* verbunden werden. Der Name der Instanz erscheint hinter einem Zuweisungszeichen der *Blockeigenschaft*.
- **Blockwert**  
Initiale Werte von *Blockwerten* können ebenfalls festgelegt werden und überlagern Vorgabewerte des *Wertetyps*. Die Angabe erfolgt mit einem Zuweisungszeichen hinter dem *Blockwert*.
- **Mehrwertige Eigenschaft**  
Die Zuordnung von Instanzen zu mehrwertigen *Eigenschaften* erfolgt genau wie die des *Blockteils/-referenz* mit dem Unterschied, dass mehrere Instanzen mit einer *Eigenschaft* verbunden sein können. Falls die *Eigenschaft* eine

geordnete Menge ist, werden die Namen der Instanzen kommasepariert aufgezählt (Reihenfolge der Zuordnung).

## 3.6 Erläuterungen zu SimML

Nachfolgend wird erklärt, aus welchen Gründen einzelne Einschränkungen und Erweiterungen gegenüber SysML bzw. UML getroffen wurden (zusätzlich zu den Variationspunkten).

Grundsätzlich ist zu erwähnen, dass viele Entscheidungen aus pragmatischen Gründen und zur Vereinfachung der Sprache in Bezug auf die Anzahl der Sprachelemente getroffen wurde.

### 3.6.1 Namensgebung

Die Beschränkung der gültigen Namen von Modellierungselementen ist in der Abbildung der Bezeichner in die Zielsprache C++ begründet. Möglich wäre ebenso eine Transformation von beliebigen Namen in gültige SimML-Bezeichner ähnlich wie in „Abbildung von SDL-200 nach Java“ von Toby Neumann [20] dargestellt.

### 3.6.2 Strukturbeschreibung

SimML erlaubt in einem Modell, nur ein *Paket* ohne Stereotyp und weitere *Pakete*, die den Stereotypen Simulation tragen. Das liegt daran, dass *Pakete* bei der Abbildung nicht berücksichtigt werden, da sie keinen Mehrwert für Art der darstellbaren Systemtypen bieten.

Außerdem dürfen Typdefinitionen nur die Sichtbarkeit *public* besitzen, da auch dies keinen Mehrwert in Bezug auf die Ziele der Arbeit stellt.

Für *Block*- oder *Werttypeigenschaften* wird ein Typ verlangt, da die Abbildung der *Eigenschaft* ohne Typangabe unvollständig im Zielmodell wäre.

### 3.6.3 Verhaltensbeschreibung

SimML unterstützt lediglich Kernmodellierungselemente für *Zustandsmaschinen*, die als eine Erweiterung eines endlichen Automaten definiert sind. Andere Verhaltensbeschreibungen (z. B. *Aktivitäten*) können ebenso mächtig sein und stellen somit nur eine andere Darstellungsform des gleichen Sachverhaltes dar. Die Kernkonzepte beinhalten unter anderem das Warten auf Zeit- oder Zustandsereignisse. Diese Konzepte finden sich auch in der „Simulation Language with Extensibility (SLX)“ wieder [6]. SLX besitzt die Aktion `waituntil`, die als Parameter eine Bedingung erwartet. Die Bedingung kann ein Vergleich mit der Modellzeit sein oder über Zustandsvariablen definiert sein.

Eine weitere Modellierungssprache, die „Specification and Description Language (SDL)“ [21], erlaubt die Beschreibung von diskretem Verhalten mit erweiterten endlichen Zustandsautomaten. Die Erweiterungen beziehen sich auf zusätzliche

Modellierungselemente, die mit Hilfe von anderen Kernmodellierungselementen dargestellt werden könnten.

### 3.6.4 Port-Kommunikation

Es existieren in SimML keine Modellelemente zur direkten Beschreibung von Ports und dem Austausch von Instanzen mit Hilfe derer. Prinzipiell ist das Empfangen einer Instanz an einem Port ein *Zustandsereignis*, welches sehr wohl von SimML unterstützt wird. Eine portbasierte Kommunikation könnte mit Hilfe von *Blockeigenschaften* realisiert werden. Diese könnten eine empfangene Instanz aufnehmen und über ein *Zustandsereignis*, welches die Änderung der *Blockeigenschaft* beinhaltet, könnte der Empfang signalisiert werden.

### 3.6.5 Zustandsereignis

Für *Zustandsereignisse* muss der Stereotyp `<<AnnotatedChangeEvent>>` verwendet werden, da die *Bedingung* in C++ formuliert ist und bekannt sein muss, welche *Eigenschaften* beteiligt sind, damit Änderungen an diesen zu einer neuen Auswertung der *Bedingung* führen. Durch Hinterlegung der *Eigenschaften* ist es somit nicht nötig, die *Bedingung* in C++ zu Parsen, um die beteiligten *Eigenschaften* zu identifizieren.



## 4 ODEMx-Erweiterungen

In diesem Kapitel werden die Erweiterungen erklärt, die an der Simulationsbibliothek ODEMx vorgenommen wurden. Das Ziel dieser Erweiterungen ist es, die anschließende Beschreibung der Abbildung zu vereinfachen. Aus diesem Grund erfolgt die prinzipielle Abbildung der Zustandsmaschinen auf ODEMx-Prozesse. Nachfolgend wird zunächst eine kurze Einführung in die prozessorientierte Simulation mit ODEMx gegeben. Das Ziel ist es, lediglich die für diese Arbeit relevanten Konzepte für ODEMx kurz einzuführen. Eine detaillierte Beschreibung befindet sich in „Objektorientierte Prozeßsimulation in C++“ von Joachim Fischer und Klaus Ahrens [8].

### 4.1 Prozessorientierte Simulationsmodelle in ODEMx

Prozesse, der prozessorientierten Simulation mit ODEMx, werden mit Hilfe der Klasse `odemx::base::Process` beschrieben. Um einen Prozess zu definieren, muss von der Klasse geerbt werden und die virtuelle Funktion `main` eine Implementierung besitzen. Die Funktion beschreibt den Lebenszyklus eines Prozesses, der zu diskreten Zeitpunkten Änderungen an Zustandsvariablen vornimmt. Die Abfolge von Aktionen erfolgt zeitlos (in Bezug auf die Modellzeit) bis auf ein Zustands- oder Zeitereignis gewartet wird. Während ein Prozess auf das Eintreten eines Ereignisses wartet, können andere Prozesse Aktionen durchführen und ebenfalls Zustandsvariablen verändern. Das Simulationsprogramm führt immer nur einen Prozess aus bzw. arbeitet seinen Lebenszyklus ab, bis dieser explizit die Steuerung abgibt oder implizit durch das Warten auf ein Ereignis diese verliert. Es existiert ein Ereigniskalender, in dem alle Zeitereignisse und aktiven Prozesse eingetragen sind. Beim Abgeben der Steuerung erhält nun der Prozess die Steuerung, der zu diesem Zeitpunkt im Ereigniskalender eingetragen ist (bei mehreren entscheidet die Reihenfolge der Einträge im Ereigniskalender über die Ausführungsreihenfolge). Erst wenn kein Prozess mehr für einen Zeitpunkt Aktionen ausführen kann wird die Modellzeit fortgeschritten bis zu dem Zeitpunkt an dem wieder ein Eintrag im Ereigniskalender vorhanden ist. Die Simulation endet, falls für die Simulation ein Endzeitpunkt angegeben wurde und dieser erreicht wird oder ein Prozess an der Simulation die Funktion `exitSimulation` aufruft.

#### 4.1.1 Warten auf Ereignisse

Prozesse können während ihres Lebenslaufes auf ein oder mehrere Ereignisse warten, um weitere Aktionen erst nach dem Eintreten des Ereignisses zu realisieren.

In ODEMX kann ein Prozess auf mehrere Objekte vom Typ `odemx::sync::Memory` gleichzeitig warten und unterbricht seinen Lebenszyklus durch Aufruf der Funktion `Process::wait`. Eines der registrierten Memory-Objekt kann das Fortsetzen des Prozess-Lebenszyklus durch Aufrufen der Funktion `Memory::alert()` veranlassen. Daraufhin wird der Prozess zum aktuellen Modellzeitpunkt in den Ereigniskalender eingetragen und kann seinen Lebenszyklus nach dem Erhalt der Steuerung fortsetzen.

#### 4.1.2 Zeitereignisse

Zeitereignisse können durch Objekte der Klasse `odemx::sync::Timer` ausgelöst werden. Ein Prozess kann mit der Funktion `Process::wait` und einem `Timer` auf ein Zeitereignis warten.

Eine weitere Möglichkeit, um auf ein Zeitereignis zu warten, bietet der Aufruf der Funktion `Process::holdFor` (Warten bis zum Ablauf einer Zeitspanne) oder `Process::holdUntil` (Warten bis zum übergebenen Zeitpunkt). Der Aufruf führt zur Unterbrechung des Prozesses bis das Zeitereignis eintritt. Ein gleichzeitiges Warten auf verschiedene Ereignisse ist mit Hilfe dieser Funktionen nicht möglich.

#### 4.1.3 Zustandsereignisse

Neben dem Warten auf einen `Timer`, bietet die Funktion `Process::wait` prinzipiell die Möglichkeit auf ein beliebiges Objekt vom Typ `Memory` zu warten, z. B. auf Zustandsereignisse, die an die Formulierung von Zustandsbedingungen geknüpft sind. Die Funktion `Memory::alert` könnte dann z. B. beim Erfüllen einer Zustandsbedingung automatisch aufgerufen werden. ODEMX bietet hierfür zum Zeitpunkt dieser Diplomarbeit keine vordefinierten Klassen, die das Prüfen und Beobachten von Zustandsbedingungen ermöglichen. Unabhängig von diesem Konzept ist es lediglich möglich auf Zustandsereignisse zu reagieren, die bei der zeitkontinuierlichen Änderung von Zustandsvariablen geprüft werden. Dieses Konzept verwendet nicht die Klasse `odemx::sync::Memory` und ist daher nicht in Verbindung mit der Funktion `Process::wait` verwendbar. Der nächste Abschnitt gibt dazu einen kurzen Einblick und zeigt anschließend, wie das existierende Konzept in Kombination mit der Klasse `odemx::sync::Memory` zu einem neuen Konzept vereinigt werden kann, damit es prinzipiell möglich wird, auf Zeit- und Zustandsereignisse gleichzeitig zu warten.

#### Zeitkontinuierliche Zustandsänderung

Zeitkontinuierliche Zustandsänderungen werden in ODEMX mit der Klasse `odemx::continuous::Continuous` beschrieben und können mit Hilfe von `odemx::base::continuous::Monitor` beobachtet werden. An diesem Monitor kann ein Objekt der Klasse `odemx::base::continuous::StateEvent` registriert werden. Objekte dieser Klasse sollen Zustandsereignisse beschreiben, die

durch Objekte der Klasse `odemx::continuous::Continuous` ausgelöst werden. Die virtuelle Funktion `StateEvent::condition` bietet die Möglichkeit eine Zustandsbedingung zu implementieren und liefert einen booleschen Wert (Auswertung der Zustandsbedingung). Mit Hilfe des Monitors wird während der zeitkontinuierlichen Änderung von Zustandsvariablen die Funktion `StateEvent::condition` aufgerufen und das Eintreten eines Zustandsereignisses geprüft. Ist die Zustandsbedingung erfüllt, so wird die virtuelle Funktion `StateEvent::action` aufgerufen, in der Aktionen beim Eintritt eines Zustandsereignisses festgelegt sind.

## 4.2 Überblick der ODEMX-Erweiterungen

Die Erweiterungen, die in den Modulen `Synchronization` und `SimML` der ODEMX-Bibliothek erstellt wurden, stehen zueinander in Beziehung. Sie können in verschiedene Schichten aufgeteilt werden, wobei die Elemente einer Schicht nur in direkter Beziehung zu den Elementen der angrenzenden Schichten stehen. Die Abbildung 4.1 zeigt diese Schichten in unterschiedlichen Farben. Die Elemente der Schichten sind die Klassen, die die Erweiterungen der Bibliothek darstellen.

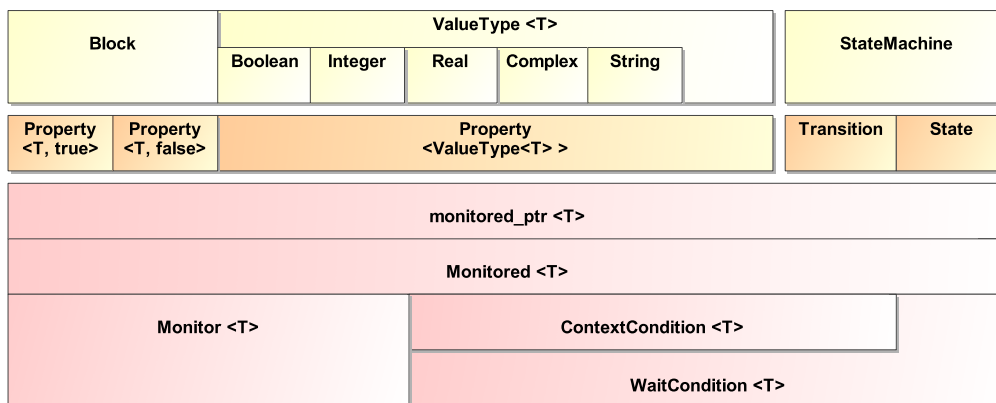


Abbildung 4.1: Überblick der ODEMX-Erweiterungen

## 4.3 ODEMX-Erweiterungen im Modul: Synchronization

Die im Abschnitt 4.1.3 vorgestellte Klasse für Zustandsereignisse kann in Verbindung mit der zugehörigen Monitor-Klasse ausschließlich für die Überwachung von zeitkontinuierliche Zustandsänderungen verwendet werden. Für den Fall, dass jedoch auf ein Zustandsereignis gewartet werden soll, welches von einer oder mehreren Zustandsvariablen, die zeitdiskret verändert werden, abhängig ist, gibt es

bisher kein Zustandsereignis- und Monitorkonzept in ODEMX. Der nächste Abschnitt beschreibt diese Neuerung.

### 4.3.1 Zeitdiskrete Zustandsänderung

Die Klasse `odemx::sync::WaitCondition` ist eine Spezialisierung der Klasse `odemx::base::continuous::StateEvent`. Die in der Superklasse bereits definierten Funktionen, sollen auch in der abgeleiteten Klasse für die Überprüfung des Eintretens und der Aktion beim Eintreten eines Zustandsereignisses verwendet werden (siehe Listing 4.1).

```
1 bool WaitCondition::isAvailable () {  
2     return ( condition (getSimulation ().getTime()));  
3 }  
4  
5 void WaitCondition::action () {  
6     removeWaitConditionAtAll();  
7     alert ();  
8 }
```

Listing 4.1: Überprüfen der Zustandsbedingung und Fortsetzen des Prozesslebenszyklus

Es ist daher möglich die formulierten Zustandsbedingungen auch für zeitkontinuierliche Zustandsänderungen zu verwenden. Zusätzlich ist die Klasse eine Spezialisierung der Klasse `odemx::sync::Memory`, damit diese in Verbindung mit der Funktion `Process::wait` für zeitdiskrete Prozesse verwendet werden kann. Ein Prozess kann somit beim Eintreten eines Zustandsereignisses wieder in den Ereigniskalender eingetragen werden.

Kommt es zu einer Änderung an einer in der Zustandsbedingung beteiligten Variablen, müssen die zugehörigen Zustandsbedingungen ausgewertet und beim Erfüllen der Bedingung muss der wartende Prozess wieder in den Ereigniskalender eintragen werden.

### Beispiel

Listing 4.2 zeigt ein Beispiel für die Verwendung von `odemx::sync::WaitCondition`. Der Prozess `p1` wartet auf das Erfüllen der Zustandsbedingung, die in `W::condition` hinterlegt wurde. Der Prozess `p2` ändert nach dem Eintreten des Zeitereignisses durch den Timer `t` einen Wert, der in der Zustandsbedingung von `p1` auftritt. Dies führt dazu, dass die Zustandsbedingung von `p1` erfüllt ist und der Prozess seinen Lebenszyklus fortsetzt.

```
1 #include <odemx/odemx.h>  
2 using namespace odemx;  
3 using namespace odemx::base;  
4 using namespace odemx::data;  
5 using namespace odemx::sync;  
6
```



```

7 class P1;
8
9 class W: public WaitCondition {
10     P1* p1;
11 public:
12     W(Simulation& s, const Label& l, P1* p1) :
13         WaitCondition(s, l, NULL), p1(p1) {}
14     virtual bool condition(SimTime t);
15 };
16
17 class P1: public Process {
18 public:
19     int i;
20     W w;
21     P1(Simulation& s, const Label& l) :
22         Process(s, l), i(0), w(s, "w: W", this) {}
23     int main();
24 };
25
26 class P2: public Process {
27 public:
28     P1* p1;
29     P2(Simulation& s, const Label& l, P1* p1) :
30         Process(s, l), p1(p1) {}
31     int main();
32 };
33
34 bool W::condition(SimTime t) {
35     return p1->i != 0;
36 }
37
38 int P1::main() {
39     wait(&this->w);
40     return 0;
41 }
42
43 int P2::main() {
44     Timer t(getSimulation(), "t: Timer");
45     t.setIn(1);
46     wait(&t);
47     p1->i = 1;
48     if (p1->w.isAvailable())
49         p1->w.action();
50     return 0;
51 }
52
53 int main() {
54     Simulation& s = getDefaultSimulation();
55     P1 p1(s, "p1: P1");
56     P2 p2(s, "p2: P2", &p1);
57     p1.activate();
58     p2.activate();
59     s.run();
60     return 0;

```

61 }

Listing 4.2: Beispiel der Verwendung der Klasse `odemx::sync::WaitCondition`

Zum Modellzeitzeitpunkt 0 beginnen die Prozesse `p1` (Zeile 42) und `p2` (Zeile 46) ihren Lebenslauf. Der Aufruf von `Process::wait` von `p1` führt dazu, dass der Prozess seinen Lebenszyklus unterbricht (Zeile 43). Das Listing 4.3 zeigt die Aufruf-Kaskade zum Merken des Prozesses in `Memory` und unterbrechen des Prozesslebenslaufes.

```

1 Process::wait (...)
2   -> if (!WaitCondition::isAvailable ())
3     -> Memory->->remember(...)
4     -> Process::sleep()
5     -> Simulation::switchTo()

```

Listing 4.3: Aufruf-Kaskade der Prozessunterbrechung

Der Prozess `p2` verändert an Prozess `p1` dann den Wert des Attributes `i`. Anschließend veranlasst er die Überprüfung der Bedingung `w` des Prozesses `p1` und ruft im Positivfall `WaitCondition::action` auf (Zeile 51-52). Der Prozess `p1` wird daraufhin reaktiviert. Im Listing 4.4 ist die Funktionsaufrufkaskade dargestellt.

```

1 if (WaitCondition::isAvailable ())
2   -> WaitCondition::Action()
3   -> Memory::alert()
4   -> if (!checkSchedForAlert (...))
5     -> Process::alertProcess (...)
6     -> Scheduler::insertSched (...)

```

Listing 4.4: Aufruf-Kaskade für die Prozessreaktivierung

Ein Nachteil der Klasse `odemx::sync::WaitCondition` ist, dass die Implementierung der Zustandsbedingung in der geerbten Funktion `StateEvent::condition` im Kontext einer Klasse erfolgen muss, die die entsprechenden Zustandsvariablen definiert. Diese Kontextklasse kann aber nicht von `odemx::sync::WaitCondition` erben, da sonst nur eine Zustandsbedingung pro Prozess existieren könnte. Sie besitzt daher Objekte der Klasse `odemx::sync::WaitCondition`, deren Zustandsbedingungen aber einen Zugriff auf `private` Attribute haben. Eine Lösung wäre die Angabe `friend` im Prozess für jede verwendete Ableitung von `odemx::sync::WaitCondition`. Aus diesem Grund wurde die Template-Klasse `odemx::sync::ContextWaitCondition` hinzugefügt. Diese ermöglicht es ein Zeiger auf ein Objekt einer beliebigen Klasse (Template-Parameter) und auf eine Funktion der Klasse festzulegen. Die Funktion muss den Rückgabewert und Parameter der Funktion `base::continuous::StateEvent::condition` erfüllen, da diese an dem festgelegtem Objekt gerufen wird (siehe Listing 4.5).

```

1 template<typename T> bool ContextWaitCondition<T>::condition(base::SimTime time) {
2     assert (getContext());
3     assert (getContextCondition());
4     return (getContext() -> *getContextCondition())(time);
5 }

```

Listing 4.5: Kontextabhängige Zustandsbedingung

### 4.3.2 Monitor-Konzept

Die Klasse `odemx::sync::WaitCondition` besitzt zusätzliche Probleme. Kommt es zu einer Änderung an einer Zustandsvariablen, müssen alle Objekte der Klasse `odemx::sync::WaitCondition`, die auf diese Zustandsvariable Bezug nehmen, durch Aufruf der Funktion `WaitCondition::condition` geprüft werden müssen. Hierfür wird das Kennen aller beteiligten Zustandsbedingungen beim Ändern einer Zustandsvariablen vorausgesetzt. Diese Probleme werden mit der Klasse `odemx::sync::Monitor` gelöst. Diese verwaltet eine Menge von registrierten Objekten der Klasse `odemx::sync::WaitCondition`. Kommt es zu einer Änderung des Wertes der Variablen, so kann dem zugehörigem Monitor über die Funktion `Monitor::checkWaitConditions` dies signalisiert werden. Daraufhin überprüft dieser die zugeordneten Zustandsbedingungen und ruft bei Erfüllung die Funktion `StateEvent::action`.

### Beispiel

Das Listing 4.6 zeigt die Veränderungen des Beispiels aus Listing 4.2. Die Klasse `P1` erhält ein Attribut `m` vom Typ `odemx::sync::Monitor` (Zeile 5), an der sich das Objekt `w` mit dem Aufruf der Funktion `Monitor::addWaitCondition` registriert (Zeile 8). Das Prüfen der Zustandsbedingung nach der Änderung an den Zustandsvariablen durch den Prozess `p2` wird mit dem Funktionsaufruf `Monitor::checkWaitConditions` realisiert (Zeile 17).

```

1 class P1: public Process {
2 public:
3     int i;
4     W w;
5     Monitor m;
6     P1(Simulation& s, const Label& l) :
7         Process(s, l), i(0), w(s, "w: W", this) {
8         m.addWaitCondition(&w);
9     }
10    int main();
11 };
12 int P2::main() {
13     Timer t(getSimulation(), "t: Timer");
14     t.setIn(1);
15     wait(&t);
16     p1->i = 1;

```

```
17 p1->m.checkWaitConditions();
18 return 0;
19 }
```

Listing 4.6: Beispiel der Verwendung der Klasse `odemx::sync::Monitor`

### 4.3.3 Erweitertes Monitor-Konzept

Änderungen an Attributen einer beliebigen Klasse müssen, wie im vorherigen Abschnitt beschrieben, explizit dem Monitor durch Aufruf der Funktion `Monitor::checkWaitConditions` mitgeteilt werden, falls diese an der Zustandsbedingung beteiligt sind. Damit dieser Aufruf implizit stattfindet, kann eine Klasse die Template-Klasse `odemx::sync::Monitored` spezialisieren. Sie besitzt Operator-Überladungen für den Zuweisungsoperator und einen Monitor, dessen Funktion `Monitor::checkWaitConditions` nach jeder Zuweisung aufgerufen wird, um Änderungen bezüglich der Zustandsbedingung zu prüfen. Objekte auf dem Stack, deren Klassendefinition diese spezialisieren, können somit beobachtet werden.

Falls Objekte auf dem Heap oder Werte von primitiven Typen betroffen sind, ist es jedoch nicht möglich bei Änderungen informiert zu werden, da der Zuweisungsoperator für diese in C++ nicht überladen werden kann.

Dieses Problem wurde mit Hilfe des Smart-Pointer-Konzepts (Kapselung durch eine Klassendefinition, die einen Zeiger besitzt, der auf das Objekt oder den Wert zeigt) gelöst. Hierfür wurde die Klasse `odemx::sync::monitored_ptr` definiert, die den Zuweisungsoperator überladet, um solche Änderungen mitzubekommen und den Monitor zu informieren.

#### Beispiel

Das Listing 4.7 zeigt die Unterschiede zu dem Beispiel aus Listing 4.2, wenn für das Attribut `P1::i` nicht der Typ `int`, sondern die Klasse `odemx::sync::monitored_ptr<int>` verwendet.

```
1 class P1: public Process {
2 public:
3     monitored_ptr<int> i;
4     P1(Simulation& s, const Label& l);
5     int main();
6 };
7 P1::P1(Simulation& s, const Label& l) :
8     Process(s, l), i(new int(0)) {
9 }
10 int P1::main() {
11     W w(getSimulation(), "w: W", this);
12     i.getMonitor().addWaitCondition(&w);
13     Timer t(getSimulation(), "t: Timer");
14     t.setIn(1);
15     Process::wait(&w, &t);
16     i.getMonitor().removeWaitCondition(&w);
```

```

17  return 0;
18  }
19  int P2::main() {
20      p->i = 1;
21      return 0;
22  }

```

Listing 4.7: Beispiel der Verwendung der Klasse `odemx::sync::monitored_ptr`

Das Objekt vom `odemx::sync::WaitCondition` in Zeile 15 kann mit dem Monitor des Attributes `P1::i` verbunden werden. Die Änderung an diesem Attribut in Zeile 24 bewirkt damit, dass der Monitor an allen registrierten Objekten den Aufruf `WaitCondition::checkWaitConditions` vornimmt. Dies führt dazu, dass `p1` wieder in den Ereigniskalender eingetragen wird (Aufrufkaskade in Listing 4.8).

```

1  monitored_ptr<T>::operator=(...)
2      -> Monitored<T>::operator=(...)
3      -> getMonitor().checkWaitConditions()

```

Listing 4.8: Aufruf-Kaskade der Monitor-Überwachung

## 4.4 ODEMX-Erweiterungen im Modul: SimML

Dieses neue Modul für ODEMX beinhaltet die Klassen, die benötigt werden, um eine möglichst strukturäquivalente Abbildung von SimML-Modelle in ODEMX-Modelle zu erhalten. Für die Abbildung von *Zustandsmaschinen* wurde das State-Pattern aus [10] verwendet.

### 4.4.1 Klasse Block

Die Klasse `Block` besitzt Funktionen und Attribute, um die Semantik eines *Blockes* aus SimML zu realisieren (siehe Listing 4.9).

```

1  class Block {
2  public:
3      Block();
4      Block(Behavior* classifierBehavior );
5      virtual ~Block();
6      Behavior* getClassifierBehavior ();
7      virtual void _run(base::SimTime time, const std::vector<long>* cases);
8      virtual bool
9      _check(base::SimTime time, const std::vector<long>* cases) const;
10     void exitSimulation ();
11
12     Block** getOwner();
13     void setOwner(Block** const owner);
14     static void setAssociationNULL(Block** const association);
15     static void setAssociation (Block** const association , Block* const value);
16     void addReferencer(Block** const referencer );

```

```
17 void removeReferencer(Block** const referencer);
18
19 private:
20     Behavior* classifierBehavior;
21     std::list<Block**> referencers;
22     Block** owner;
23 };
```

Listing 4.9: Klasse Block in ODEMX

Um eine Instanz mit einer Verhaltensbeschreibung zu verbinden, kann diese per Konstruktor übergeben werden. Verhaltensbeschreibungen werden mit der Klasse `odemx::simml::StateMachine` beschrieben und sind ODEMX-Prozesse. Die virtuellen Funktionen `Block::_run` und `Block::_check` werden zum Ausführen von Aktionen und Prüfen von Bedingungen im Blockkontext benötigt, da die Verhaltensbeschreibung in einer separaten Klasse erfolgt, dies aber ein Abbildungsdetail ist und in der Verhaltensbeschreibung in SimML von der Blockdefinition als Kontext ausgegangen wird. Wenn eine Verhaltensbeschreibung hinterlegt ist, so ruft die Funktion `Block::exitSimulation` die Funktion `odemx::base::Simulation::exitSimulation`, welche über die Prozessklasse erreichbar ist und die Simulation beendet.

#### 4.4.2 Klasse Property

Ein Objekt der Klasse `Block` kann mit anderen Objekten der Klasse `Block` verbunden sein. Besteht eine Teil-Ganzes-Beziehung, so muss der Besitzer am besitzenden Objekt mit `Block::setOwner` gespeichert werden. Andernfalls muss der Referenzierer sich mit der Funktion `Block::addReferencer` registrieren. Wechselt ein Objekt seinen Besitzer oder wird gelöscht, so kann beim Besitzer oder Referenzierer dies signalisiert werden.

Die Verbindung zu einem Objekt der Klasse `Block` kann als Attribut vom Typ `odemx::simml::Property` angelegt werden. Diese Template-Klasse besitzt zwei Template-Parameter, zum einen den Typ des zu verbindenden Objektes und zum anderen die Art der Verbindung (Teil-Ganzes- oder Referenzierende-Beziehung). Die Template-Klasse besitzt partielle Template-Spezialisierungen in Abhängigkeit des zweiten Template-Parameters. Diese beinhalten die Überladung des Zuweisungsoperators und können so speziell auf die Art der Beziehung Besitzer oder Referenzierer über eine Änderung des verbundenen Objektes informieren.

Die Klasse ist außerdem eine Spezialisierung von `odemx::sync::monitored_ptr`. Das Objekt, welches mit der Klasse `odemx::simml::Property` verbunden werden soll, ist somit über einen Smart-Pointer erreichbar. Dies sichert das Bestehen eines Objektes, falls es nicht in einer Teil-Ganzes-Verbindung steht und der Referenzierer gelöscht wird. In diesem Fall wird das Objekt der Klasse `odemx::simml::Property` gelöscht, nicht aber das über den Smart-Pointer erreichbare Objekt.

### 4.4.3 Klasse ValueType

Um *Wertetypen* aus SimML abbilden zu können, wird eine Klasse benötigt, deren Objekte Wertesemantik besitzen. Die in C++ vorhandenen primitiven Typen sind hierfür nicht ausreichend, da zum einen strukturierte *Wertetypen* definierbar sein müssen und zum anderen Wertänderungen beobachtbar sein müssen (siehe Abschnitt 4.3.1). Die Klasse `odemx::simml::ValueType<typenameT>` bietet all diese Eigenschaften. Die Wertesemantik wird durch die Definition des Operators `operatorT()` realisiert. Wertetypdefinitionen können somit diese Klasse spezialisieren und über den Typ-Parameter den Typ für den Wert der Objekte der Klasse bestimmen. Die vordefinierten *Wertetypen* besitzen zum Teil partielle Template-Spezialisierung dieser Klasse, so z. B. der *Wertetyp String*. Dieser kann auf Grund dessen überall dort verwendet werden, wo die Klasse `std::string` verlangt wird, da der Wert eines Objektes dieser Klasse nicht in dem vorgesehenem Attribut, sondern in `std::string` selbst gespeichert wird.

### 4.4.4 Klasse StateMachine

Die angewandte Methode zur Beschreibung einer *Zustandsmaschine* ist angelehnt an das Muster Quantum Platform Event Processor (QEP) aus „Practical Statecharts in C/C++“ von Miro Samek [10]. QEP ist aus den Abbildungsverfahren der geschachtelten Switch-Anweisungen, Zustandstabellen und des State-Patterns (siehe „Design Patterns with Applying UML and Patterns and Iterative Development“ von Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides, Craig Larman [4]) hervorgegangen.

Das Muster der geschachtelten Switch-Anweisungen wählt auf der obersten Switch-Ebene den *Zustand* aus. Darunter finden sich für jeden *Zustand* eine Switch-Anweisung mit jeweils einem Fall für ein mögliches *Ereignis*. Der aktuelle *Zustand* wird in einer Variablen gespeichert und beim Eintreten eines *Ereignisses* wird in den passenden Fall der Switch-Anweisung gesprungen. Dort erfolgt über die nächste Switch-Anweisung die Behandlung des Ereignisses. Das Ändern der Zustandsvariablen stellt den Zustandswechsel dar.

Die Abbildung von *Zustandsmaschinen* mit Hilfe einer Zustandstabelle besitzt ein zweidimensionales Feld, das horizontal die möglichen *Zustände* und vertikal die Ereignisse kodiert. Der Inhalt eines Eintrages beschreibt dann die Änderung beim Eintreten eines Ereignisses. Zusätzlich wird auch bei diesem Verfahren der Zustand in einer Variablen gespeichert.

Das State-Pattern ist ein objektorientierter Ansatz zur Beschreibung von *Zustandsmaschinen*. Bei diesem Verfahren sind *Zustände* Objekte, wobei für jeden *Zustand* eine Klassendefinition existieren muss, innerhalb derer Funktionen beschrieben sind, die die *Transitionen* dieses *Zustandes* beschreiben. Alle Klassendefinitionen besitzen eine gemeinsame Basisklasse, so dass die Klassendefinition der *Zustandsmaschine* ein Attribut dieser Basisklasse besitzt, um den aktuellen *Zustand* d. h. das entsprechende Objekt zu speichern.

Das Verfahren von QEP ist ähnlich dem des State-Patterns mit dem Unterschied, dass ein *Zustand* kein Objekt ist, sondern durch eine Funktion repräsentiert wird. Der aktuelle *Zustand* wird als Zeiger auf die entsprechende Funktion gespeichert. Die Verwendung ist somit auf Sprachen wie C/C++ oder andere, die Funktionszeiger als Teil der Sprache besitzen, beschränkt.

Das Abbildungsmuster mit QEP wurde dahingehend verändert, dass die Beschreibung eines aktiven *Blockes* mit Hilfe eines Prozesses aus ODEMX erfolgen kann. Prinzipiell kann ein aktiver *Block* in SimML eine Verhaltensbeschreibung besitzen, daher besitzt die Klasse `Block` einen Zeiger auf ein Objekt der Klasse `Behavior`. Eine Spezialisierung der Klasse kann dadurch eine beliebige Verhaltensbeschreibungsart sein. Die Klasse erbt von `odemx::base::Process`, so dass ein Objekt der Verhaltensbeschreibung alle Funktionalitäten eines Prozesses besitzt. `odemx::simml::StateMachine` ist eine Spezialisierung von `odemx::simml::Behavior` und soll als Abbildungsgrundlage für *Zustandsmaschinen* aus SimML dienen. Der Vorteil der separaten Klasse für die *Zustandsmaschine* ist, dass die generierten Attribute und Funktionen bei der Abbildung eines SimML-Modells keinen Konflikt mit den im Modell definierten und abgebildeten Attributen und Funktionen des *Blockes* verursachen. Diese führt allerdings zu dem Nachteil, dass die Aktionen von *Effekten* nicht ohne weitere Veränderung in die generierte Zustandsmaschinenklasse, da der Bezug der verwendeten Attribute und Funktionen nicht mehr korrekt ist. Der nachfolgende Abschnitt geht daher auf die Lösung dieses Problems ein und im Anschluss wird die Implementierung der Klasse `odemx::simml::StateMachine` erklärt.

## Blockkontextfunktionen

Die Klasse `odemx::simml::Block` stellt zwei virtuelle Funktionen bereit. Eine dient der Auswertung von *Bedingungen* (`StateMachine::ContextConstraint`) und eine weitere dem Ausführen von *Verhalten* (`StateMachine::ContextBehavior`) innerhalb des Blockkontextes (siehe Abschnitt 4.4.1). Die Zustandsmaschinenklasse besitzt zwei Funktionszeiger auf diese Blockkontextfunktionen und die Basisklasse `odemx::simml::Behavior` hat das Attribut `Behavior::context`, so dass eine Veränderung oder Prüfung von *Blockeigenschaften* eines *Blockes* möglich ist. Die Funktionen besitzen verschiedene Teile, die jeweils eine Ausführungs- oder Prüfungssequenz (Menge von zusammengehörigen Anweisungen) entsprechen. Damit die richtige Sequenz ausgeführt wird, besitzen die Funktionen einen Parameter, der eine Vektor von Zahlen ist. Die Position im Vektor codiert die Ebene und der Wert den Fall der Ebene für eine geschachtelte Switch-Anweisungen. Im Fall der Verhaltensbeschreibungsart der *Zustandsmaschine* werden nur zweidimensionale Vektoren benötigt. Die erste Dimension codiert den Zustand und die zweite die Transition (siehe `Transition::Cases`).



## Zustand

Ein *Zustand* wird durch eine Funktion in einer Ableitung der Klasse `odemx::simml::StateMachine` dargestellt. Diese darf weder einen Rückgabewert, noch einen Parameter besitzen (Funktionszeigertyp `typedef void(StateMachine::*State)()`). Der aktuelle Zustand wird im Funktionszeiger `StateMachine::state` gespeichert. In der Funktion werden ausgehenden *Transitionen* im Attribut `StateMachine::possibleTransitions` abgelegt. Die Auswahl einer *Transition* zum eingetretenen Ereignis findet im Lebenszyklus der *Zustandsmaschine* statt (siehe 4.4.4).

## Pseudozustand

SimML definiert die *Pseudozustände*: *Initialknoten*, *Terminierungsknoten* und *Entscheidungsknoten*. Für den *Initialknoten* existiert die rein virtuelle Funktion `StateMachine::initial` und der *Terminierungsknoten* wird mit `StateMachine::StateStateMachine::terminate=NULL` repräsentiert. Der *Entscheidungsknoten* muss eine Funktion besitzen, die die gleichen Eigenschaften wie die Funktion für einen *Zustand* besitzt und muss innerhalb dieser Funktion das Attribut `StateMachine::pseudoState` auf `PseudoState::CHOICE` setzen. Dies ist nötig, damit im Lebenszyklus der *Zustandsmaschine* anhängig vom aktuellen Knoten (*Zustand* oder *Pseudozustand*) die Ausführungssemantik realisiert werden kann.

## Transition

Die *Transitionen* einer *Zustandsmaschine* werden mit der Klasse `odemx::simml::Transition` beschrieben. Eine *Transition* wird mit Hilfe des Konstruktors spezifisch für einen *Trigger* erstellt. Es gibt hierfür Konstruktoren für *Transitionen* ohne *Trigger* oder mit einem *Trigger* für das Warten auf ein Zeitergebnis oder auf ein Zustandsereignis. Die Erstellung erfordert außerdem, dass ein Funktionszeiger auf den Nachfolgezustand angegeben wird. Zusätzlich kann die Codierung für einen *Effekt* oder einen *Wächter*, die in der entsprechenden Blockkontextfunktion ausgewertet wird (siehe Abschnitt Blockkontextfunktionen), übergeben werden.

## Lebenszyklus

Der Lebenszyklus einer *Zustandsmaschine* ist in der rein virtuellen Funktion `Process::main` und im Listing 4.10 dargestellt.

```

1 int StateMachine::main() {
2     while (getState() != terminate) {
3         (this->*(getState()))();
4         Transition* transition = fireTransition ();
5         if (! transition )
6             Process::sleep ();

```

```
7  else
8      (getContext() ->*(contextBehavior))(getCurrentTime(),
9          transition ->getParameter());
10     setState( transition ->getTarget());
11     setPseudoState(NONE);
12     deletePossibleTransitions ();
13 }
14
15 return 0;
16 }
```

Listing 4.10: Abarbeitung der Zustandsmaschine

Der Lebenszyklus wird so lange ausgeführt bis der *Pseudozustand* `terminate` erreicht wird (Zeile 2). Ist dies nicht der Fall, so wird die Funktion aufgerufen, die den aktuellen Zustand beschreibt (Zeile 3). Diese ermittelt alle ausgehenden Transitionen und wählt mit Hilfe der Funktion `StateMachine::fireTransition` die *Transition* aus, die als nächstes verwendet werden soll (Zeile 4).

Die Funktion prüft im Falle eines *Entscheidungsknotens* die *Wächter* und liefert die *Transition*, dessen *Wächterbedingung* erfüllt ist. Ist dies für mehrere der Fall, so wird willkürlich eine ausgewählt.

Im Falle eines *Zustandes* werden zuerst die *Transitionen* ohne *Trigger* und falls vorhanden deren *Wächter* behandelt. Ist keine *Wächterbedingung* erfüllt, wird mit Hilfe der Prozessfunktionalität `Process::wait` auf Ereignisse für die *Trigger* der übrigen Transitionen gewartet. Da diese Prozessfunktionalität nur einen *Trigger*, dessen Ereignis eingetreten ist, als Ergebnis liefert, werden anschließend alle *Transitionen* mit erfüllten *Trigger* auf deren *Wächterbedingung* geprüft. Bei Mehrdeutigkeit, kommt es wieder zur Auswahl einer Beliebigen. Ist keine *Wächterbedingung* erfüllt, wird erneut mit Hilfe der Prozessfunktionalität auf ein Ereignis gewartet. Falls es von vornherein keine *Transition* mit einem *Trigger* oder die *Wächterbedingung* von solchen nicht erfüllt ist, so wird die Funktion ohne Rückgabe einer verwendbaren *Transition* verlassen.

Der Rückgabewert der Funktion `StateMachine::fireTransition` wird geprüft und falls dieser ungültig ist, so wird der Prozess unterbrochen (Zeile 6).

Falls eine Nachfolgetransition ausgewählt wurde und besitzt die *Transition* einen *Effekt*, so wird dieser aufgerufen (Zeile 8-9). Hierbei wird die Codierung des *Effektes* für die Blockkontextfunktion `StateMachine::contextBehavior` verwendet. Anschließend wird der aktuelle *Zustand* auf den der *Transition* folgenden gesetzt (Zeile 10), das Attribut zum setzen eines *Pseudozustandes* zurückgesetzt (Zeile 11), die ausgehenden Transitionen gelöscht (Zeile 12) und die Abarbeitung fortgesetzt.

## 5 Abbildung eines SimML-Modells nach ODEmx

Die vorherigen Kapitel umfassten die Beschreibung der Sprache SimML (inklusive der Bezüge zu SysML bzw. UML) und die Neuerungen der ODEmx-Bibliothek. Das folgende Kapitel gibt die Abbildungsregeln an, um ein SimML-Modell in ein ODEmx-Modell zu transformieren. Es wird gezeigt, wie die in SimML festgelegte Semantik sich in dem ausführbaren Simulationsmodell widerspiegelt.

### 5.1 Abbildungsregeln

Die in diesem Kapitel gezeigten Abbildungsregeln sind in einer Erweiterung der Sprache MOF Model To Text Transformation Language (MOFM2T) [11] beschrieben. Die Erweiterung umfasst die Verwendung von Java als Beschreibungssprache für Operationen. Die Sprache dient der Definition von Transformationsregeln, die auf Basis eines MOF-Metamodells (z. B. das UML-Metamodell) beschreiben, wie entsprechende Modelle in Text umgewandelt werden.

Der erste Abschnitt beschäftigt sich mit der Modellprüfung vor der eigentlichen Transformation. Dies beinhaltet die Validierung eines SimML-Modells anhand von OCL-Ausdrücken und stellt sicher, dass das Modell den Anforderungen der Sprache SimML genügt und eine Transformation erfolgreich durchgeführt werden kann. Der folgende Abschnitt führt die Abbildung der Strukturbeschreibung aus SimML vor. Dies betrifft die Typdefinition *Block*. Im zweiten Abschnitt wird die Abbildung der Verhaltensbeschreibung in Form der *Zustandsmaschine* aus SimML gezeigt. Anschließend können Objekte anhand der generierten Klassenbeschreibungen angelegt und miteinander in Beziehung gesetzt werden. Zusätzlich wird die Abbildung der Simulationsausführungseigenschaften gezeigt, so dass ein ausführbares C++-Programm generiert werden kann.

### 5.2 Validierung

Die Validierung eines Modells beinhaltet die Prüfung (Validierungsfunktion) der eingeschränkten Syntax von SimML gegenüber SysML. Die Regeln wurden bereits in Kapitel 3 in Form von OCL-Bedingungen (Validierungsregeln) formuliert und können direkt in MTL verwendet werden. Die Validierung wird vor jeder Modelltransformation ausgeführt und erst bei erfolgreicher Prüfung wird die Transformation gestartet. Eine Auflistung der Regeln befindet sich im Anhang.

## 5.3 Strukturabbildung

Die Abbildung der Struktur beschränkt sich auf die Generierung von Klassen, deren Attribute und Operationen anhand der Typbeschreibungen im SimML-Modell. Konkret bedeutet dies, dass alle *Blöcke*, die im Modell definiert wurden, die folgenden Transformationsregeln durchlaufen. Zunächst werden die Transformationsregeln erklärt und anschließend folgt ein abstraktes Beispiel für die Abbildung der Strukturbeschreibung.

### 5.3.1 Block-Abbildungsregeln

Jede Blockdefinition führt zu einer Generierung der Klassendeklaration (.h-Datei) und der Klassendefinition (.cpp-Datei). Dies ermöglicht eine in C++ bewährte Vorgehensweise zur Trennung von Deklarationen und Definitionen. Die Deklarationen können einzeln in anderen Deklarationen eingebunden werden ohne die komplette Definition erneut einzubinden.

#### Block-Header-Datei

Das Listing 5.1 zeigt die Transformationsregeln pro Blockdefinition für die Generierung der C++-Header-Datei.

```

1  [template public BlockH(block : Class) post (format())]
2  #ifndef [block.name.toUpper()/]_H_
3  #define [block.name.toUpper()/]_H_
4
5  #include <[getODEMXSimMLClass('/', 'IncludeBlock')/].h>
6  [for (include : Type | getIncludes(block))]
7  #include "[include.name/].h"
8  [/for]
9  [if block.isActive]
10 #include "[block.classifierBehavior.name/].h"
11 [/if]
12 [for (forward : Type | getForwardDeclaration(block))]
13 class [forward.name/];
14 [/for]
15
16 class [block.name/]: public virtual [getODEMXSimMLClass(':', 'Block')/][for (general :
    Classifier | getGeneralClassifiers(block))], public virtual [general.name/][for] {
17 public:
18     [block.name/]();
19     virtual ~[block.name/]();
20
21     [if block.isActive]
22     [block.name/]([block.classifierBehavior.name/] * classifierBehavior );
23     virtual void
24         _run(odemx::base::SimTime time, const std::vector<long>* cases);
25     virtual bool
26         _check(odemx::base::SimTime time, const std::vector<long>* cases) const;
27
28     [/if]

```

```

29 [for (property : Property | block.attribute)]
30   [if not property.ocllsTypeOf(Port)]
31     [getODEMxSimMLPackage('::')/::[property.getPropertyElementType()/]<[property.
32       getPropertyParameter()/]>[property.getMultivaluedTyp()/] [property.name/];
33   [/if]
34 [/for]
35 [for (operation : Operation | block.getOperations())]
36   [if operation.isStatic] static [/if] [operation.getOperationDeclaration()/] [if operation.
37     isQuery]const[/if];
38 [/for]
39 [if (getPropertyWithDefaultValue(block)->size() > 0)]
40 private :
41   void _setDefaultValues();
42
43 [/if]
44 };
45
46 #endif /* [block.name.toUpper()/]_H_ */
47
48 [/template]

```

Listing 5.1: Abbildungsregeln für die Block-Header-Datei

**Präprozessor-Anweisungen** In der Datei für die Klassendeklarationen wird zunächst die Definition von Präprozessor-Anweisungen festgelegt (Zeile 2-3). Diese sollen durch Mehrfacheinbindung der Header-Datei die Mehrfachdeklaration verhindern.

**Einbindung von Deklarationen** Es folgt die Ermittlung der Einbindung der vordefinierten Block-Header-Datei (Zeile 5) aus ODEMX. Diese enthält die Deklaration der Klasse `odemx::simml::Block` und der vordefinierten *Wertetypen*. Es folgt die Einbindung der benötigten Header-Dateien aller Superklassen, der Typen von *Blockeigenschaften* und der Verhaltensbeschreibung (Zeile 6-11), falls es sich um einen *aktiven Block* handelt. Die Verhaltensbeschreibung wird, wenn vorhanden, für die Deklarationen eines zusätzlichen Konstruktors benötigt. Falls eine Klasse verwendet wird, die die im folgenden zu generierende Klasse benutzt, werden Forward-Deklarationen festgelegt (Zeile 12-14).

**Klassendeklaration** Für einen *Block* aus SimML wird eine Klasse mit dem Namen des *Blockes* (siehe Namensgebung in 3.1) generiert, die die Klasse `odemx::simml::Block` spezialisiert. Die Vererbungsbeziehungen der *Blöcke* aus SimML spiegeln sich in der Vererbung der generierten Klassen wieder. Damit eine generierte Klasse auf Grund Ihrer Superklassen nicht mehrfach die Klasse `odemx::simml::Block` spezialisiert, ist diese Spezialisierung `virtual` (Zeile 16). Kommt es dazu, dass eine Klasse über verschiedene Wege mehrfach geerbt wird, dann

liegt sie im Klassenlayout nur einmal vor. Das gleiche gilt für die Mehrfachvererbung von *Blöcken*. Die Vererbung hat die Sichtbarkeit `public`, so dass eine generierte Klasse immer den Basistyp `odemx::simml::Block` besitzt, um typkonform an anderen Stellen verwendet zu werden.

**Konstruktor- und Destruktor-Deklaration** Es wird ein parameterloser Konstruktor und ein Destruktor generiert (Zeile 18-19). Zusätzlich wird ein Konstruktor generiert, der als Parameter eine Verhaltensbeschreibung verlangt, falls es sich um einen *aktiven Block* handelt (Zeile 22). In diesem Fall werden ebenfalls die beiden Kontextfunktionen zum Prüfen von Bedingungen und zum Ausführen von Anweisungen im Blockkontext generiert (Zeile 23-26).

**Klassenattribute** Die Abbildung der *Blockteile*, *-referenz* und *Blockwerte* wird durch die Verwendung der Klasse `template<typename T, const bool isComposite=true>odemx::simml::Property` realisiert, falls es sich nicht um eine mehrwertige *Blockeigenschaft* handelt. Bei mehrwertigen *Blockeigenschaften* muss der Mengentyp anhand weiterer Eigenschaften ermittelt werden (siehe Abschnitt 3.2.4). Der erste Template-Parameter wird mit dem Typ der *Blockeigenschaft* belegt. Der zweite Parameter erhält den Wert `true`, falls es sich um ein *Blockteil* oder *Blockwert* handelt und `false` bei einer *Blockreferenz* (Zeile 31) (dies gilt ebenso für die Instanzen mehrwertiger *Blockeigenschaften*). Die Teil-Ganzes-Beziehung für *Blockteile* und das Auflösen von Referenzen, falls das Objekt gelöscht wird, kann sichergestellt werden.

**Operationsdeklaration** Die Operationen eines *Blockes* sind für dessen Header-Datei insofern relevant, als dass eine Funktionsdeklaration generiert werden muss (Zeile 36). Anhand der Eigenschaft *statisch* wird die *Operation* als `static` und anhand der Eigenschaft *abfragend* als `const` deklariert. Es wird geprüft, ob die *Operation* einen *Rückgabewert* und *Parameter* besitzt. Falls vorhanden, wird der Typ eines *Rückgabewertes* als Funktionsrückgabetyt verwendet. Für jeden *Parameter* wird mit Hilfe des Parameternamens und des Parametertyps ein Funktionsparameter generiert.

**Vorgabewerte-Funktionsdeklaration** Abschließend wird die Funktion `_setDefaultValues` deklariert, die zum Setzen von Vorgabewerten verwendet wird (Zeile 41).

### Block-Implementierungsdatei

Die Implementierung der generierten Block-Klasse wird mit den im Listing 5.2 angegebenen Transformationsregeln beschrieben.

```
1 [template public BlockCPP(block : Class) post (format())]  
2 #include "[block.name/].h"  
3 [if block.isActive ]
```

```

4  [ if not block . classifierBehavior . getAppliedStereotype( 'StateMachine::
    StateMachineWithDependencies').oclIsUndefined() ]
5  [ for ( includeClassifier : Element | getTaggedValues(block . classifierBehavior , '
    StateMachine::StateMachineWithDependencies', 'dependencies')) ]
6  #include "[ includeClassifier ->filter( Classifier ).name/] .h"
7  [/for]
8  [/if]
9  [/if]
10
11 [let propertyWithDefaultValue : Set(Property) = getPropertyWithDefaultValue(block)]
12 [block.name/]::[block.name/](): [getODEMXSimMLClass('::', 'Block')/]() {
13 [ if (getPropertyWithDefaultValue(block)->size() > 0) ] [block.getDefaultValueFunction()/]();
14 [/if]
15 }
16 [block.name/]::~[block.name/]() {
17 }
18 [ if block . isActive ]
19
20 [block.name/]::[block.name/]([block . classifierBehavior . name/] * classifierBehavior ): [
    getODEMXSimMLClass('::', 'Block')/]( classifierBehavior ) {
21 [ if (getPropertyWithDefaultValue(block)->size() > 0) ] [block.getDefaultValueFunction()/]();
22 [/if]
23 }
24
25 void [block.name/]::_run(odemx::base::SimTime time,
26     const std::vector<long>* cases){
27     [generateSwitchCase(block, false)/]
28 }
29
30 bool [block.name/]::_check(odemx::base::SimTime time,
31     const std::vector<long>* cases) const {
32     [generateSwitchCase(block, true)/]
33     return [getODEMXSimMLClass('::', 'Block')/]:_check(time, cases);
34 }
35
36 [/if]
37
38 [for (operation : Operation | block.getOperations())]
39 [operation . getOperationDeclaration()/] [ if operation . isQuery ]const[/if]{
40 [ if operation . method->forAll(oclIsKindOf(OpaqueBehavior))]
41 [operation . method.oclAsType(OpaqueBehavior)._body/]
42 [/if]
43 }
44
45 [/for]
46
47 [ if (propertyWithDefaultValue->size() > 0) ]
48 void [block.name/]::[block.getDefaultValueFunction()/]() {
49 [ for (property : Property | propertyWithDefaultValue) ]
50 [ if not property . default . oclIsUndefined() ]
51 [property . getNamespaceName(property.namespace)/] = [property.default/];
52 [else]
53 [ for (defaultValue : ValueSpecification | property . defaultValue) ]
54 [generateInstanceValue( defaultValue , property . getNamespaceName(property.

```

```

53         namespace), property.type, property.namespace)/]
54     [/for]
55 [/if]
56 }
57 [/if]
58 [/let]
59
60 [/template]

```

Listing 5.2: Abbildungsregeln für die Block-Implementierungsdatei

**Einbindung von Deklarationen** In der Datei der Klassendefinition werden die verwendeten Typen eingebunden. Zeile 2 zeigt die Einbindung der zugehörigen Block-Klassendeklaration. Des Weiteren werden alle Typen eingebunden, die von lokalen Variablen verwendet werden (Zeile 4-9). Lokale Variablen aus einer eventuell vorhandenen Verhaltensbeschreibung werden ermittelt, falls die Beschreibung mit Hilfe einer *Zustandsmaschine* geschieht, die den Stereotyp <<StateMachineWithDependencies>> anwendet (siehe Abbildung 3.2).

**Konstruktor- und Destruktordefinition** Die Block-Klassendefinition enthält die Definition des Default-Konstruktors (Zeile 12). Der Konstruktor einer generierten Klasse ruft immer implizit alle parameterlosen Konstruktoren der Basisklassen und es bedarf daher hierfür keiner Transformationsregel. Die Basisklasse `odemx::simml::Block` bildet jedoch eine Ausnahme. Diese wird nur dann parameterlos gerufen, falls keine Verhaltensbeschreibung vorhanden ist. Ist der *Block aktiv*, so wird ein weiterer Konstruktor definiert (Zeile 20). Dieser wird mit einem Objekt der zugehörigen, generierten Klasse für die *Zustandsmaschine* als Parameter aufgerufen (Näheres siehe Abschnitt 5.4). Im Konstruktor wird die Funktion zum Erstellen der Vorgabewerte gerufen, falls Vorgabewerte existieren (Zeile 12 und Zeile 21). Zeile 16 enthält die Definition des Destruktors, der keine Anweisungen enthält, da alle Objekte der Klasse auf dem Stack erzeugt werden und der Speicher bei Objektvernichtung nicht explizit freigegeben werden muss.

**Blockkontextfunktionen** Die Blockkontextfunktion `_run` dient dem Ausführen von Anweisungen im Blockkontext (Zeile 24-27). Dies wird benötigt, da Anweisungen der Verhaltensbeschreibung mit dem Kontext des *Blockes* beschrieben sind und für die Verhaltensbeschreibung in eine eigene Klasse generiert wird. Aus dieser kann dann über den Blockkontext die Blockkontextfunktion aufgerufen werden und anhand der übergebenen Parameter (siehe Abschnitt 3.3.1) die gewünschte Anweisung der Verhaltensbeschreibung ausgeführt werden. Das Gleiche gilt für die Prüfung von *Bedingungen* der *Wächter* oder der *Trigger*, welche ebenfalls im Blockkontext formuliert sind. Hierfür steht die Blockkontextfunktion `_check` zur Verfügung (Zeile 29-33).



**Operationsdefinition** Weiterhin wird für jede Operationsdeklaration eine Operationsdefinition generiert (Zeile 37-42). Anhand der Eigenschaft *abfragend* wird die *Operation* als `const` definiert. Diese enthält die Operationsbeschreibung, die bereits in C++ formuliert sein muss (siehe Abschnitt 3.2.5).

**Vorgabewerte-Funktionsdefinition** Für *Blockeigenschaften* können Vorgabewerte im Modell festgelegt werden. Die zu erzeugenden Instanzen für einen Vorgabewert müssen in der Blockfunktion `_setDefaultValues` angelegt werden (Zeile 45-57). Die Generierung einer Block-Klasse sieht dabei lediglich die Generierung eines Attributs vom Typ `odemx::simml::Property` oder `odemx::simml::MultiplicityElement` für *Blockeigenschaften* vor. Die eigentliche Block- oder Wertetyp-Instanz ist als Attribut dieser Klasse hinterlegt, welches auf dem Heap erzeugt werden muss. Daher existieren nach Aufruf des Konstruktors der generierten Block-Klasse keine Instanzen für dessen *Blockeigenschaften*. Die Blockfunktion `_setDefaultValues` wird innerhalb des Konstruktors gerufen und enthält generierte Anweisungen zum Erstellen von Instanzen für hinterlegte Vorgabewerte.

### 5.3.2 Beispiel für die Abbildung von Blöcken

Die Abbildung 5.1 zeigt ein *Blockdefinitionsdiagramm* in SimML. Dieses abstrakte Beispiel soll dazu dienen, einzelne Aspekte der Strukturabbildung vorzuführen und die Semantik von SimML-Modellen in C++ demonstrieren. Hierfür folgen der Modellbeschreibung Ausschnitte aus dem generierten ODEMX-Modell und ein Testprogramm. Dieses wurde mit dem Framework Unit++ beschrieben (siehe „Unit++“ von Claus Draeby [3]), welches es ermöglicht Unittests in C++ zu formulieren.

#### Modellbeschreibung

Das Modell enthält vier *Blöcke*, die in einer Vererbungshierarchie stehen. Der *Block* `DerivedDerivedBlock` stellt eine Besonderheit in dem Modell dar, da er von zwei *Blöcken* erbt, die wiederum von ein und dem selben *Block* erben. `DerivedBlock1` und `DerivedBlock2` besitzen beide *Blockreferenzen* mit dem Typ des jeweils anderen *Blockes*. Der *Block* `BaseBlock` besitzt einen *Blockwert* vom Typ `Integer` mit einem Vorgabewert. `DerivedDerivedBlock` kann in der *Sequence* `derivedDerivedProperty` *Blockteile* verwalten. Über die *Operation* `add` wird eine Instanz zu dieser *Sequence* hinzugefügt werden. Der Rückgabewert der *Operation* gibt an, ob die Instanz hinzugefügt werden konnte. Zusätzlich besitzt der *Block* ein *Blockteil* auf eine Instanz vom Typ `BaseBlock`.

#### ODEMX-Modell

Das Listing 5.3 zeigt Ausschnitte, der mit Hilfe der Abbildungsregeln generierten ODEMX Header- und Implementierungsdateien des *Blockes*.

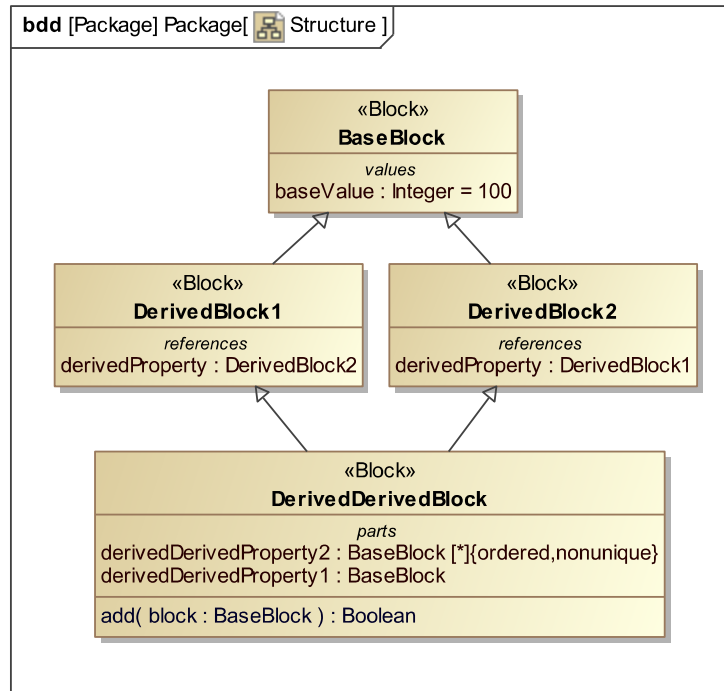


Abbildung 5.1: Blockdefinitionsdiagramm

```

1 // BaseBlock.h
2 class BaseBlock: public virtual odemx::simml::Block {
3 public:
4     BaseBlock();
5     virtual ~BaseBlock();
6     odemx::simml::Property<Integer, true> baseValue;
7 private:
8     void _setDefaultValues();
9 };
10 // BaseBlock.cpp
11 void BaseBlock::_setDefaultValues() {
12     baseValue = 100;
13 }
14 // DerivedBlock1.h
15 class DerivedBlock1;
16 class DerivedBlock2: public virtual odemx::simml::Block,
17     public virtual BaseBlock {
18 public:
19     DerivedBlock2();
20     virtual ~DerivedBlock2();
21     odemx::simml::Property<DerivedBlock1, false> derivedProperty;
22 };
23 // DerivedBlock2.h
24 class DerivedBlock2;
  
```

```

25 class DerivedBlock1: public virtual odemx::simml::Block,
26     public virtual BaseBlock {
27 public:
28     DerivedBlock1();
29     virtual ~DerivedBlock1();
30     odemx::simml::Property<DerivedBlock2, false> derivedProperty;
31 };
32 // DerivedDerivedBlock.h
33 class DerivedDerivedBlock: public virtual odemx::simml::Block,
34     public virtual DerivedBlock1,
35     public virtual DerivedBlock2 {
36 public:
37     DerivedDerivedBlock();
38     virtual ~DerivedDerivedBlock();
39     odemx::simml::Sequence<BaseBlock*, true>::Type derivedDerivedProperty2;
40     odemx::simml::Property<BaseBlock, true> derivedDerivedProperty1;
41     Boolean add(BaseBlock* block);
42 };
43 // DerivedDerivedBlock.cpp
44 Boolean DerivedDerivedBlock::add(BaseBlock* block) {
45     int size = derivedDerivedProperty2.size();
46     derivedDerivedProperty2.push_back(block);
47     return size != derivedDerivedProperty2.size();
48 }

```

Listing 5.3: Ausschnitte der generierten Header- und Implementierungsdateien des Blockes

### Testprogramm

Das Testprogramm (siehe Listing 5.4) erstellt eine Instanz von `DerivedDerivedBlock`, die als globale Variable verfügbar ist und ruft dann die definierten Testfälle auf. Diese konnten alle erfolgreich durchlaufen werden.

```

1 DerivedDerivedBlock* ddb;
2 int main() {
3     ddb = new DerivedDerivedBlock();
4     return UnitTest::RunAllTests();
5 }

```

Listing 5.4: Testprogramm des Beispiels der Strukturabbildung

**Test: Vorgabewert** Der erste Test prüft, ob der Vorgabewert für `baseValue` gesetzt wurde.

```

1 TEST(DefaultValue){
2     CHECK(ddb->baseValue == 100);
3 }

```

Listing 5.5: Test: Vorgabewerte

**Test: Wertetyp** Anhand einer Kopie eines *Wertes* eines *Wertetyps* wird geprüft, ob eine Änderung an der Kopie den Original-Wert nicht beeinflusst.

```
1 TEST(ValueType){
2   ddb->baseValue = 0;
3   Integer i(ddb->baseValue);
4   i = i + 1;
5   CHECK(ddb->baseValue != i);
6 }
```

Listing 5.6: Test: Wertetyp

**Test: Mehrfachvererbung** Die mehrfach geerbte Eigenschaft kann über die verschiedenen Basen angesprochen und verändert werden. Im Objektlayout liegt immer nur eine Instanz vor und eine Änderung kann über alle Basen abgefragt werden.

```
1 TEST(MultipleInheritance){
2   ddb->DerivedBlock2::baseValue = 0;
3   ddb->DerivedBlock1::baseValue = 1;
4   CHECK(ddb->DerivedBlock2::baseValue == ddb->DerivedBlock1::baseValue);
5 }
```

Listing 5.7: Test: Mehrfachvererbung

**Test: Operationsaufruf** Der Operationsaufruf erhält als Übergabeparameter eine Instanz, die zu einer *Sequence* hinzugefügt werden soll. Der Erfolg der *Operation* wird über den Rückgabewert mitgeteilt und kann somit geprüft werden.

```
1 TEST(OperationCall){
2   BaseBlock* bb = new BaseBlock();
3   CHECK(ddb->add(bb));
4 }
```

Listing 5.8: Test: Operationsaufruf

**Test: Mengentypen** Die *Sequence* kann eine weitere Instanz aufnehmen. Anschließend kann überprüft werden, ob diese in der geordneten Menge als letztes Element eingefügt wurde.

```
1 TEST(MultiplicityElement){
2   BaseBlock* bb = new BaseBlock();
3   bb->baseValue = 101;
4   ddb->derivedDerivedProperty2.push_back(bb);
5   CHECK(ddb->derivedDerivedProperty2.back()->baseValue == bb->baseValue);
6 }
```

Listing 5.9: Test: Mengentypen

**Test: Kompositionsbeziehung** Das Entfernen einer Instanz mit einer *Blockreferenz* führt nicht zum Löschen der referenzierten Instanz. Bei einem *Blockteil* hingegen muss die Instanz ebenfalls gelöscht sein.

```

1 TEST(Composition){
2   DerivedBlock1* db1 = new DerivedBlock1();
3   DerivedBlock2* db2 = new DerivedBlock2();
4   db1->derivedProperty = db2;
5   db2->derivedProperty = db1;
6   ddb->derivedDerivedProperty1 = db2;
7   delete db1;
8   db1 = NULL;
9   CHECK(db2->derivedProperty == NULL);
10  CHECK(ddb->derivedDerivedProperty1 != NULL);
11  db1 = new DerivedBlock1();
12  db1->derivedProperty = db2;
13  db2->derivedProperty = db1;
14  delete ddb;
15  ddb = NULL;
16  CHECK(db1->derivedProperty == NULL);
17 }

```

Listing 5.10: Test: Kompositionsbeziehung

## 5.4 Verhaltensabbildung

Die Verhaltensbeschreibung eines *Blockes* für zeitdiskrete Änderungen erfolgt mit einer *Zustandsmaschine*. Die Erkenntnisse aus dem Buch „Practical Statecharts in C/C++“ von Miro Samek [10] werden genutzt, um eine Abbildung zu beschreiben. In dem Buch wurde ein Rahmenwerkzeug in C++ in Form einer Bibliothek beschrieben, mit dessen Hilfe Ereignisse und deren Abarbeitung beschrieben und eine Abbildung einer UML-Zustandsmaschine in ein Programm in C++ vorgenommen werden kann, um diese auszuführen.

### 5.4.1 Zustandsmaschinen-Abbildungsregeln

Eine *Zustandsmaschine* wird auf die Klasse `odemx::simml::StateMachine` abgebildet, indem genau wie bei der Abbildung eines *Blockes* eine Klassendefinition mit dem Namen der *Zustandsmaschine* generiert wird, die die ODEMX-Klasse spezialisiert. Im Listing 4.10 wurde gezeigt, wie die Ausführung der *Zustandsmaschine* erfolgt.

#### Zustandsmaschinen-Header-Datei

Das nachfolgende Listing 5.11 zeigt die Abbildungsregel für die Header-Datei der *Zustandsmaschine*.

```
1 [template public StateMachineH(stateMachine : StateMachine) post (format())]
2 #ifndef [stateMachine.name.toUpper()/_H_]
3 #define [stateMachine.name.toUpper()/_H_]
4
5 #include <[getODEmXSimMLClass('/', 'IncludeStateMachine')]/.h>
6 #include "[stateMachine._context.name/]_h"
7
8 class [stateMachine.name/]: public [getODEmXSimMLClass(':', 'StateMachine')]/ {
9 public:
10     [stateMachine.name/]( [getODEmXSimMLClass(':', 'Block')]/* context,
11         const odemx::data::Label& label = "[stateMachine.name/]",
12         odemx::base::Simulation& sim = odemx::getDefaultSimulation(),
13         odemx::base::ProcessObserver* obs = 0);
14     virtual ~[stateMachine.name/](){};
15
16     virtual void initial ();
17     [let region : Region = stateMachine.getFirstRegion()]
18     [for (state : State | region.getStates())]
19     void [state.name/](){};
20     [/for]
21     [/let]
22 };
23
24 #endif /* [stateMachine.name.toUpper()/_H_] */
25
26 [/template]
```

Listing 5.11: Zustandsmaschinen-Header-Datei

**Präprozessor-Anweisungen** In der Datei für die Klassendeklarationen einer *Zustandsmaschine* werden genau wie bei der Klassendeklaration eines *Blockes* zunächst Präprozessor-Anweisungen zur Vermeidung von Mehrfachdeklarationen eingeführt (Zeile 2-3).

**Einbindung von Deklarationen** Damit die Klasse `odemx::simml::StateMachine` als Basis verwendet werden kann, muss die vordefinierte Zustandsmaschinen-Header-Datei eingebunden werden (Zeile 5).

Des Weiteren wird die Deklaration der Blockkontext-Klasse benötigt (Zeile 6), damit der Blockkontext, der über den Zeiger `odemx::simml::StateMachine::getContext()` erreichbar ist, vom Basistyp `odemx::simml::Block` auf die spezielle Blockkontextklasse getypst werden kann. Diese wird benötigt, um den Prozess der *Zustandsmaschine* für die Überwachung von Änderungen an *Blockeigenschaften* an diesen zu registrieren.

**Klassendeklaration** Für eine *Zustandsmaschine* wird eine Klasse mit dem Namen der *Zustandsmaschine* generiert, die die Klasse `odemx::simml::StateMachine` spezialisiert (Zeile 8). Die Basisklasse bietet die Grundfunktionalität für die Ausführung. Des Weiteren ist sie eine Spezialisierung der Prozess-

Klasse aus ODEmx, deren Instanzen im Rahmen einer Simulation ausgeführt werden.

**Konstruktor- und Destruktordeklaration** Zeile 10-14 zeigen die Transformationsregeln für die Konstruktor- und Destruktordeklarationen. Der Konstruktor besitzt alle Parameter der Basisklasse.

**Initialknotenfunktion** Jede *Zustandsmaschine* in SimML muss einen *Initialknoten* besitzen, für den eine Initialknotenfunktion generiert wird (Zeile 16).

**Zustandsfunktion** Jeder *Zustand* wird auf eine Zustandsfunktion abgebildet. Zusätzlich wird auch für den *Entscheidungsknoten* eine solche Zustandsfunktion generiert (Zeile 18-20). Der Hintergrund ist, dass all diese *Knoten* einer *Zustandsmaschine* *Trigger* und/oder *Wächter* besitzen dürfen, deren Prüfung auf die gleiche Weise mit Hilfe der Funktionalitäten der Basisklasse `odemx::simml::StateMachine` erfolgt. Für den *Terminierungsknoten* hingegen wird die Deklaration `odemx::simml::StateMachine::terminate` verwendet, da von diesem *Knoten* keine *Transitionen* ausgehen dürfen und der Prozess innerhalb der Simulation terminiert werden muss.

### Zustandsmaschinen-Implementierungsdatei

Im folgenden Listing 5.12 werden die Transformationsregeln zur Generierung der Klassendefinition beschrieben.

```

1  [template public StateMachineCPP(stateMachine : StateMachine) post (format())]
2  [let region : Region = stateMachine.getFirstRegion()]
3  #include "[stateMachine.name]/.h"
4
5  [stateMachine.name/]::[stateMachine.name/](odemx::simml::Block* context,
6      const odemx::data::Label& label, odemx::base::Simulation& sim,
7      odemx::base::ProcessObserver* obs) :
8      odemx::simml::StateMachine(context, sim, label, obs) {
9  }
10
11 [stateMachine.name/]::~[stateMachine.name/]() {
12 }
13
14 void [stateMachine.name/]:: initial () {
15     [region . getPseudostateInitial () . getVertexImplementation()/]
16 }
17
18 [for (state : State | region . getStates())]
19 void [stateMachine.name/]::[state . name/]() {
20     [state . getVertexImplementation()/]
21 }
22
23 [/for]
24 [/let]

```

25 `[/template]`

Listing 5.12: Abbildungsregeln für die Zustandsmaschinen-Implementierungsdatei

**Einbindung von Deklarationen** In der Datei der Klassendefinition wird zunächst die zugehörige Zustandsmaschinen-Klassendeklaration eingebunden (Zeile 3).

**Konstruktor- und Destruktordefinitionen** Zeile 5-12 enthalten die Generierungsvorschriften der Konstruktor- und Destruktordefinitionen. Der Konstruktor übergibt alle Parameter an die Basisklasse. Innerhalb derer findet die Erstellung eines ODEMX-Prozesses statt. Gleichzeitig wird nach der Erstellung im Basisklassenkonstruktor der Prozess für die Simulation aktiviert. Die Basisklasse besitzt nach dem Konstruktoraufufruf einen Zeiger auf die aktuelle Zustandsfunktion (am Beginn: Zustandsinitialfunktion), eine leere Liste zum Verwalten der ausgehenden *Transitionen* und ein Attribut, dass angibt, ob der aktuelle *Zustand* ein *Entscheidungsknoten* ist. Dies ist nötig, da die Aktivierung der zu durchlaufenden, ausgehenden *Transition* sich in diesem Fall unterscheidet.

**Zustandsfunktionen** Die Zustandsinitialfunktion (Zeile 14) unterscheidet sich dahingehend von den restlichen Zustandsfunktionen (Zeile 19), dass lediglich der Name der Funktion festgelegt ist und nur für *Zustände* der im Modell vorgegebene Name verwendet wird.

Die Definition der Funktionen beinhaltet die Erstellung von Instanzen der Klasse `odemx::simml::Transition` (Transitionsobjekt) abhängig von dem *Ereignis*, welches getriggert werden kann.

Eine Besonderheit stellt die Erzeugung eines Transitionsobjektes dar, welches auf ein *Zustandsereignis* triggert. Im Modell muss das *Zustandsereignis* mit dem Stereotypen `<<AnnotatedChangeEvent>>` versehen sein. Dieser besitzt ein *Tag variables*, mit dem *Blockeigenschaften* festgelegt werden können, die zum Auslösen des *Zustandsereignis* führen können. Jede *Blockeigenschaft* wird so abgebildet, dass es die Basisklasse `odemx::sync::Monitored<T>` besitzt, die ein Attribut vom Typ `odemx::sync::Monitor` hat. Alle Monitorobjekte, der für das *Zustandsereignis* relevanten *Blockeigenschaften*, müssen an den Konstruktor zum Erstellen des Transitionsobjektes übergeben werden. Dies sichert die Aktivierung des Prozesses und die Prüfung der *Bedingung* für das *Zustandsereignis* bei der Änderung einer relevanten *Blockeigenschaft*.

Das Ausführen von Anweisungen von *Effekten* oder die Prüfung von *Bedingungen* von *Wächtern* oder *Zustandsereignissen* wird mit Hilfe der Blockkontextfunktionen wie beschrieben realisiert. Hierfür ist lediglich beim Erstellen des Transitionsobjektes die Angabe des zugehörigen Parameters entscheidend.

Die Kodierung des Parameters `odemx::simml::Transition::Parameter` lässt



sich wie folgend aufschlüsseln. Der Parameter besitzt drei ganzzahlige Werte vom Typ `long`. Die erste Zahl kodiert den *Knoten*, die zweite die *Transition*. Die dritte Zahl kann den *Wächter* (Wert = 0) oder einen *Trigger* (Wert > 0) kodieren. Der Wert -1 besagt, dass es weder das eine noch das andere gibt und die *Transition* direkt durchlaufen werden kann.

#### 5.4.2 Beispiel für die Abbildung von Zustandsmaschinen

Die Abbildung 5.2 zeigt ein *Zustandsdiagramm* in SimML. Anhand dessen sollen genau wie bei der Strukturabbildung einzelne Aspekte der Verhaltensabbildung vorgeführt und die Semantik von SimML-Modellen in C++ demonstriert werden. Zunächst gibt es eine Modellbeschreibung und daran anschließend Ausschnitte aus dem generierten ODEMX-Modell und ein Testprogramm. Dieses wurde ebenfalls mit dem Framework Unit++ beschrieben. Diese Art der Tests dienen nicht von Hause aus der Prüfung von ausführbaren Prozessen. Es können aber Werte oder Instanzen zu einem Zeitpunkt der Ausführung getestet werden. Um die Ausführung schrittweise durchzuführen, wird die Funktion `odemx::base::Simulation::step()` verwendet. Diese lässt die Simulation bis zum nächsten im Ereigniskalender eingetragenen Objekt fortschreiten.

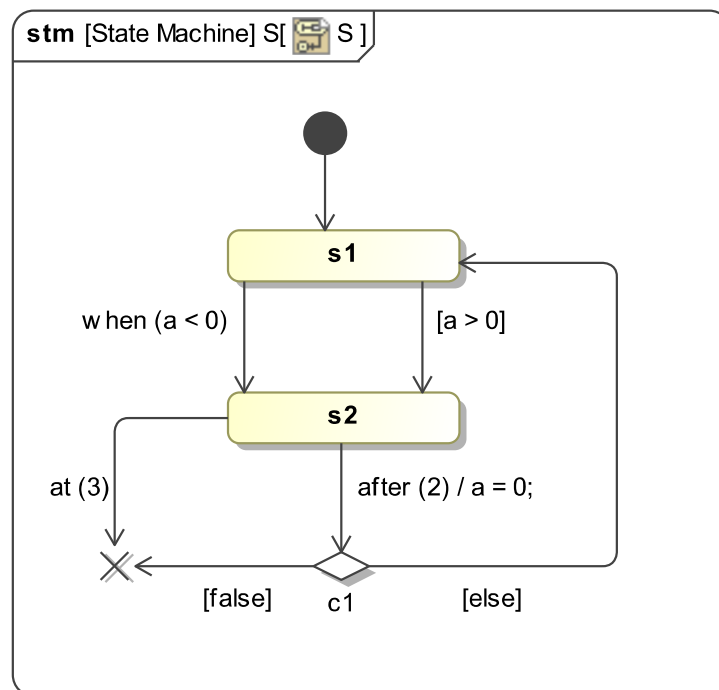


Abbildung 5.2: Zustandsdiagramm

## Modellbeschreibung

Das Modell besteht aus einem *aktiven Block* *A*, der ein *Blockwert* *a* vom Typ *Integer* mit Vorgabewert 1 besitzt (nicht dargestellt).

Die *Zustandsmaschine* des *Blockes* ist in der Abbildung 5.2 dargestellt. Sie besitzt zwei *Zustände*, einen *Initialknoten*, einen *Entscheidungsknoten* und einen *Terminierungsknoten*. Vom *Initialknoten* gelangt man direkt zum *Zustand* *s1*. Der Übergang kann sofort vollzogen werden. Von *s1* gelangt man zu *s2* bei Eintreten eines *Zustandsereignisses* oder ohne Ereigniseintritt (laut UML das sog. Completion-Event), allerdings muss hierfür die *Bedingung* des *Wächters* erfüllt sein. Zu einem absoluten *Zeitereignis* wird der *Zustand* *s2* verlassen und der *Terminierungsknoten* betreten. Tritt vorher das relative *Zeitereignis* ein, so wird der *Entscheidungsknoten* betreten und der festgelegte *Effekt* ausgeführt. Die *Bedingungen* der ausgehenden *Transitionen* können nur zum Übergang zu *Zustand* *s1* führen. Während des Übergangs wird auch hier der definierte *Effekt* wirksam.

## ODEmX-Modell

Das Listing 5.13 zeigt Ausschnitte, der mit Hilfe der Abbildungsregeln generierten ODEmX Header- und Implementierungsdateien der *Zustandsmaschine*.

```
1 // S.h
2 class S: public odemx::simml::StateMachine {
3 public:
4     S(odemx::simml::Block* context, const odemx::data::Label& label = "S",
5       odemx::base::Simulation& sim = odemx::getDefaultSimulation(),
6       odemx::base::ProcessObserver* obs = 0);
7     virtual ~S();
8
9     virtual void initial ();
10    void s2();
11    void s1();
12    void c1();
13 };
14 // S.cpp
15 void S::s2() {
16     // Trigger: 2
17     odemx::simml::StateMachine::addPossibleTransition (
18         new odemx::simml::Transition(odemx::base::Process::getSimulation(),
19             "time event", true, 2,
20             new odemx::simml::Transition::Parameter(2, 1, 1),
21             odemx::simml::StateMachine::Function<S>::Pointer(&S::c1)));
22     odemx::simml::StateMachine::addPossibleTransition (
23         new odemx::simml::Transition(
24             odemx::base::Process::getSimulation(),
25             "time event",
26             false,
27             3,
28             new odemx::simml::Transition::Parameter(2, 2, 1),
29             odemx::simml::StateMachine::Function<S>::Pointer(
30                 odemx::simml::StateMachine::terminate)));
```

```

31 }
32 }
33 void S::s1() {
34     // Trigger: 1
35     std::vector<odemx::sync::Monitor*>* monitors;
36     monitors = new std::vector<odemx::sync::Monitor*>();
37     monitors->push_back(
38         &dynamic_cast<A*>(odemx::simml::Behavior::getContext())->a.odemx::sync::
39             Monitored<
40                 Integer>::getMonitor());
41     odemx::simml::StateMachine::addPossibleTransition(
42         new odemx::simml::Transition(odemx::base::Process::getSimulation(),
43             "change event", odemx::simml::Behavior::getContext(),
44             monitors, new odemx::simml::Transition::Parameter(3, 1, 1),
45             odemx::simml::StateMachine::Function<S>::Pointer(&S::s2)));
46     odemx::simml::StateMachine::addPossibleTransition(
47         new odemx::simml::Transition(odemx::base::Process::getSimulation(),
48             "completion event",
49             new odemx::simml::Transition::Parameter(3, 2, -1),
50             odemx::simml::StateMachine::Function<S>::Pointer(&S::s2)));
51 }

```

Listing 5.13: Ausschnitte der generierten Header- und Implementierungsdateien der Zustandsmaschine

### Testprogramm

Das Testprogramm (siehe Listing 5.14) erstellt eine Instanz von `odemx::base::Simulation`, der Zustandsmaschine `S` und des Blockes `A`, die als globale Variablen verfügbar sind. Anschließend werden die Instanzen miteinander verbunden, damit der Blockkontext innerhalb der *Zustandsmaschine* und umgekehrt, verfügbar ist. Es folgt der Aufruf der definierten Testfälle, welche erfolgreich durchlaufen wurden.

```

1 odemx::base::Simulation* sim;
2 A* aBlock;
3 S* sStateMachine;
4 int main() {
5     sim = &odemx::getDefaultSimulation();
6     sStateMachine = new S(NULL, "aBlock", *sim);
7     aBlock = new A(sStateMachine);
8     return UnitTest::RunAllTests();
9 }

```

Listing 5.14: Testprogramm des Beispiels der Verhaltensabbildung

**Test: Initialknoten** Zunächst wird überprüft, ob der aktuelle *Zustand* der *Initialknoten* ist. Anschließend wird geprüft, ob das Zustandsmaschinenobjekt mit der Simulation verbunden ist. Der Blockkontext wird abgefragt und mit dem

globalen Objekt des *Blockes* verglichen. Abschließend wird geprüft, ob der Vorgabewert gesetzt wurde.

```

1 TEST(PseudostateInitial)
2 {
3   CHECK(&S::initial == sStateMachine->odemx::simml::StateMachine::getState());
4   CHECK(&sStateMachine->odemx::simml::StateMachine::getSimulation() == sim);
5   A
6   * context =
7     dynamic_cast<A*> (sStateMachine->odemx::simml::StateMachine::getContext());
8   CHECK(context == aBlock);
9   CHECK(context->a == 1);
10 }
```

Listing 5.15: Test: Initialknoten

**Test: Abschlussereignis** Wenn die Simulation für einen Schritt ausgeführt wird, muss das auslösende Ereignis ein *Abschlussereignis* sein. Dies ist erkennbar daran, dass der Prozess nicht unterbrochen und demnach auch nicht neu eingeplant werden musste. Der aktuelle Zustand muss durch zwei aufeinanderfolgende *Abschlussereignisse* der Zustand *s2* sein.

```

1 TEST(CompletionEvent)
2 {
3   sim->step();
4   CHECK(sStateMachine->odemx::simml::StateMachine::getAlerter() == NULL);
5   CHECK(&S::s2 == sStateMachine->odemx::simml::StateMachine::getState());
6 }
```

Listing 5.16: Test: Abschlussereignis

**Test: relatives Zeitereignis** Bisher ist noch keine Zeit vergangen (Zeitpunkt 0) und der nächste Schritt der Simulation führt bis zum Zeitpunkt 2. An dieser Stelle ist das relative Zeitereignis aufgetreten.

```

1 TEST(TimeEventRelative)
2 {
3   CHECK(sim->getTime() == 0);
4   sim->step();
5   CHECK(sim->getTime() == 2);
6   CHECK(sStateMachine->odemx::simml::StateMachine::getAlerter()->getMemoryType() ==
          odemx::sync::IMemory::TIMER);
7 }
```

Listing 5.17: Test: relatives Zeitereignis

**Test: Entscheidungsknoten** Wird die Simulation einen weiteren Schritt weit ausgeführt, so wurde der *Entscheidungsknoten* durchlaufen und anschließend der Zustand *s1* betreten. Durch den *Effekt* der letzten Transition hat der *Blockwert* den Wert 0.

```

1 TEST(PseudostateChoice)
2 {
3   sim->step();
4   CHECK(sStateMachine->odemx::simml::StateMachine::getAlerter() == NULL);
5   CHECK(&S::s1 == sStateMachine->odemx::simml::StateMachine::getState());
6   CHECK(aBlock->a == 0);
7 }

```

Listing 5.18: Test: Entscheidungsknoten

**Test: Zustandsereignis** Ein Fortfahren der Simulation führt zu keinem Zustandswechsel mehr, da der *Zustand* *s1* nur verlassen werden kann, wenn die *Bedingung* des *Wächters* an der *Transition* mit dem *Abschlussereignis* erfüllt ist oder die *Bedingung* des *Zustandsereignisses* der anderen *Transition* erfüllt ist. Das Setzen des *Blockwertes* führt dazu, dass die *Bedingung* des *Zustandsereignisses* erfüllt ist. Dies kann über den Wert des Prozessaktivierers abgefragt werden. Der aktuelle *Zustand* muss daraufhin *s2* sein.

```

1 TEST(ChangeEvent)
2 {
3   sim->step();
4   CHECK(&S::s1 == sStateMachine->odemx::simml::StateMachine::getState());
5   aBlock->a = -1;
6   CHECK(sStateMachine->odemx::simml::StateMachine::getAlerter()->getMemoryType() ==
          odemx::sync::IMemory::USER_DEFINED);
7   sim->step();
8   CHECK(&S::s2 == sStateMachine->odemx::simml::StateMachine::getState());
9 }

```

Listing 5.19: Test: Zustandsereignis

**Test: absolutes Zeitereignis** Der aktuelle Zeitpunkt muss unverändert 2 sein. Das Fortsetzen der Simulation löst das absolute *Zeitereignis* zum Zeitpunkt 3 aus. Dies kann erneut über den Prozessaktivierer geprüft werden.

```

1 TEST(TimeEventAbsolute)
2 {
3   CHECK(sim->getTime() == 2);
4   sim->step();
5   CHECK(sim->getTime() == 3);
6   CHECK(sStateMachine->odemx::simml::StateMachine::getAlerter()->getMemoryType() ==
          odemx::sync::IMemory::TIMER);
7 }

```

Listing 5.20: Test: absolutes Zeitereignis

**Test: Terminierungsknoten** Das Fortsetzen der Simulation führt zum *Terminierungsknoten*. Die Simulation muss damit beendet sein, da der Prozess nicht mehr in den Ereigniskalender eingetragen werden kann.

```
1 TEST(PseudoStateTerminate)
2 {
3     sim->step();
4     CHECK(odemx::simml::StateMachine::terminate == sStateMachine->odemx::simml::
        StateMachine::getState());
5     CHECK(sim->isFinished());
6 }
```

Listing 5.21: Test: Terminierungsknoten

## 5.5 Simulationsbeschreibung

Die Abbildung der Konfiguration beinhaltet die Generierung einer Implementierungsdatei, die als Simulationsprogramm dient.

### 5.5.1 Simulationsbeschreibungsabbildung

Für die Modellelemente im Paket, welche den Stereotypen `<<Simulation>>` (siehe Abschnitt 3.5) verwenden, müssen die initialen Instanzen der *Blöcke*, *Zustandsmaschinen* und ein Objekt der Klasse `odemx::base::Simulation` generiert werden. Die Simulation kann mit Hilfe der entsprechenden *Tags* auf eine Start und/oder Endzeit eingestellt werden.

#### Simulationsprogramm

Im folgenden Listing 5.22 werden die Transformationsregeln zur Generierung des Simulationsprogrammes beschrieben.

```
1 [template public SimulationCPP(aPackage : Package) post (format())]
2 [for ( classifier : Classifier | aPackage. getClassifier ())]
3 #include "[ classifier .name/].h";
4 [/for]
5
6 int main() {
7     odemx::base::Simulation& sim = odemx::getDefaultSimulation();
8     [generateSimulationConstraints ()/]
9     [for ( instanceSpecification : InstanceSpecification | aPackage.getInstance())]
10     [ instanceSpecification . generateInstances ()/]
11 [/for]
12 [for ( instanceSpecification : InstanceSpecification | aPackage.getInstance())]
13 [ instanceSpecification . generateConnections ()/]
14 [/for]
15 }
16 [/template]
```

Listing 5.22: Simulationsprogramm

**Simulationsobjekt** Das Simulationsobjekt wird über die Funktionalität der ODEMX-Bibliothek erzeugt und an den Konstruktor etwaiger Prozesse übergeben.

**Instanz-Spezifikationen** Zunächst wird für jede im Paket vorhandene *Instanz-Spezifikation* ein Objekt erzeugt. Dies beinhaltet ebenso die Belegung von *Block-werten*. Anschließend werden über die Namen der *Instanz-Spezifikationen*, die als Variablennamen im Programm vorkommen, die Instanzen untereinander verbunden, falls Verbindungen existieren.

### Beispiel für die Abbildung eines Simulationsprogrammes

Die Abbildung 5.3 zeigt die *Simulationsbeschreibung*, die die definierten Typen der vorangegangenen Abschnitte verwendet.

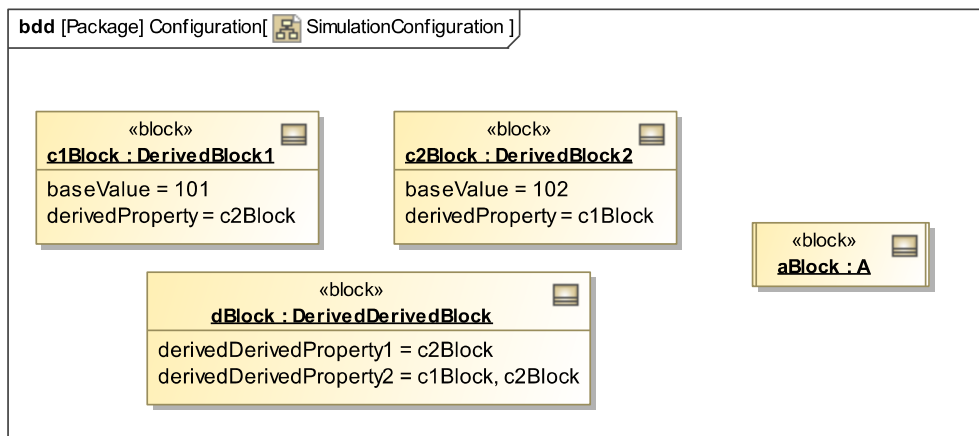


Abbildung 5.3: Simulationsbeschreibung

### Modellbeschreibung

Das Modell beinhaltet vier *Instanz-Spezifikationen*, die untereinander verbunden sind. *c1Block* und *c1Block* besitzen außerdem *Wertspezifikationen*. Die *Instanz-Spezifikation* *aBlock* ist vom Typ *A*, welcher ein *aktiver Block* ist. Somit existiert auch eine *Zustandsmaschine*.

### ODEMX-Modell

Das Listing 5.23 zeigt einen Ausschnitt mit der Funktion `main`, des mit Hilfe der Abbildungsregeln generierten ODEMX-Simulationsprogrammes.

```

1 int main() {
2     odemx::base::Simulation& sim = odemx::getDefaultSimulation();

```

```
3 Simulation dBlock = new DerivedDerivedBlock();
4 aBlock = new A(new S(NULL, "S", sim));
5 c1Block = new DerivedBlock1();
6 c1Block->baseValue = 101;
7 c2Block = new DerivedBlock2();
8 c2Block->baseValue = 102;
9 dBlock->derivedDerivedProperty2.insert(c1Block);
10 dBlock->derivedDerivedProperty2.insert(c2Block);
11 dBlock->derivedDerivedProperty1 = c2Block;
12 c1Block->derivedProperty = c2Block;
13 c2Block->derivedProperty = c1Block;
14 }
```

Listing 5.23: Ausschnitte des generierten Simulationsprogrammes



## 6 Zusammenfassung

Die vorliegende Arbeit beschreitet den Weg von einem Modell in einer Modellierungssprache zu einem ausführbaren Simulationsmodell. Ausgehend von SysML wurden Erweiterungen und Semantik definiert, die in Hinblick auf eine Abbildung getroffen wurden, so dass die Sprache SimML entstand. Mit Hilfe dieser Sprache lassen sich ausführbare Simulationsmodelle beschreiben. Um diese Modelle auf einem Rechner ausführen zu können, wurde die C++-Simulationsbibliothek ODEMX um notwendige Konzepte erweitert, so dass diese die Abbildung der SimML-Modelle ermöglicht.

Modelle in SysML sind nicht detailliert genug, um ausführbare Simulationsmodelle zu beschreiben. Sie ermöglichen aber die Veranschaulichung und Beschreibung von Systemen unabhängig von einer konkreten Ausführungsplattform und verwenden dabei Konzepte der typbasierten Beschreibung von Systemelementen. Zusätzlich werden Verhaltensbeschreibungen mit abstrakten Kalkülen z. B. von Zustandsmaschinen oder Aktivitäten vorgenommen (Beispiele können der Studienarbeit „Systemmodellierung mit SysML“ [17] des Autors entnommen werden). Die Verhaltensbeschreibungen an sich verwenden Basiskonzepte wie das Warten auf Zeit- oder Zustandereignisse. ODEMX verfügt über diese Basiskonzepte, die im Rahmen dieser Arbeit ergänzt wurden, so dass eine Abbildung zwischen den Konzepten von SimML und ODEMX möglich ist.

Die Neuerungen der ODEMX-Bibliothek gewährleisten, dass die Abbildung von Struktur- und Verhaltensbeschreibung strukturäquivalent erfolgt. Um dieses zu gewährleisten, wurden neben den geforderten Basiskonzepten erweiterte Konzepte zur Verfügung gestellt, die die Basiskonzepte verwenden. Die zu ODEMX hinzugefügten Klassen tragen in diesem Fall den gleichen Namen wie die Konzepte der Sprache SimML. Außerdem gibt es in der Abbildung eine Umsetzung der SimML-Semantik, die die Repräsentation von Instanzen (Objekte oder Werte) und den Lebensbereich von Instanzen festlegt, innerhalb dessen ein erfolgreicher Zugriff auf die Instanz ermöglicht werden muss. Damit Objekte den definierten Lebensbereich aus SimML besitzen und es eine Wertesemantik für die aus SimML vordefinierten Typen gibt, mussten die entsprechenden Klassen in ODEMX erstellt werden, die diese Semantik umsetzen. Für die Verhaltensbeschreibung mussten Erweiterungen für die Abbildung des State-Patterns nach ODEMX beschrieben werden. Diese umfassen auch die Basiskonzepte zum Warten auf Zustandereignisse und dem Beobachten von zeitdiskreten Zustandsvariablen. In SimML ist es außerdem möglich kombinierte Systeme zu beschreiben, die strukturäquivalent mit Hilfe der hinzugefügten Klassen zu ODEMX abgebildet werden können. Solch eine Abbildung war keine Anforderung an diese Arbeit,

wird aber im folgenden Abschnitt skizziert.

## 6.1 Ausblick

Nachfolgend sind Ansätze für zukünftige Forschungen aufgeführt, die sich im Rahmen der Arbeit ergeben haben.

### 6.1.1 Abbildung kombinierter Systeme

Die Abbildung von kombinierten Systemmodellen in SimML mit zeitkontinuierlichen Zustandsänderungen nach ODEmx ist eine offene Forschungsfrage. Zeitkontinuierliche Zustandsänderungen werden in SysML mit Differentialgleichungen beschrieben, die in *Bedingungen* von *Bedingungsblöcken* formuliert werden. Es dürfen ausschließlich Differentialgleichungen 1. Grades verwendet werden, da sonst eine direkte Abbildung nach ODEmx nicht möglich wäre (Näheres siehe „Objektorientierte Prozeßsimulation in C++“ von Joachim Fischer und Klaus Ahrens [8]). Die beteiligten Zustandsvariablen, die sich zeitkontinuierlich ändern, können gleichzeitig zeitdiskret durch Aktionen in Zustandsmaschinen verändert oder gelesen werden. In dieser Situation müssen die Werte der Zustandsvariablen für exakt diesen Zeitpunkt berechnet werden. Hierfür kann die Klasse `odemx::base::continous::Monitor`, die Berechnung der Werte mit einem Prozess bzw. einer Instanz einer Zustandsmaschine synchronisieren (`Monitor::addProcessToWaitList`). Außerdem können Zustandsergebnisse über diese Zustandsvariablen definiert werden, so dass beim Eintritt des *Ereignisses* das zeitkontinuierliche Ändern unterbrochen wird. Die Klasse `odemx::sync::WaitCondition` kann mit diesem Monitor verbunden werden (`Monitor::addStateEvent`), so dass die darin beschriebene Zustandsbedingung während der Berechnung überwacht wird. Tritt das Zustandsergebnis ein, so unterbricht die Berechnung und die Werte der Zustandsvariablen können für exakt diesen Zeitpunkt verwendet werden. Prinzipiell wird bei der Verwendung von `odemx::base::continous::Monitor` empfohlen, Differentialgleichungen, die gekoppelt sind, mit genau einem Monitor zu verwenden. Außerdem kann eine Zustandsbedingung nur mit genau einem Monitor verbunden werden, d. h. genau ein Monitor kann eine Zustandsbedingung überwachen. Enthält eine Zustandsbedingung Zustandsvariablen, die in unterschiedlichen Differentialgleichungen unterschiedlicher Monitore verändert werden, so könnte beim Eintreten des Ereignisses eine der Zustandsvariablen einen Wert besitzen, der nicht exakt für den Zeitpunkt des Ereignisses berechnet wurde (da ein anderer Monitor die Wertänderung vorgenommen hat, der nicht mit diesem Ereignis synchronisiert wurde). Für diese Fragestellung gibt es bisher kein Konzept zur Lösung in der ODEmx-Bibliothek. Eine Möglichkeit wäre, zu fordern, dass in diesem Fall die Differentialgleichungen zum selben Monitor gehören, d. h. im Kontext eines Blockes definiert sind. Eine andere Möglichkeit wäre, dass die Zustandsbedingungen in beiden Monitoren registriert werden, so dass beide beim Erfüllen der Bedingung die Werte für den

Zeitpunkt des Ereignisses berechnen. Eine weitergehende Arbeit müsste sich konkret mit den Vor- und Nachteilen beider Lösungen auseinandersetzen. Zusätzlich müsste die Simulationszeit beider Ansätze betrachtet werden, da diese aufgrund der Anzahl der Berechnungsschritte entscheidend beeinflusst wird.

### 6.1.2 Zufallszahlen

Die Verwendung von Zufallszahlen spielt bei der Definition von Systemen eine wichtige Rolle. SysML bietet bisher keine Beschreibungsmöglichkeit für Verteilungsfunktionen. Die Sprache schlägt jedoch Stereotypen für einzelne Wahrscheinlichkeitsverteilungen vor, deren Realisierung von einem ausführbaren System vorgenommen werden muss. Konkret bedeutet dies, dass mit Hilfe von festgelegten Stereotypen für Wahrscheinlichkeitsverteilungen diese für die Erzeugung von Zufallszahlen auf Zufallszahlgeneratoren abgebildet werden müssten. ODEmx bietet bereits eine Vielzahl von Zufallszahlgeneratoren und es bleibt zu untersuchen, ob diese für eine Abbildung genutzt werden können.

### 6.1.3 Aktionssprache

Eine weitere offene Frage bleibt die Verwendung einer Aktionssprache, die gepart zum Zeitpunkt der Abbildung vorliegt. Hierfür gibt es bereits Werkzeugunterstützung bei der Definition und der Einbindung solcher Sprachen in die Werkzeuge für die Abbildung von Modellen (siehe „Xtext Documentation“ der itemis AG [15]). Prinzipiell könnte somit automatisch an einer Anweisung oder Bedingung erkannt werden, welche Variablen beteiligt sind und die Angabe über ein *Tag* wie aktuell in SimML vorgeschrieben würde entfallen. Zusätzlich müssten die Operatoren der Typen von Blockeigenschaften in ODEmx nicht überladen sein, um bei Änderung entsprechend reagieren zu können. Das damit verbundene Prüfen von Zustandsbedingungen müsste dann durch die Abbildung an entsprechender Stelle generiert werden. Die aktuelle Lösung wurde gewählt, da ohne eine Realisierung dieser Aktionssprache, keine Vorteile gegenüber der Aktionssprache in C++ in Hinblick auf das Gesamtziel erkennbar sind. Durch die Entwicklung und Anbindung in die Abbildung könnte dieser Fragestellung nachgegangen werden.

### 6.1.4 Konvertierung von Werten

In der Sprache SysML ergeben sich mit der Einführung von *Wertetypen* weitere Probleme, da die Semantik und Syntax für diese Typen nicht vollständig definiert ist. Die Sprachdefinition sieht nämlich für solche Typen die Definition von *Einheiten* und *physikalischer Größe* vor, macht aber gleichzeitig keine Aussage darüber, wie diese formal definiert werden können. Des Weiteren ist es bisher nicht möglich Beziehungen zwischen *Wertetypen* gleicher *physikalischer Größe*, aber mit unterschiedlicher Einheit, miteinander in Beziehung zu setzen. Dies könnte bei der Verwendung dieser Typen eine automatische Umrechnung

der Werte bewirken. Wie dies in SimML definiert und durch die Abbildung in ODEmx realisiert werden kann, bleibt weiter zu untersuchen.

## 6.2 Resultat

Es wurde gezeigt, wie ein SimML-Modell mit zeitdiskreter Verhaltensbeschreibung in ein ausführbares Simulationsprogramm, welches ODEmx verwendet, transformiert werden kann.

## A MTL-Transformationsregeln

```

1 [comment encoding = UTF-8 /]
2 /**
3  * The documentation of the module generate.
4  */
5 [module generate('http://www.eclipse.org/uml2/3.0.0/UML')/]
6 [import hub::sam::lang::simml::simml2odemx::services::ValidateService /]
7 [import hub::sam::lang::simml::simml2odemx::files::BlockH/]
8 [import hub::sam::lang::simml::simml2odemx::files::BlockCPP/]
9 [import hub::sam::lang::simml::simml2odemx::files::StateMachineH/]
10 [import hub::sam::lang::simml::simml2odemx::files::StateMachineCPP/]
11 [import hub::sam::lang::simml::simml2odemx::files::SimulationCPP/]
12
13 /**
14  * The documentation of the template generate.
15  * @param aModel
16  */
17 [template public generate(aModel : Model)]
18 [comment @main/]
19 [if (not getAppliedProfile('SimML Profile').oclIsUndefined())/]
20 [let validModel : Boolean = validate(aModel)]
21   [if validModel]
22     [for (nestedPackage : Package | aModel.nestedPackage)]
23       [if not (nestedPackage.getAppliedStereotype('Kernel::Simulation').oclIsUndefined())]
24         [generateSimulation()/]
25       [else]
26         [for (packageableElement : PackageableElement | nestedPackage.packageableElement)]
27           [if packageableElement.oclIsTypeOf(Class)]
28             [packageableElement.oclAsType(Class).generateBlock()/]
29           [/if]
30         [/for]
31       [/if]
32     [/for]
33   [/if]
34 [/let]
35 [if /]
36 [/template]
37
38 [template public generateBlock(block : Class) ? (not getAppliedStereotype('Blocks::Block').oclIsUndefined())]
39 [file (block.name.concat('.h'), false, 'UTF-8')]
40 [block.BlockH()/]
41 [/file]
42
43 [file (block.name.concat('.cpp'), false, 'UTF-8')]
44 [block.BlockCPP()/]
45 [/file]
46
47 [if block.isActive and block.classifierBehavior.oclIsTypeOf(StateMachine)]
48 [let stateMachine : StateMachine = block.classifierBehavior.oclAsType(StateMachine)]
49 [file (stateMachine.name.concat('.h'), false, 'UTF-8')]
50 [stateMachine.StateMachineH()/]
51 [/file]
52 [file (stateMachine.name.concat('.cpp'), false, 'UTF-8')]

```

```

53 [stateMachine.StateMachineCPP()/]
54 [ / file ]
55 [ /let ]
56 [ /if ]
57 [ /template ]
58
59 [template public generateSimulation(simulation : Package)]
60 [ file (simulation.name.concat('.cpp'), false, 'UTF-8')]
61 [simulation.SimulationCPP()/]
62 [ / file ]
63 [ /template ]

```

Listing A.1: generate.mtl

```

1 [comment encoding = UTF-8 /]
2 /**
3  * The documentation of the module ValidateService.
4  */
5 [module ValidateService('http://www.eclipse.org/uml2/3.0.0/UML')]
6
7 /**
8  * The documentation of the query validate .
9  * @param aModel
10 */
11 [query public validate (aModel : Model) : Boolean =
12 validatePackages() and
13 nestedPackage.packagedElement->filter(Type)->forAll(typeVisibility()) and
14 nestedPackage.packagedElement->filter(Class)->forAll(activeClass()) and
15 nestedPackage.packagedElement->filter(Class)->forAll(classProperty()) and
16 nestedPackage.packagedElement->filter(Class).ownedOperation->forAll(operationParameter())
17 and
18 nestedPackage.packagedElement->filter(StateMachine)->forAll(stateMachineRegion()) and
19 nestedPackage.packagedElement->filter(StateMachine).region->forAll(regionInitial()) and
20 nestedPackage.packagedElement->filter(StateMachine).region->forAll(regionTerminate()) and
21 nestedPackage.packagedElement->filter(StateMachine).region.subvertex->filter(State)->forAll
22 (simpleState()) and
23 nestedPackage.packagedElement->filter(StateMachine).region.subvertex.outgoing.trigger.event
24 ->forAll(eventType())
25 /]
26
27 [query public validatePackages(aModel : Model) : Boolean =
28 nestedPackage->reject(name.matches('UML Standard Profile'))->select(getAppliedStereotype('
29 Kernel::Simulation')).oclIsUndefined()->size() = 1
30 /]
31
32 [query public typeVisibility (type : Type) : Boolean =
33 visibility = VisibilityKind :: public
34 /]
35
36 [query public activeClass (class : Class) : Boolean =
37 if isActive then not ( classifierBehavior .oclAsType(StateMachine).oclIsUndefined()) else
38 classifierBehavior .oclIsUndefined() endif
39 /]

```

```
36 [query public classProperty ( class : Class ) : Boolean =
37   attribute ->forAll( visibility = VisibilityKind :: public and (type.ocllsKindOf( Class ) or type.
      occllKindOf( DataType ))
38 /]
39
40 [query public operationParameter ( operation : Operation ) : Boolean =
41   ownedParameter ->forAll ( direction = ParameterDirectionKind :: inout or direction =
      ParameterDirectionKind :: return )
42 /]
43
44 [query public stateMachineRegion ( stateMachine : StateMachine ) : Boolean =
45   stateMachine.region ->size() = 1
46 /]
47
48 [query public regionInitial ( region : Region ) : Boolean =
49   subvertex ->select ( occllKindOf ( Pseudostate ) ).oclAsType ( Pseudostate ) ->select ( kind =
      PseudostateKind :: initial ) ->size() = 1
50 /]
51
52 [query public regionTerminate ( region : Region ) : Boolean =
53   subvertex ->select ( occllKindOf ( Pseudostate ) ).oclAsType ( Pseudostate ) ->select ( kind =
      PseudostateKind :: terminate ).outgoing ->size() = 1
54 /]
55
56 [query public simpleState ( state : State ) : Boolean =
57   isSimple
58 /]
59
60 [query public eventType ( event : Event ) : Boolean =
61   occllTypeOf ( TimeEvent ) or not ( getAppliedStereotype ( ' Communications :: AnnotatedChangeEvent
      ' ).ocllsUndefined () )
62 /]
```

Listing A.2: ValidateService.mtl

```
1 [comment encoding = UTF-8 /]
2 /**
3  * The documentation of the module UtilityService .
4  */
5 [module UtilityService ( ' http://www.eclipse.org/uml2/3.0.0/UML' )]
6
7 [query public format ( arg0 : String ) : String
8   = invoke ( ' hub.sam.lang.simml.simml2odemx.services.CodeFormatterService ' , ' format ( java .
      lang . String ) ' , Sequence { arg0 } ) /]
9
10 [query public getTaggedValues ( e : Element , stereoTypeName : String , propName : String ) :
      Set ( Element ) =
11   e.getValue ( e.getAppliedStereotype ( stereoTypeName ) , propName )
12 /]
13
14 [template public getNamespaceName ( namedElement : NamedElement , namespace : Namespace )
      ]
15 [if not namedElement.namespace.name.ocllsUndefined () and not ( namedElement.namespace =
      namespace )][namedElement.namespace.name/]::[/if][namedElement.name/]
```



```

16 [/template]
17
18 [template public getODEMxSimMLPackage(seperator : String)]
19 odemx[seperator/]simml
20 [/template]
21
22 [template public getODEMxSimMLClass(seperator : String, class : String)]
23 [getODEMxSimMLPackage(seperator)/][seperator/][class/]
24 [/template]
25
26 [template public generateInstanceValue( defaultValue : ValueSpecification , definingFeature :
    String , definingFeatureType : Type, definingNamespace : Namespace)]
27 [ if defaultValue.ocllsKindOf( InstanceValue )]
28 [let instance : InstanceSpecification = defaultValue.oclAsType(InstanceValue).instance ]
29 [definingFeature/] = new [instance. classifier .name/]() ;
30 [for ( slot : Slot | instance . slot )]
31 [let slotDefiningFeature : String = slot. definingFeature .getNamespaceName(
    definingNamespace)]
32 [for (slotValue : ValueSpecification | slot .value)]
33 [if not instance. classifier ->includes(definingFeatureType)]
34 [generateInstanceValue( slotValue , 'dynamic_cast<' + instance. classifier .name + '*> (' +
    definingFeature + '.ptr)' + '->' + slotDefiningFeature, slot . definingFeature .type,
    definingNamespace)/]
35 [else]
36 [generateInstanceValue( slotValue , definingFeature + '->' + slotDefiningFeature, slot .
    definingFeature .type, definingNamespace)/]
37 [/if]
38 [/for]
39 [/let]
40 [/for]
41 [/let]
42 [elseif defaultValue.ocllsKindOf( LiteralInteger )]
43 [definingFeature/] = [defaultValue.oclAsType( LiteralInteger ).value/];
44 [elseif defaultValue.ocllsKindOf( LiteralUnlimitedNatural )]
45 [definingFeature/] = [defaultValue.oclAsType( LiteralUnlimitedNatural ).value/];
46 [elseif defaultValue.ocllsKindOf( LiteralString )]
47 [definingFeature/] = [defaultValue.oclAsType( LiteralString ).value/];
48 [elseif defaultValue.ocllsKindOf( LiteralBoolean )]
49 [definingFeature/] = [defaultValue.oclAsType( LiteralBoolean ).value/];
50 [/if]
51 [/template]
52
53 [template public getConstraint( constraints : Sequence(String))]
54 [for ( constraint : String | constraints )]
55 [if i = constraints->size()]
56 [if constraint .equalsIgnoreCase( 'ELSE')]true[else][ constraint /][if]
57 [else]
58 [constraint /]
59 [/if]
60 [/for]
61 [/template]

```

Listing A.3: UtilityService.mtl

```

1 [comment encoding = UTF-8 /]
2 /**
3  * The documentation of the module BlockService.
4  */
5 [module BlockService('http://www.eclipse.org/uml2/3.0.0/UML')]
6 [import hub::sam::lang::simml::simml2odemx::services:: UtilityService /]
7 [import hub::sam::lang::simml::simml2odemx::services:: StateMachineService/]
8
9 [query public getGeneralClassifiers (block : Class) : Set( Classifier ) =
10 block.general->asSet()
11 /]
12
13 [query public rejectBuildInTypes (types : Set(Type)) : Set(Type) =
14 types->reject(qualifiedName.startsWith('Blocks:: '))
15 /]
16
17 [query public getIncludes (block : Class) : Set(Type) =
18 rejectBuildInTypes (block. getGeneralClassifiers ()->filter(Type)->union(block.attribute.type
19   ->asSet())->union(block.ownedOperation.ownedParameter.type->asSet()))
20 /]
21
22 [query public getForwardDeclaration (block : Class) : Set(Type) =
23 block. attribute .type->asSet()->select(t | t->select(oclIsKindOf( Classifier )) .oclAsType(
24   Classifier ). attribute .type->asSet())->includes(block))
25 /]
26
27 [query public getPropertyWithDefaultValue (block : Class) : Set(Property) =
28 block. attribute ->select(a | not a.default .oclIsUndefined () or not a.defaultValue .
29   oclIsUndefined ())
30 /]
31
32 [template public getDefaultValueFunction (block : Class)]
33 _setDefaultValues
34 [/template]
35
36 [template public getPropertyParameter (property : Property)]
37 [if property . isMultivalued ()][property .type.getType()/][else][property .type.name/][if] , [
38   property .isComposite. toString ()/]
39 [/template]
40
41 [template public getType (type : Type)]
42 [type.name/][if type.qualifiedName.contains('Blocks:: Integer') or type.qualifiedName.
43   contains('Blocks:: Real') or type.qualifiedName.contains('Blocks:: String') or type.
44   qualifiedName.contains('Blocks:: Boolean')][else]*[/if]
45 [/template]
46
47 [template public getPropertyElementType (property : Property)]
48 [if not property . isMultivalued ()]Property[else][property . getMultiplicityElementType ()/][if]
49 ]
50 [/template]
51
52 [template public getMultiplicityElementType ( multiplicityElement : MultiplicityElement )]
```

```

46 [if multiplicityElement .isOrdered][if multiplicityElement .isUnique]OrderedSet[else]Sequence
   [/if][elseif not multiplicityElement .isUnique]Bag[else]Set[/if]
47 [/template]
48
49 [template public getMultivaluedTyp(property : Property)]
50 [if property .isMultivalued ()] :: Type[/if]
51 [/template]
52
53 [template public getOperationType(operation: Operation)]
54 [if (operation .type .oclIsUndefined ())]void[else][operation .type .name/][[/if]
55 [/template]
56
57 [template public getParameterType(parameter : Parameter)]
58 [if parameter .isMultivalued ()][getODEMxSimMLPackage('::')/][parameter.
   getMultiplicityElementType()/]<[parameter.type.getType()/]>::Type[else][parameter.type.
   getType()/][[/if]
59 [/template]
60
61 [template public getOperationParameter(operation : Operation)]
62 [for (parameter : Parameter | operation.ownedParameter) separator(',') ? (parameter.
   direction <> ParameterDirectionKind::return)][parameter.getParameterType()/] [
   parameter.name/][[/for]
63 [/template]
64
65 [template public getOperationReturn(operation : Operation)]
66 [operation.ownedParameter->any(parameter | parameter.direction = ParameterDirectionKind::
   return).getParameterType()/]
67 [/template]
68
69 [template public getOperationDeclaration(operation : Operation)]
70 [operation .getOperationReturn()/] [operation .name/][([operation .getOperationParameter()/])
71 [/template]
72
73 [template public generateSwitchCase(block : Class, constraint : Boolean)]
74 [if block .classifierBehavior .oclIsTypeOf(StateMachine)]
75 switch ((*cases)['[/]0]) {
76 [for (iVertex : Vertex | block .classifierBehavior .oclAsType(StateMachine).getFirstRegion().
   getStates())]
77 [let i : Integer = i]
78 case [i/]:
79   switch ((*cases)['[/]1]) {
80     [for (jTransition : Transition | iVertex.outgoing)]
81     [let j : Integer = i]
82     case [j/]:
83       [if constraint]
84       switch ((*cases)['[/]2]) {
85         case 0:
86           [if not (jTransition .guard .oclIsUndefined ())]
87           [if jTransition .guard .specification .oclIsTypeOf(OpaqueExpression)]
88           return ([let constraints : Sequence(String) = jTransition .guard .specification
89                   .oclAsType(OpaqueExpression)._body][getConstraint(constraints)/][[/let]);
90           [/if]
91           [/if]
           break;

```

```
92     [for (kTrigger : Trigger | jTransition . trigger )]
93     [if kTrigger . event . oclIsTypeOf(ChangeEvent)]
94     [let k : Integer = i]
95     case [k/]:
96     return ([kTrigger . event . oclAsType(ChangeEvent).changeExpression . oclAsType(
97         OpaqueExpression)._body/]);
97     break;
98     [/let]
99     [/if]
100    [/for]
101    }
102    [elseif not (jTransition . effect . oclIsUndefined())]
103    [if jTransition . effect . oclIsTypeOf(OpaqueBehavior)]
104    [jTransition . effect . oclAsType(OpaqueBehavior)._body/]
105    [/if]
106    [/if]
107    break;
108    [/let]
109    [/for]
110    }
111    break;
112 [/let]
113 [/for]
114 }
115 [/if]
116 [/template]
```

Listing A.4: BlockService.mtl

```
1 [comment encoding = UTF-8 /]
2 /**
3  * The documentation of the module BlockH.
4  */
5 [module BlockH('http://www.eclipse.org/uml2/3.0.0/UML')/]
6 [import hub::sam::lang::simml::simml2odemx::services:: BlockService/]
7 [import hub::sam::lang::simml::simml2odemx::services:: UtilityService /]
8
9 /**
10 * The documentation of the template BlockH.
11 * @param block
12 */
13 [template public BlockH(block : Class) post (format())]
14 #ifndef [block.name.toUpper()/_H_]
15 #define [block.name.toUpper()/_H_]
16
17 #include <[getODEMxSimMLClass('/', 'IncludeBlock')/_].h>
18 [for (include : Type | getIncludes(block))]
19 #include "[include.name/_].h"
20 [/for]
21 [if block.isActive ]
22 #include "[block.classifierBehavior.name/_].h"
23 [/if]
24 [for (forward : Type | getForwardDeclaration(block))]
25 class [forward.name/];
```

```

26 [/for]
27
28 class [block.name/]: public virtual [getODEMxSimMLClass('::', 'Block')]/[for (general :
    Classifier | getGeneralClassifiers (block))], public virtual [general.name/]/[for] {
29 public:
30     [block.name/]()();
31     virtual ~[block.name/]()();
32
33     [if block.isActive]
34     [block.name/]/([block.classifierBehavior.name/] * classifierBehavior );
35     virtual void
36         _run(odemx::base::SimTime time, const std::vector<long>* cases);
37     virtual bool
38         _check(odemx::base::SimTime time, const std::vector<long>* cases) const;
39
40     [/if]
41     [for (property : Property | block.attribute)]
42         [if not property.ocllsTypeOf(Port)]
43             [getODEMxSimMLPackage('::')/]:[property.getPropertyElementType()/]<[property.
                getPropertyParameter()/]>[property.getMultivaluedTyp()/] [property.name/];
44     [/if]
45 [/for]
46
47 [for (operation : Operation | block.getOperations())]
48     [if operation.isStatic] static [/if] [operation.getOperationDeclaration()/] [if operation.
        isQuery]const[/if];
49 [/for]
50
51 [if (getPropertyWithDefaultValue(block)->size() > 0)]
52 private :
53     void _setDefaultValues();
54
55 [/if]
56 };
57
58 #endif /* [block.name.toUpper()/]_H_ */
59
60 [/template]

```

Listing A.5: BlockH.mtl

```

1 [comment encoding = UTF-8/]
2 [module BlockCPP('http://www.eclipse.org/uml2/3.0.0/UML')/]
3 [comment][import hub::sam::lang::simml::simml2cpp::prototype:: files::StateMachineH]/[/
    comment]
4 [import hub::sam::lang::simml::simml2odemx::services:: BlockService/]
5 [import hub::sam::lang::simml::simml2odemx::services:: UtilityService /]
6
7 [template public BlockCPP(block : Class) post (format())]
8 #include "[block.name/].h"
9 [if block.isActive]
10     [if not block.classifierBehavior.getAppliedStereotype('StateMachine::
        StateMachineWithDependencies').ocllsUndefined())]

```

```

11  [for ( includeClassifier : Element | getTaggedValues(block. classifierBehavior , '
      StateMachine::StateMachineWithDependencies', 'dependencies'))]
12    #include "[ includeClassifier ->filter( Classifier ).name/.h"
13  [/for]
14  [/if]
15 [/if]
16
17 [let propertyWithDefaultValue : Set(Property) = getPropertyWithDefaultValue(block)]
18 [block.name/]:[block.name/]() : [getODEMxSimMLClass('::', 'Block')/]() {
19  [if (getPropertyWithDefaultValue(block)->size() > 0)] [block.getDefaultValueFunction()/]();
    [/if]
20 }
21
22 [block.name/]:~[block.name/]() {
23 }
24  [if block. isActive ]
25
26 [block.name/]:[block.name/]( [block. classifierBehavior .name/] * classifierBehavior ) : [
    getODEMxSimMLClass('::', 'Block')/]( classifierBehavior ) {
27  [if (getPropertyWithDefaultValue(block)->size() > 0)] [block.getDefaultValueFunction()/]();
    ; [/if]
28 }
29
30 void [block.name/]:_run(odemx::base::SimTime time,
31   const std :: vector<long>* cases){
32   [generateSwitchCase(block, false )/]
33 }
34
35 bool [block.name/]:_check(odemx::base::SimTime time,
36   const std :: vector<long>* cases) const {
37   [generateSwitchCase(block, true)/]
38   return [getODEMxSimMLClass('::', 'Block')/]:_check(time, cases);
39 }
40
41 [/if]
42
43 [for (operation : Operation | block.getOperations())]
44   [operation . getOperationDeclaration ()/] [if operation . isQuery]const[/if]{
45     [if operation . method->forall(oclIsKindOf(OpaqueBehavior))]
46       [operation . method.oclAsType(OpaqueBehavior)._body/]
47     [/if]
48 }
49
50 [/for]
51 [if (propertyWithDefaultValue->size() > 0)]
52   void [block.name/]:[block.getDefaultValueFunction()/]() {
53     [for (property : Property | propertyWithDefaultValue)]
54       [if not property . default . oclIsUndefined ()]
55         [property . getNamespaceName(property.namespace)/] = [property.default/];
56       [else]
57         [for (defaultValue : ValueSpecification | property . defaultValue)]
58           [generateInstanceValue( defaultValue , property . getNamespaceName(property.
              namespace), property.type, property.namespace)/]
59     [/for]

```

```

60     [/if]
61   [/for]
62 }
63 [/if]
64 [/let]
65
66 [/template]

```

Listing A.6: BlockCPP.mtl

```

1  [comment encoding = UTF-8 /]
2  /**
3   * The documentation of the module StateMachineService.
4   */
5  [module StateMachineService('http://www.eclipse.org/uml2/3.0.0/UML')]
6  [import hub::sam::lang::simml::simml2odemx::services:: UtilityService /]
7
8  [query public getFirstRegion(stateMachine : StateMachine) : Region =
9  stateMachine.region->asSequence()->first()
10 /]
11
12 [query public getStates(region : Region) : Set(Vertex) =
13 region.subvertex->select(ocllsTypeOf(State))->union(region.subvertex->filter(Pseudostate)
14   ->select(kind = PseudostateKind::choice or kind = PseudostateKind:: initial ))
15 /]
16
17 [query public getFunctionStates(region : Region) : Set(Vertex) =
18 region.subvertex->select(ocllsTypeOf(State))->union(region.subvertex->filter(Pseudostate)
19   ->select(kind = PseudostateKind::choice))
20 /]
21
22 [query public getPseudostatesByKind(region : Region, pseudostateKind : PseudostateKind) :
23   Set(Pseudostate) =
24 region.subvertex->filter(Pseudostate)->select(kind = pseudostateKind)
25 /]
26
27 [query public getPseudostateInitial (region : Region) : Pseudostate =
28 region.getPseudostatesByKind(PseudostateKind:: initial )->asSequence()->first()
29 /]
30
31 [template public getVertexName(vertex : Vertex, stateMachineName : String)]
32 [if vertex.ocllsTypeOf(Pseudostate) and vertex.oclAsType(Pseudostate).kind =
33   PseudostateKind::terminate]
34 [getODEMxSimMLClass('::', 'StateMachine')/]:[vertex.oclAsType(Pseudostate).kind/]
35 [else]
36 [stateMachineName/]:[vertex.name/]
37 [/if]
38 [/template]
39
40 [template public generatePossibleTransition1 (label : String)]
41 [getODEMxSimMLClass('::', 'StateMachine::addPossibleTransition')/](
42 new [getODEMxSimMLClass('::', 'Transition')/](odemx::base::Process:: getSimulation(), [label
43 /]
44 [/template]

```

```

40
41 [template public generatePossibleTransition2 (stateMachineName : String, targetVertex :
    Vertex)]
42 [getODEMxSimMLClass(':', 'StateMachine::Function')/<
43 [stateMachineName/]>::Pointer([targetVertex.getVertexName('&' + stateMachineName)/]]);
44 [/template]
45
46 [template public generateMonitors(changeEvent : ChangeEvent)]
47 monitors = new std::vector<odemx::sync::Monitor*>();
48 [for (property : Property | changeEvent.getTaggedValues('Communications::
    AnnotatedChangeEvent', 'variables')->filter(Property))]
49 monitors->push_back(&dynamic_cast<[property.class.name/]*> ([getODEMxSimMLClass(':', '
    Behavior')/]>::getContext())->[property.name/].odemx::sync::Monitored<[property.type.
    name/]>::getMonitor());[/for]
50 [/template]
51
52 [template public getVertexImplementation(vertex : Vertex, region : Region)]
53 [let stateMachineName : String = region.stateMachine.name]
54 // Trigger: [vertex.outgoing.trigger->size()/]
55 [if not vertex.outgoing.trigger.event->filter(ChangeEvent)->isEmpty()]
56 std::vector<odemx::sync::Monitor*>* monitors;
57 [/if]
58 [if vertex.ocllsTypeOf(Pseudostate)]
59 [if vertex.oclAsType(Pseudostate).kind = PseudostateKind::choice]
60 [getODEMxSimMLClass(':', 'StateMachine')/]>::setPseudoState(
61 [getODEMxSimMLClass(':', 'StateMachine::CHOICE')/]);
62 [/if]
63 [/if]
64 [for (transition : Transition | vertex.outgoing)]
65 [if transition.trigger->isEmpty()]
66 [generatePossibleTransition1 ("completion event"/],
67 [getTransitionParameter(region, vertex, transition, null)/],
68 [generatePossibleTransition2 (stateMachineName, transition.target)/]
69 [else]
70 [for (trigger : Trigger | transition.trigger)]
71 [if trigger.event.ocllsTypeOf(TimeEvent)]
72 [let timeEvent : TimeEvent = trigger.event.oclAsType(TimeEvent)]
73 [generatePossibleTransition1 ("time event"/],
74 [timeEvent.isRelative /], [timeEvent.when.expr.oclAsType(LiteralString).value/],
75 [getTransitionParameter(region, vertex, transition, trigger)/],
76 [generatePossibleTransition2 (stateMachineName, transition.target)/]
77 [/let]
78 [elseif trigger.event.ocllsTypeOf(ChangeEvent)]
79 [let changeEvent : ChangeEvent = trigger.event.oclAsType(ChangeEvent)]
80 [generateMonitors(changeEvent)/]
81 [generatePossibleTransition1 ("change event"/],
82 [getODEMxSimMLClass(':', 'Behavior')/]>::getContext(), monitors,
83 [getTransitionParameter(region, vertex, transition, trigger)/],
84 [generatePossibleTransition2 (stateMachineName, transition.target)/]
85 [/let]
86 [/if]
87 [/for]
88 [/if]
89 [/for]

```



```

90 [/let]
91 [/template]
92
93 [template public getTransitionParameter(region : Region, gVertex : Vertex, hTransition :
    Transition, iTrigger : Trigger)]
94 [for (vertex : Vertex | region.getStates())]
95 [if gVertex = vertex]
96 [let g : Integer = i]
97 [for (transition : Transition | gVertex.outgoing)]
98 [if hTransition = transition]
99 [let h : Integer = i]
100 [if iTrigger = null]
101     new [getODEMxSimMLClass(':', 'Transition::Parameter')]/([g/], [h/], -1)
102 [else]
103     [for (trigger : Trigger | hTransition.trigger)]
104     [if iTrigger = trigger]
105         new [getODEMxSimMLClass(':', 'Transition::Parameter')]/([g/], [h/], [i/])
106     [/if]
107 [/for]
108 [/if]
109 [/let]
110 [/if]
111 [/for]
112 [/let]
113 [/if]
114 [/for]
115 [/template]

```

Listing A.7: StateMachineService.mtl

```

1 [comment encoding = UTF-8 /]
2 [module StateMachineH('http://www.eclipse.org/uml2/3.0.0/UML')/]
3 [import hub::sam::lang::simml::simml2odemx::services:: StateMachineService/]
4 [import hub::sam::lang::simml::simml2odemx::services:: UtilityService /]
5
6 [template public StateMachineH(stateMachine : StateMachine) post (format())]
7 #ifndef [stateMachine.name.toUpper()/_H_]
8 #define [stateMachine.name.toUpper()/_H_]
9
10 #include <[getODEMxSimMLClass('/', 'IncludeStateMachine')/].h>
11
12 class [stateMachine.name/]: public [getODEMxSimMLClass(':', 'StateMachine')/] {
13 public:
14     [stateMachine.name/]/([getODEMxSimMLClass(':', 'Block')/]* context,
15         const odemx::data::Label& label = "[stateMachine.name/]",
16         odemx::base::Simulation& sim = odemx::getDefaultSimulation(),
17         odemx::base::ProcessObserver* obs = 0);
18     virtual ~[stateMachine.name/]() ;
19
20     virtual void initial () ;
21     [let region : Region = stateMachine.getFirstRegion()]
22     [for (state : Vertex | region.getFunctionStates())]
23     void [state.name/]() ;
24 [/for]

```

```
25  [/let]
26  };
27
28  #endif /* [stateMachine.name.toUpper()/_H_ */
29
30  [/template]
```

Listing A.8: StateMachineH.mtl

```
1  [comment encoding = UTF-8 /]
2  [module StateMachineCPP('http://www.eclipse.org/uml2/3.0.0/UML')]
3  [import hub::sam::lang::simml::simml2odemx::services::StateMachineService/]
4  [import hub::sam::lang::simml::simml2odemx::services::UtilityService /]
5
6  [template public StateMachineCPP(stateMachine : StateMachine) post (format())]
7  [let region : Region = stateMachine.getFirstRegion()]
8  #include "[stateMachine.name/].h"
9  #include "[stateMachine.__context.name/].h"
10
11  [stateMachine.name/]:[stateMachine.name/](odemx::simml::Block* context,
12    const odemx::data::Label& label, odemx::base::Simulation& sim,
13    odemx::base::ProcessObserver* obs) :
14    odemx::simml::StateMachine(context, sim, label, obs) {
15  }
16
17  [stateMachine.name/]:~[stateMachine.name/]() {
18  }
19
20  void [stateMachine.name/]:initial () {
21    [region.getPseudostateInitial().getVertexImplementation(region)/]
22  }
23
24  [for (state : Vertex | region.getFunctionStates())]
25  void [stateMachine.name/]:[state.name/]() {
26    [state.getVertexImplementation(region)/]
27  }
28
29  [/for]
30  [/let]
31  [/template]
```

Listing A.9: StateMachineCPP.mtl

```
1  [comment encoding = UTF-8 /]
2  [**
3   * The documentation of the module SimulationService.
4   */]
5  [module SimulationService('http://www.eclipse.org/uml2/3.0.0/UML')]
6
7
8  [**
9   * The documentation of the query getInstances.
10  * @param aPackage
11  */]
```

```

12 [query public getInstances(aPackage : Package) : Collection ( InstanceSpecification ) =
13 aPackage.packageElement->filter( InstanceSpecification )
14 /]
15
16 [query public getClassifier (aPackage : Package) : Set( Classifier ) =
17 aPackage.getInstances(). classifier ->asSet()
18 /]

```

Listing A.10: SimulationService.mtl

```

1 [comment encoding = UTF-8 /]
2 /**
3  * The documentation of the module SimulationCPP.
4  */
5 [module SimulationCPP('http://www.eclipse.org/uml2/3.0.0/UML')]
6 [import hub::sam::lang::simml::simml2odemx::services:: SimulationService /]
7 [import hub::sam::lang::simml::simml2odemx::services:: UtilityService /]
8
9 /**
10 * The documentation of the template SimulationCPP.
11 * @param aPackage
12 */
13 [template public SimulationCPP(aPackage : Package) post (format())]
14 [for ( classifier : Classifier | aPackage. getClassifier () )]
15 #include "[ classifier .name/]h";
16 [/for]
17
18 int main() {
19 odemx::base::Simulation& sim = odemx::getDefaultSimulation();
20 [ generateSimulationConstraints ()/]
21 [for ( instanceSpecification : InstanceSpecification | aPackage.getInstances())]
22 [ instanceSpecification . generateInstances ()/]
23 [/for]
24 [for ( instanceSpecification : InstanceSpecification | aPackage.getInstances())]
25 [ instanceSpecification . generateConnections()/]
26 [/for]
27 }
28 [/template]
29
30 [template public generateInstances( instanceSpecification : InstanceSpecification )]
31 [ instanceSpecification .name/] = new [instanceSpecification . classifier .name/][[
32 instanceSpecification . classifier .generateStateMachine()/]];
33 [for ( slot : Slot | instanceSpecification . slot )]
34 [if valueSpecification .oclIsKindOf( LiteralInteger )]
35 [ instanceSpecification .name/]->[slot.definingFeature.name/] = [valueSpecification .
36 oclAsType( LiteralInteger ).value/];
37 [elseif valueSpecification .oclIsKindOf( LiteralUnlimitedNatural )]
38 [ instanceSpecification .name/]->[slot.definingFeature.name/] = [valueSpecification .
39 oclAsType( LiteralUnlimitedNatural ).value/];
40 [elseif valueSpecification .oclIsKindOf( LiteralString )]
41 [ instanceSpecification .name/]->[slot.definingFeature.name/] = [valueSpecification .
42 oclAsType( LiteralString ).value/];
43 [elseif valueSpecification .oclIsKindOf( LiteralBoolean )]

```

```
41      [ instanceSpecification .name/]->[slot.definingFeature.name/] = [ valueSpecification .
      oclAsType(LiteralBoolean).value/];
42    [/if]
43  [/for]
44 [/for]
45 [/template]
46
47 [template public generateConnections( instanceSpecification : InstanceSpecification )]
48 [for ( slot : Slot | instanceSpecification .slot )]
49   [for ( valueSpecification : ValueSpecification | slot .value)]
50     [if valueSpecification .oclIsKindOf( InstanceValue)]
51       [ instanceSpecification .name/]->[slot.definingFeature.name/] = [ valueSpecification .
      oclAsType(InstanceValue).instance .name/];
52     [/if]
53   [/for]
54 [/for]
55 [/template]
56
57 [template public generateStateMachine( classifier : Classifier )]
58 [if classifier .oclIsKindOf( Class)][if classifier .oclAsType(Class).isActive]new [ classifier .
  oclAsType(Class). classifierBehavior .oclAsType(StateMachine).name/](NULL, "[classifier .
  oclAsType(Class). classifierBehavior .name/]", sim)[/if][/if]
59 [/template]
```

Listing A.11: SimulationCPP.mtl



## B Anhang

### B.1 Fachbegriffe

| Deutsch                      | Englisch                    | Sprachdefinition |
|------------------------------|-----------------------------|------------------|
| Abteil                       | compartment                 | UML              |
| Aggregation                  | shared aggregation          | UML              |
| Aktivitätsdiagramm           | activity diagram            | UML              |
| Assoziation                  | association                 | UML              |
| Bedingung                    | constraint                  | UML              |
| Block                        | block                       | SysML            |
| Blockbedingung               | block constraint property   | SysML            |
| Blockdefinitionsdiagramm     | block definition diagram    | SysML            |
| Blockeigenschaft             | block property              | SysML            |
| Blockinstanz                 | block instance              | SysML            |
| Blockreferenz                | block reference property    | SysML            |
| Blockteil                    | block part property         | SysML            |
| Blockwert                    | block value property        | SysML            |
| Datentyp                     | data type                   | UML              |
| Eigenschaft                  | property                    | UML              |
| Einheit                      | unit                        | SysML            |
| Element                      | item                        | SysML            |
| Elementfluss                 | item flow                   | SysML            |
| Flusseigenschaft             | flow property               | SysML            |
| Flussport                    | flow port                   | SysML            |
| Flussspezifikation           | flow specification          | SysML            |
| Instanz-Spezifikationen      | instance specification      | UML              |
| Internes Blockdiagramm       | internal block diagram      | SysML            |
| gekapselt                    | encapsulated                | SysML            |
| Klasse                       | class                       | UML              |
| Klassendiagramm              | class diagram               | UML              |
| Komponente                   | component                   | UML              |
| Kompositionsbeziehung        | composite aggregation       | UML              |
| Kompositionsstrukturdiagramm | composite structure diagram | UML              |
| Konnektor                    | connector                   | UML              |
| Objektfluss                  | object flow                 | UML              |
| Operation                    | operation                   | UML              |
| Paket                        | package                     | UML              |
| physikalische Größe          | quantity kind               | SysML            |
| Port                         | port                        | UML              |
| Verhaltensbeschreibung       | behavior                    | UML              |
| Werte Verteilung             | distributed property        | SysML            |
| Werttyp                      | value type                  | SysML            |
| Zustandsmachinendiagramm     | state machine diagram       | UML              |

# Literaturverzeichnis

- [1] Bjarne Stroustrup. *The C++ Programming Language: Special Edition*. Addison-Wesley Longman, Amsterdam, 2000.
- [2] David Harel. A Visual Formalism for Complex Systems. Technical Report 8/1987, Science of Computer Programming, North Holland, 1987.
- [3] Claus Draeby. Unit++. <http://unitpp.sourceforge.net/>, August 2011.
- [4] Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides, Craig Larman. *Design Patterns with Applying UML and Patterns and Iterative Development*. Verlag Addison-Wesley, 2005.
- [5] Hartmut Bossel. *Systeme, Dynamik, Simulation - Modellbildung, Analyse und Simulation komplexer Systeme*. Books on Demand GmbH, Norderstedt, 2004.
- [6] James O. Henriksen. Slx: pyramid power. In *Proceedings of the 31st conference on Winter simulation: Simulation—a bridge to the future - Volume 1*, WSC '99, pages 167–175, New York, NY, USA, 1999. ACM.
- [7] International Council on Systems Engineering (INCOSE). Systems Engineering Vision 2020. Technical Report INCOSE-TP-2004-004-02, 2007.
- [8] Joachim Fischer, Klaus Ahrens. *Objektorientierte Prozeßsimulation in C++*. Addison-Wesley (Deutschland) GmbH, 1996.
- [9] Leon McGinnis, Volkan Ustun. A simple Example of SysML-driven Simulation. Technical Report GA 3032-0205, School of Mechanical Engineering, Georgia Institute of Technology, Atlanta, USA, 2009.
- [10] Miro Samek, Ph. D. *Practical Statecharts in C/C++*. CMP Books, imprint of CMP Media LLC, 2002.
- [11] OMG. MOF Model to Text Transformation Language, v1.0. Technical Report formal/2010-05-06, OMG, 2008.
- [12] OMG. Object Constraint Language (OCL), Version 2.2. Technical Report formal/2010-02-01, OMG, 2010.
- [13] OMG. OMG Systems Modeling Language (OMG SysML) Version 1.2. Technical Report formal/2010-06-02, OMG, 2010.

- [14] OMG. OMG Unified Modeling Language (OMG UML), Superstructure und Infrastructure Version 2.3. Technical Report formal/2010-05-06, OMG, 2010.
- [15] OMG. Xtext Documentation 2.0. Technical report, itemis AG, 2010.
- [16] Paul A. Fishwick. *Simulation Model Design and Execution: Building Digital Worlds*. Prentice Hall, Inc., 1995.
- [17] Peer Hausding. Studienarbeit. Systemmodellierung mit SysML. Lehr- und Forschungseinheit Systemanalyse, Institut für Informatik, Humboldt-Universität zu Berlin, 2010.
- [18] Thomas A. Johnson. Integrating Models and Simulations of continuous dynamic System Behavior into SysML. Master's thesis, Georgia Institute of Technology, 2008.
- [19] Tim Weilkiens. *Systems Engineering mit SysML/UML*. dpunkt.verlag GmbH, 2. Auflage 2008.
- [20] Toby Neumann. Diplomarbeit. Abbildung von SDL-200 nach Java. Lehr- und Forschungseinheit Systemanalyse, Institut für Informatik, Humboldt-Universität zu Berlin, 2000.
- [21] International Telecommunication Union. *ITU-T Z.100. Specification und Description Language (SDL)*, 2002.



# Abbildungsverzeichnis

|     |                                                            |    |
|-----|------------------------------------------------------------|----|
| 2.1 | Bestandteile von Systemen . . . . .                        | 12 |
| 2.2 | SysML-Diagrammtypen . . . . .                              | 13 |
| 2.3 | Block- und Wertetyp-Definitionen . . . . .                 | 17 |
| 2.4 | Block Systemblock . . . . .                                | 18 |
| 2.5 | Blockbedingung des Blockes Block3 . . . . .                | 19 |
| 2.6 | Instanz des Blockes Systemblock . . . . .                  | 20 |
| 3.1 | Stereotyp für SimML-Zustandsereignis . . . . .             | 27 |
| 3.2 | Zustandsmaschine mit zusätzlichen Abhängigkeiten . . . . . | 29 |
| 3.3 | Beispiel Aktionsprache: Block A . . . . .                  | 30 |
| 3.4 | Initiale Systemkonfigurationsdefinition . . . . .          | 33 |
| 4.1 | Überblick der ODEMX-Erweiterungen . . . . .                | 39 |
| 5.1 | Blockdefinitionsdiagramm . . . . .                         | 58 |
| 5.2 | Zustandsdiagramm . . . . .                                 | 65 |
| 5.3 | Simulationsbeschreibung . . . . .                          | 71 |



# Tabellenverzeichnis

|     |                                                       |    |
|-----|-------------------------------------------------------|----|
| 3.1 | Initiale Belegung vordefinierter Wertetypen . . . . . | 23 |
| 3.2 | Mengentypen mehrwertiger Blockeigenschaften . . . . . | 24 |
| 3.3 | Mengentypen mehrwertiger Blockeigenschaften . . . . . | 32 |



# Index

- abfragend, 54, 57
- Abschlussereignis, 27, 68, 69
- Abteil, 14, 18
- Aggregationsbeziehung, 15
- aktiv, 53, 54, 56, 66, 71
- Aktivität, 34
- Aktivitätsdiagramm, 17
- Anwendungsfalldiagramm, 17
- Assoziation, 13, 14, 20
  
- Bedingung, 17–19, 26, 28, 35, 48, 56, 64, 66, 69, 74
- Bedingungsblock, 15, 17–19, 74
- Bedingungsparameter, 18, 19
- Bindungskonnektor, 19
- Block, 14–20, 23, 25, 28, 29, 33, 45, 48, 51–54, 56, 57, 61, 62, 66, 68, 70, 71
- Blockbedingung, 15, 18, 19
- Blockdefinitionsdiagramm, 14, 16, 20, 57
- Blockeigenschaft, 14, 15, 18, 20, 23–25, 28–30, 32–35, 48, 53, 54, 57, 62, 64
- Blockreferenz, 15, 16, 28, 29, 33, 54, 57
- Blockteil, 15, 16, 28, 29, 33, 54, 57, 61
- Blockwert, 15–17, 19, 20, 28, 31, 33, 54, 57, 66, 68, 69, 71
- Boolean, 14, 22, 23
  
- Complex, 14, 22, 23, 31
  
- Datentyp, 13, 14
  
- Effekt, 27–29, 48–50, 64, 66, 68
  
- Eigenschaft, 13–16, 18–20, 23, 28, 31, 33–35
- Einheit, 14, 16, 75
- Element, 15
- Elementfluss, 15
- Entscheidungsknoten, 26, 49, 50, 63, 64, 66, 68
- Ereignis, 18, 27, 47, 64, 74
  
- Flusseigenschaft, 15
- Flussport, 15, 19
- Flussspezifikation, 15
  
- gekapselt, 14
  
- Initialknoten, 26, 49, 63, 66, 67
- Instanz-Spezifikation, 14, 20, 71
- Integer, 14, 22
- Interaktionsdiagramm, 17
- Internes Blockdiagramm, 14, 16, 18
  
- Klasse, 13, 14
- Klassendiagramm, 16
- Klasseneigenschaft, 14
- Knoten, 25, 27, 63, 65
- Komponente, 13
- Kompositionsbeziehung, 15
- Kompositionsstrukturdiagramm, 16
- Konnektor, 13, 14
  
- Link, 20
  
- Numbers, 14
  
- Objektfluss, 17
- Operation, 14, 23, 25, 32, 54, 57, 60
  
- Paket, 14, 16, 22, 34

- Parameter, 19, 25, 29, 54
- Parameterdiagramm, 17–19
- physikalische Größe, 14, 16, 75
- Port, 13–15, 18
- Pseudozustand, 18, 25, 26, 49, 50
- public, 22, 23, 34
  
- Rückgabewert, 25, 31, 32, 54
- Real, 14, 22, 23, 31
- Region, 25
  
- Schnappschuss, 20
- Schnittstelle, 15
- Sequence, 57, 60
- Simulationsbeschreibung, 71
- statisch, 54
- String, 14, 22, 23, 28, 31, 47
  
- Tag, 28, 64, 70, 75
- Terminierungsknoten, 26, 30, 49, 63, 66, 69
- Transition, 18, 25–28, 47, 49, 50, 63–66, 69
- Trigger, 18, 26, 27, 49, 50, 56, 63, 65
  
- Verhalten, 48
- Verhaltensbeschreibung, 17, 18
- Vorgabewert, 29
  
- Wächter, 26, 49, 50, 56, 63–66, 69
- Wächterbedingung, 50
- Wert, 60
- Wertetyp, 14–16, 19, 20, 22, 23, 25, 28, 31, 33, 47, 53, 60, 75
- Wertetypeigenschaft, 22, 23, 34
- Werte Verteilung, 16
- Wertspezifikation, 71
  
- Zeitereignis, 27, 28, 33, 66, 69
- Zustand, 18, 25, 26, 47–50, 63, 64, 66, 67, 69
- Zustandsautomat, 18
- Zustandsdiagramm, 17–19, 65
- Zustandsereignis, 27, 33, 35, 64, 66, 69
- Zustandsmaschine, 23, 25, 26, 28, 29, 33, 34, 45, 47–49, 51, 56, 61–63, 66, 67, 70, 71

# C Erklärungen

## Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Berlin, ..... ..

## Einverständniserklärung

Ich erkläre hiermit mein Einverständnis, dass die vorliegende Arbeit in der Bibliothek des Institutes für Informatik der Humboldt-Universität zu Berlin ausgestellt werden darf.

Berlin, ..... ..