Carl v. Ossietzky
UNIVERSITÄT
OLDENBURG
Fachbereich Informatik

Diplomarbeit

Hintergrundsimulation von SPS-Automaten

Vorgelegt von : Michael Möller
Erstprüfer : Prof. Dr. Ernst-Rüdiger Olderog
Zweitprüfer : Dipl.-Inf. Henning Dierks

Oldenburg, den 16. Juli 1999

Zusammenfassung

Im Editorsystem Moby/plc[1] können auf der Basis von SPS-Automaten Systeme für Realzeitanwendungen in einer graphischen Notation entworfen und automatisch implementiert werden. Zu einem ersten Test der entworfenen Steuerlogik – bevor Modelchecking und andere formale Methoden Anwendung finden – bietet dieses System die Möglichkeit der interaktiven Simulation. Um diese Möglichkeit für etwas größere Testfälle handhabbar zu machen, soll der Simulator um Hintergrundsimulation erweitert werden.

In dieser Diplomarbeit wird deshalb ein Konzept entwickelt, das es erlaubt Eingaben für einen Simulationslauf a priori festzulegen, einen Simulationslauf aufzuzeichnen und einen solchen in angemessener Form graphisch darzustellen. Dieses Konzept wird mit Hilfe der Moby-Klassenbibliothek implementiert und mit dem vorhandenen System gekoppelt.

[1]Modeling of distributed Systems / Programmable Logic Controllers

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Der Entwurf von Steuerungskomponenten für Systeme des täglichen Lebens wird seit geraumer Zeit zusehends komplexer. Diese Komplexität wächst noch enorm im Falle von Anwendungen, die Realzeitanforderungen genügen müssen. Aber nicht nur die Komplexität solcher Steuerungen nimmt zu, sondern auch deren Sicherheitsanforderungen. So kann es z.B. nicht toleriert werden, wenn ein Bahnübergang erst mit einiger Verzögerung geschlossen wird, da in einem solchen Fall menschliches Leben in Gefahr wäre.

Die Abteilung Semantik an der Universität Oldenburg befaßt sich daher seit einiger Zeit mit der systematischen Entwicklung korrekter Software. In diesem Zusammenhang wird ein transformationeller Ansatz angestrebt, bei dem aus einer korrekten Spezifikation durch korrekte Transformationen eine korrekte Implementierung entsteht. Eine von der Abteilung entwickelte Methode, die diesem Ansatz gerecht werden soll, ist die Spezifikation durch SPS-Automaten.

SPS-Automaten

Timed Automata

Realzeit

SPS-Automaten sind eine erweiterte Form von endlichen Automaten, die von Henning Dierks in [Die97] eingeführt wurden. Die Erweiterungen betreffen, wie auch bereits bei *Timed Automata* [AD94], im wesentlichen Realzeitaspekte. SPS-Automaten haben eine formal wohldefinierte Semantik im *Duration Calculus*[1], so daß *formale Methoden*[2] angewandt werden können. Ein Vorteil gegenüber Timed Automata liegt darin, daß SPS-Automaten aus einer Spezifikation, die durch eine Menge implementierbarer Formeln beschrieben ist, synthetisiert werden können. Da diese Automatenklasse speziell für den Entwurf von zyklischen Realzeitsystemen entwickelt wurde, liegt ihr Vorteil auch in ihrer direkten Implementierbarkeit, so daß ein und dasselbe mathematische Modell sowohl zur Verifikation als auch zur Implementierung benutzt wird, und so ein Bruch im Entwicklungsprozeß vermieden werden kann.

[1] Hierbei handelt es sich um eine Logik und ein Kalkül für Realzeitsysteme, die in [ZHR91] eingeführt wurden.

[2] siehe hierzu z.B. [JW97]

Der Entwurf eines Realzeitsystems am Computer auf der Basis von SPS-Automaten wird durch das graphische Editorsystem MOBY/PLC unterstützt, dessen Grundlagen in der Projektgruppe IEKOΣ gelegt wurden. Eine erste Einführung in die Bedienung des Programms bietet [BFvG+97].

Während des Entwurfs eines Realzeitsystems mit SPS-Automaten soll der Benutzer des Editor-Systems bereits die Möglichkeit erlangen, das von ihm spezifizierte System zu *testen*, d.h. er soll sehen können, wie das System auf bestimmte Eingaben mit fortschreitender Zeit reagiert. Hierzu wird das graphisch beschriebene mathematische Modell simuliert. Bei kleineren Teilproblemen kann dieser erste Test interaktiv erfolgen, d.h. der Benutzer teilt dem Simulator in einem Dialog mit, welche Eingabesignale vorliegen und bestimmt wieviel Simulationszeit vergeht, bis neue Signale anliegen sollen bzw. bis die Ausgabe des Simulators kontrolliert wird. Parallel zum Editorsystem wurde zu diesem Zweck von Johannes Schröder in seiner Diplomarbeit ([Sch97]) ein SPS-Simulator entwickelt, der genau diesen Anforderungen genügt.

Simulation

Soll jedoch ein Test für mehrere Eingabesignale über einen längeren Zeitraum erfolgen, bei denen zudem häufige Signalwechsel stattfinden, und bei dem die Ausgangssignale für den kompletten Simulationszeitraum von Interesse sind, ist die vollständig interaktive Simulation nicht mehr praktikabel. Zum einen kann es eine Vielzahl von Eingängen geben, die zu unterschiedlichen Zeitpunkten ihre Werte ändern sollen, zum anderen läßt die fortschreitende Simulationszeit keinen Rückblick auf Ausgaben in der (Simulations-) Vergangenheit zu, so daß eine effektive Berechnung der Simulation durch Benutzerinteraktion gebremst wird. In diesem Problembereich soll die Erweiterung des Simulators, wie sie in dieser Diplomarbeit entworfen und implementiert wird, Abhilfe schaffen.

1.1 Aufgaben und Ziele

Die erste Aufgabe dieser Diplomarbeit besteht darin, den Simulationsvorgang aufzuzeichnen. Für eine Simulation (mit irgendwie bestimmten Eingabesignalen für jeden Simulationszeitpunkt) sollen alle relevanten Ereignisse eines SPS-Automaten-Systems aufgezeichnet werden können. Bei diesen Ereignissen handelt es sich sowohl um die nach außen sichtbaren Wertänderungen an Eingangs- und Ausgangssignalen, als auch um die internen Zustandswechsel der Automaten, sowie deren lesender Zugriff auf Eingangssignale. Für diese Aufzeichnung bzw. Beschreibung der exakten zeitlichen Abfolge von Ereignissen, die für die Simulation relevant sind, soll im folgenden der Begriff *Protokoll* benutzt werden.

Protokoll

Ausgehend von diesem low-level (Simulations-)Protokoll soll eine Eingabesprache entwickelt werden, das Eingabeprotokoll, das vom Simulator interpretiert wird, um eine Simulation „nachzuspielen“. Bei geeigneten Erweiterungen der Sprache des Ausgabeprotokolls kann hierdurch eine kompakte Eingabesprache entstehen, bei der nichtspezifizierte Ereignisse durch Simulation neu berechnet werden. Dieses ist im allgemeinen dann notwendig, wenn ein Eingabeprotokoll direkt vom Benutzer, z.B. mit einem Texteditor, erstellt wurde. In einem solchen Protokoll wird i.a. nur die Belegung der Eingabesignale für den Simulationszeitraum beschrieben, nicht aber z.B. die Zustandswechsel der Automaten.

Wird ein Eingabeprotokoll nun benutzt, um eine Simulation durchzuführen, bei der der Benutzer nicht mehr interaktiv eingreifen muß und bei der die Simulationsergebnisse in einem Ausgabeprotokoll festgehalten werden, so soll im folgenden hierfür der Begriff *Hintergrundsimulation* verwendet werden. Bei einer solchen Hintergrundsimulation muß dem Benutzer ein Werkzeug an die Hand gegeben werden, mit dem er die Simulation auswerten kann. In der Diplomarbeit werden zu diesem Zweck zwei Wege verfolgt. Zum einen wird es dem Benutzer durch zuvor festgelegte Bedingungen ermöglicht, bestimmte Systemzustände zu erkennen, die durch Meldungen oder den Halt der Simulation mitgeteilt werden können. Desweiteren wird eine Visualisierungskomponente entwickelt, deren Aufgabe es ist, ein Ausgabeprotokoll in angemessener Weise auf dem Bildschirm zu präsentieren, so daß eine Beurteilung des Simulationslaufs ermöglicht wird.

Hintergrund-simulation

1.2 Inhalt der Diplomarbeit

Die Diplomarbeit wird im nächsten Kapitel in die notwendigen Grundlagen einführen. Hierbei handelt es sich um eine knappe Definition der SPS-Automaten und um eine Beschreibung des vorhanden Simulators – hierbei wird vor allem auf das in [Sch97] gewählte Simulationskonzept, die ereignis-diskrete Simulation, eingegangen. Hierauf aufbauend erfolgt eine Betrachtung der Semantik, durch die eine Rechtfertigung für die Simulatorimplementierung gegeben wird.

Im darauffolgenden Kapitel wird der Entwurf der Simulatorerweiterung und der Protokollsprache in drei Abschnitten beschrieben. Zunächst erfolgt eine Beschreibung des Gesamtsystems. Dieses wird durch den Entwurf eines simulatornahen Protokolls, dessen interne Struktur und die Erweiterungen, die Aufgrund dieses Protokolls am Simulator durchzuführen sind, verfeinert. Im zweiten Schwerpunkt werden die externen Protokolle entwickelt, hierbei insbesondere textuelle Repräsentationen und die Programmkomponenten zu deren Verarbeitung. Interpretation bzw. Erzeugung der textuellen in eine bzw. aus einer programminternen Repräsentation wird ebenso beschrieben, wie der Mechanismus zur Übersetzung von und in das simulatornahe Protokoll. Der letzte Teil dieses Kapitels befaßt sich mit der Visualisierungskomponente, die zur Darstellung der Simulationsergebnisse aus der Hintergrundsimulation entwickelt wird.

Das vierte Kapitel beschreibt Details zur Implementierung der Programmkomponenten, die im vorangegangen Kapitel entworfen wurden. Insbesondere wird auch auf die Integration der neuen Komponenten in das Gesamtsystem eingegangen.

Durch eine Beispielsitzung soll das so entstandene Simulationssystem in seiner praktischen Anwendung vorgestellt werden.

Abschließend erfolgt eine Betrachtung möglicher bzw. wünschenswerter Erweiterungen, die leider in dieser Diplomarbeit nicht realisiert wurden, weil sie ein zu weites Gebiet abdecken würden.

Kapitel 2

Grundlagen

Dieses Kapitel soll in die Grundlagen einführen, die für das Verständnis der folgenden Kapitel notwendig sind. Zu diesen Grundlagen gehört ein Überblick über die SPS-Automaten, um die Implementierung für deren Simulation nachvollziehen zu können, sowie eine Beschreibung der bisherigen Simulatorimplementierung. Auf diese Weise sollen die Entwurfsentscheidungen für die Unterstützung der Hintergrundsimulation nachvollziehbar gemacht werden. Simulation und formale Semantik werden anschließend in einen Zusammenhang gebracht, um zu zeigen, daß die Implementierung des Simulators sinnvoll ist. Zum Schluß des Kapitels wird noch kurz auf die Moby-Klassenbibliothek eingegangen.

2.1 SPS-Automaten

Die von Henning Dierks entwickelten SPS-Automaten bilden die formale Grundlage für das vorhandene und in dieser Arbeit zu erweiternde Simulationssystem. Bei SPS-Automaten handelt es sich um eine erweiterte Form von endlichen Automaten, die Echtzeitanforderungen berücksichtigen. Die Semantik dieser Automaten wird in [Die97] durch Formeln des *Duration Calculus* [ZHR91, Zho93] ausgedrückt, so daß sie formal gegeben ist. Diese Semantik macht einen Hauptunterschied zu den bereits in [AD94] definierten Timed Automata aus, da sie die Synthese eines SPS-Automaten aus einer Spezifikation ermöglicht, die durch *Implementables*, einer Teilmenge von implementierbaren Formeln des Duration Calculus, angegeben wird.

Duration Calculus

In [DT98] wird diese Semantik erweitert, um hierarchischen SPS-Automaten in gleicher Weise eine formale Semantik auf der Basis des Duration Calculus zu geben. Die Hierarchie wurde eingeführt, um die Ausdrucksmöglichkeiten der SPS-Automaten zu erweitern und so komplexere Zeitanforderungen formulieren zu können. Als abkürzende Schreibweise wird im folgenden für diese Semantik-Definition der Begriff *DC-Semantik* benutzt. Zudem wird in

DC-Semantik

[DFMV98] für SPS-Automaten eine operationelle Semantik auf der Basis von Timed Automata angegeben und ein Beweis erbracht, daß diese Semantik mit der DC-Semantik verträglich ist. Die Abbildung auf Timed Automata ist von Vorteil, da so andere Verifikationswerkzeuge, wie z.B. Kronos[1], genutzt werden können. Diese Semantik, die im folgenden kurz als *TA-Semantik* bezeichnet wird, wird im Abschnitt 2.3.1 zusammengefaßt beschrieben.

TA-Semantik

Gerade bei Entwurf und Implementierung eines Simulators für eine formale Spezifikationssprache wie SPS-Automaten ist es wichtig, daß eine Simulation sich auch so verhält, wie man es aufgrund der formalen Semantik erwarten würde. Durch die Implementierung des Simulators wird nämlich eine Semantik den SPS-Automaten induziert, die im folgenden als *Simulator-Semantik* bezeichnet wird. Da der Simulator auf operationelle Weise SPS-Automaten simuliert, soll eine Analogie der Simulator-Semantik mit der formalen TA-Semantik angegeben werden, so daß hierüber der Zusammenhang zur DC-Semantik gegeben ist und damit die Verwendung des Simulators für erste Tests einer Spezifikation legitimiert wird. Dieser Zusammenhang wird aber erst in Abschnitt 2.3.2 dargestellt, nachdem die Arbeitsweise des Simulators erläutert wurde (s. Abschnitt 2.2.3).

Simulator-Semantik

Die Entwicklung der SPS-Automaten wurde durch eine Zusammenarbeit mit der Firma INSY GmbH, Berlin, im BMBF-Projekt *UniForM*[2] [KBPO$^+$96] angeregt. Dort benutzt man Speicherprogrammierbare Steuerungen – kurz als SPS bezeichnet – z.B. zur Realisierung von Straßenbahnsignalanlagen. Die Idee war es, für diese Bausteine ein formales Konzept zur Programmierung zu entwickeln, so daß die Testkosten, die bei der Zulassung der entwickelten Steuerungen anfallen, reduziert werden können. Mit den bisherigen Programmiermethoden, die eher an Hardwareentwurf und Maschinensprache erinnern, ist eine formale Betrachtung der Programmierung äußerst schwierig, da eine exakte, mathematisch beschriebene Semantik nicht gegeben ist. SPS-Automaten schaffen hier Abhilfe, da durch die graphische Notation eine intuitiv erlernbare Programmierung ermöglicht wird, bei deren Entwicklung jedoch besonders auf die Anwendbarkeit formaler Methoden und automatischer Übersetzbarkeit geachtet wurde. Die so entstanden SPS-Automaten sind jedoch nicht nur für die Programmierung von SPSen geeignet, sondern für alle Steuereinheiten, die dem im folgenden als *SPS-Zyklus* beschriebenen Schema bezüglich ihres Ein-/Ausgabeverhaltens genügen.

UniForM

SPS

SPS-Zyklus

2.1.1 Der SPS-Zyklus

Inspiriert durch das Verhalten der SPSen arbeiten auch die SPS-Automaten nach einem zyklischen Schema, das in Abbildung 2.1 dargestellt ist. Im

[1] Kronos ist ein Modelchecker, wobei die Modellbeschreibung durch Timed Automata erfolgt. Eine Beschreibung findet sich in [OY93].

[2] **Uni**verselle Entwicklungsumgebung für **For**male **M**ethoden

ersten Schritt eines jeden Zyklus werden die Eingänge gelesen. Der Wert eines Eingangs, bzw. einer Eingangsvariable zu exakt diesem Zeitpunkt ist für die Berechnung relevant. Diesen Vorgang bezeichnet man auch als *Pollen* der Eingänge. Nach dem Pollen aller Eingänge folgt die Berechnungsphase, in der die neuen Werte für Ausgangssignale oder -variablen und der neue Automatenzustand bestimmt werden. In dieser Phase werden auch Zeitrestriktionen berücksichtigt, die durch *Timer* (siehe hierzu Abschnitt 2.1.2) ausgedrückt werden. In der letzten Phase werden die berechneten Werte auf die Ausgangssignale oder -variablen geschrieben, so daß die Berechnung nach außen sichtbar wird. Auch wird in dieser Phase für das Setzen der Timer gesorgt, da dieses ebenfalls als Schreibvorgang zu betrachten ist – in die Timer wird ein neuer Startwert geschrieben.

Pollen

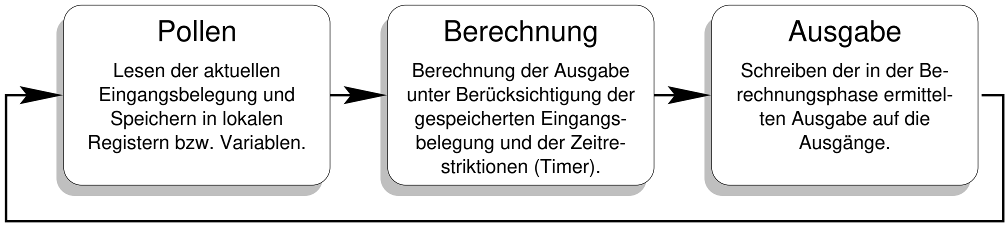

Abbildung 2.1: SPS-Zyklus

2.1.2 Hierarchische endliche Automaten mit Zeit

Der SPS-Automat ist eine Erweiterung des Konzeptes der endlichen Automaten um Aspekte, die Realzeitanforderungen ausdrücken. So verfügt ein SPS-Automat wie ein endlicher Automat über eine Zustandsmenge, und ein Eingabealphabet. Ebenso wird durch eine Ausgabefunktion jedem Zustand des SPS-Automaten ein Element des Ausgabealphabets zugeordnet. Zusätzlich hat ein SPS-Automat einen Parameter, der die maximale Zyklusdauer angibt, und der häufig mit ε (Epsilon) bezeichnet wird. Es handelt sich um eine obere Schranke für die Zeit, die für einen Durchlauf durch den SPS-Zyklus, benötigt wird, so daß die einzelnen Phasen beliebig (unter Beibehaltung ihrer Reihenfolge) über dieses Zeitintervall verteilt sein können. Da die Zyklusdauer aber nur eine obere Schranke ist, können in dieser Zeit entsprechend auch mehrere Zyklen stattgefunden haben.

maximale Zyklusdauer ε

Um sicherzustellen, daß eine Eingabe vom SPS-Automaten gelesen wird, ist eine Stabilitätsanforderung an die Eingabe in Abhängigkeit des ε-Parameters zu stellen. Weil die Phasen des SPS-Zyklus beliebig über das ε-Zeitintervall verteilt sein können, muß eine Eingabe über einen Zeitraum von 2ε stabil bleiben, um garantieren zu können, daß sie vom SPS-Automaten registriert wird. Sonst könnte der Fall eintreten, daß der SPS-Automat unmittelbar vor der Eingabeänderung pollt, für den Zyklus die maximale Zyklusdauer voll ausschöpft und erst kurz vor dem Ablaufen der zweiten ε-Zeitspanne wieder pollt. Somit ist das Halten der Eingabe für nur eine maximale Zyklusdauer nicht ausreichend.

Desweiteren sind den Zuständen der SPS-Automaten Beschriftungen zugeordnet, die es erlauben, untere Schranken für einen Zustandswechsel festzulegen. Jedem Zustand wird eine Zeitkonstante und eine Teilmenge des Eingabealphabets zugeordnet, die besagen, welche Eingaben für die gegebene Zeitdauer nach Erreichen des Zustandes ignoriert werden, so daß kein Zustandswechsel durch diese Eingaben erfolgt. Diese Bedeutung ist vergleichbar damit, als würde beim Erreichen eines Zustandes eine Stoppuhr, der *Timer*, gestartet. Wird der Zustand lange genug beibehalten, also wenn die festgelegte Zeit (und damit der Timer) abgelaufen ist, so werden alle Eingaben wieder akzeptiert. Ein Timer legt also fest, wie lange welche Eingaben für Zustandswechsel ignoriert werden sollen.

Timer

Um mehr Ausdrucksmöglichkeiten für Zeitrestriktionen bereitzustellen, wurden die SPS-Automaten um ein Hierarchiekonzept erweitert. Eine Menge von Automatenzuständen kann zusammengefaßt und wiederum mit einem Timer versehen werden. Die Bedeutung eines solchen Timers ist es, daß die Zustände erst dann mit Eingaben der spezifizierten Menge verlassen werden können, wenn der Timer abgelaufen ist. Innerhalb der Zustandsmenge können hingegen beliebig viele Zustandswechsel stattfinden.

Hierarchie

2.1.3 Definition eines SPS-Automaten

Die Entwicklung der TA-Semantik in [DFMV98] beruht auf der ersten Version der SPS-Automaten aus [Die97] und behandelt deshalb noch nicht die Hierarchie, die in [DT98] eingeführt wurde. Da der Zusammenhang der Semantiken zunächst anhand dieser Definition gezeigt wird, soll hier auch die ursprüngliche Definition wiedergegeben werden.

SPS-Automat

Definition 2.1: Ein Tupel $\mathcal{A} = (Q, \Sigma, \delta, q_0, \varepsilon, S_e, S_t, \Omega, \omega)$ ist ein (deterministischer) *SPS-Automat*, wenn folgende Eigenschaften erfüllt sind:

- Q ist eine nichtleere, endliche Menge von *Zuständen*
- Σ ist eine nichtleere, endliche Menge von Eingabesymbolen, das *Eingabealphabet*
- δ ist eine Funktion vom Typ $Q \times \Sigma \to Q$, die *Transitionsfunktion*
- $q_0 \in Q$ ist der *Anfangszustand*
- $\varepsilon > 0$ ist obere Schranke für die Zyklusdauer

- S_t ist eine Funktion vom Typ $Q \to \mathbb{R}_{\geq 0}$, die jedem Zustand eine *Verzögerungszeit* zuordnet
- S_e ist eine Funktion vom Typ $Q \to \wp(\Sigma)$, die jedem Zustand die Eingabesymbole zuordnet, die in der durch S_t festgelegten Zeitspanne ignoriert werden sollen[3]
- Ω ist eine nichtleere, endliche Menge von Ausgabesymbolen, das *Ausgabealphabet*
- ω ist eine Funktion vom Typ $Q \to \Omega$, die *Ausgabefunktion*

und es gilt die Bedingung

$$\forall q \in Q, a \in \Sigma : S_t(q) > 0 \wedge a \notin S_e(q) \Rightarrow \delta(q, a) \neq q \tag{2.1}$$

□

Die ersten vier Komponenten eines SPS-Automaten entsprechen hiernach denen eines endlichen Automaten. Die weiteren Bestandteile erlauben das Ausdrücken der Echtzeitaspekte des Automaten. Die Benutzung von Timern wird durch die Zustandsbeschriftungen S_t und S_e ausgedrückt. Ein SPS-Automat mit einer Beschriftung $S_t(q) > 0$ kann seinen Zustand q mit einem Eingabesymbol $a \in S_e(q)$ erst nach der Zeit $S_t(q)$ verlassen. Für die gegebene Zeitspanne werden also die Eingaben aus $S_e(q)$ ignoriert, so daß ausgedrückt werden kann, daß ein Zustand eine bestimmte Zeitspanne nicht verlassen werden darf. Umgekehrt bedeutet dieses für alle Elemente der Menge $(\Sigma \setminus S_e(q))$, daß der Zustand q auch verlassen werden kann, wenn in ihm noch nicht die geforderte Zeitspanne verweilt wurde. Diese Elemente, mit denen ein vorzeitiger „Notausstieg" aus dem Zustand möglich ist, werden als *Exitsymbole* bezeichnet. Gleichung 2.1 besagt nun, daß ein solcher „Notausstieg" nicht wieder in den Ausgangszustand führen kann. Hierbei handelt es sich jedoch nicht um eine echte Einschränkung, denn die Forderung kann leicht erreicht werden, indem alle Eingaben $a \in \Sigma$ mit $\delta(q, a) = q$ für $q \in Q$ in die Menge $S_e(q)$ aufgenommen werden, denn ein neuer Zustand wird durch die Eingabe a nicht angenommen, also kann sie auch ignoriert werden. Die Bedingung muß lediglich erfüllt sein, um unnötige Komplikationen in der Definition der DC-Semantik zu vermeiden.

Beispiel 2.1: In Abbildung 2.2 wird für solch einen SPS-Automaten ein sehr einfaches Beispiel graphisch dargestellt. Der Automat stellt eine Ansteuerung für ein Blinklicht dar, dessen Eingang das Blinklicht an- bzw. ausschaltet und dessen Ausgang die Glühbirne des Blinklichts ansteuert. Der SPS-Automat startet im Zustand *aus* und mit der Ausgabe *aus*, wobei das Ausgabealphabet durch $\Omega = \{an, aus\}$ definiert sei. Das

[3] Mit $\wp(X)$ sei die Potenzmenge der Menge X bezeichnet, also $\wp(X) = \{Y | Y \subseteq X\}$

Eingabealphabet sei ebenso definiert durch $\Sigma = \{an, aus\}$. Sobald der SPS-Automat mindestens 20 Zeiteinheiten (ZE) in diesem Startzustand verblieben ist, kann er durch die Eingabe *an* in den Zustand *an* wechseln, wobei die Ausgabe ebenfalls zu *an* geändert wird. In diesem Zustand wird nur die Eingabe *an* ignoriert, so daß der Automat durch die Eingabe *aus* sofort zurückwechseln kann. Bleibt die Eingabe hingegen *an*, bzw. ist sie zu den Pollzeitpunkten jeweils *an*, so muß der Automat für mindestens 20 ZE in dem Zustand verweilen. Die Verweilzeit hängt hierbei von der nicht vorherbestimmbaren Zykluszeit ab, eine obere Schranke errechnet sich aber aus der Verzögerungszeit und der doppelten maximalen Zykluszeit, und demnach beträgt sie mit $\varepsilon = 10$ für diesen Automaten 40 ZE.

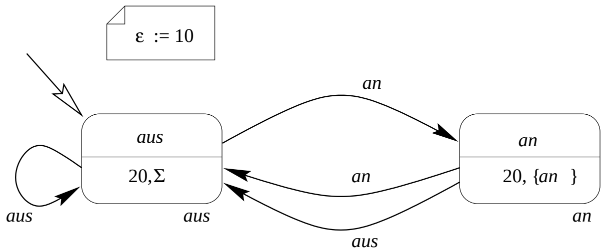

Abbildung 2.2: Beispiel eines einfachen SPS-Automaten

In [Sch97] wurde eine weitere Variante der SPS-Automaten vorgestellt, die Eingabe und Ausgabe über Variablen ausdrücken, und als SPS_V-Automaten bezeichnet werden. Eine knappe formale Definition dieser Automaten wird im Anhang A in Definition A.1 gegeben. Diese Automaten lassen sich nicht semantisch sinnvoll in die einfachen SPS-Automaten überführen, so daß eine Erweiterung um Hierarchie erarbeitet wurde. Die Überführung eines SPS_V-Automaten in solch einen sogenannten $\text{SPS}_{\mathcal{H}}$-Automaten geschieht durch eine Entfaltungsfunktion, wodurch den SPS_V-Automaten die Semantik des durch die Entfaltung gewonnenen $\text{SPS}_{\mathcal{H}}$-Automaten zugeordnet wird.

hierarchischer SPS-Automat

Definition 2.2: Ein Tupel $\mathcal{A} = (Q, S, \Sigma, \delta, q_0, \varepsilon, T, S_t, S_e, \Omega, \omega)$ ist ein *hierarchischer SPS-Automat*, wenn folgende Eigenschaften erfüllt sind:

- für $Q, \Sigma, \delta, q_0, \varepsilon, \Omega$ und ω gelten die Eigenschaften aus Definition 2.1
- S ist endliche Menge von *Superzuständen*, die auch als Hierarchieknoten bezeichnet werden.
- $T = (\{r\} \cup S \cup Q, E)$ ist ein (Hierarchie-) Baum mit dem Wurzelknoten $r \notin (S \cup Q)$, den Elementen von S als innere Knoten und mit den Elementen von Q als Blätter.
- S_t ist eine Funktion vom Typ $S \cup Q \to \mathbb{R}_{\geq 0}$, die jedem (Super-) Zustand eine Verzögerungszeit zuordnet.

- S_e ist eine Funktion vom Typ $S \cup Q \to \wp(\Sigma)$, die jedem (Super-) Zustand die Eingabesymbole zuordnet, die in der durch S_t festgelegten Zeitspanne ignoriert werden sollen.

und es gilt die Bedingung 2.1 □

Durch den hier eingeführten Hierarchiebaum wird es möglich, Zustände zusammenzufassen und diesen eine gemeinsame Zeitbeschriftung, also Verzögerungszeit und Menge zu ignorierender Elemente, zu geben, so daß diese Menge von Zuständen, ausgedrückt durch den Superzustand s, nur mit Eingaben außerhalb der Menge $S_e(s)$ vorzeitig verlassen werden kann. Um dieses Zusammenfassen auf einem vorhandenen SPS-Automaten zu realisieren wird ein neuer Hierarchieknoten in den Baum eingefügt, dem über S_t und S_e die gewünschte Zeitbeschriftung zugeordnet wird, und der als Kindsknoten die betrachtete Menge von Zuständen erhält. Auf die gleiche Weise können Mengen von Hierarchieknoten und Knoten zu weiteren Hierarchieebenen zusammengefaßt werden.

Das Konzept der Hierarchie kann analog auch für SPS$_V$-Automaten eingeführt werden, so daß auch hier die erweiterten Möglichkeiten der Zeitbehandlung zur Verfügung stehen. Diese in [Sch97] als hierarchische SPS$_V$-Automaten vorgestellten Automaten, deren Syntax ebenfalls im Anhang in Definition A.2 gegeben ist, lassen sich ebenso wie SPS$_V$-Automaten in SPS$_\mathcal{H}$-Automaten überführen. Diese Entfaltung entspricht im wesentlichen der Entfaltung von SPS$_V$-Automaten; es ist lediglich der Hierarchiebaum des hierarchischen SPS$_V$-Automaten und damit eine Erweiterung der Beschriftungsfunktionen S_t und S_e neu zu beachten. Weil es sich bei SPS$_V$-Automaten und hierarchischen SPS$_V$-Automaten um die Automaten handelt, die im Editorsystem Moby/plc eingegeben werden und schließlich auch dem Simulator als Modell dienen, wird für die semantische Betrachtung im Folgenden die in [Sch97] gegebene Entfaltung mit der Erweiterung zur Hierarchiebehandlung informell beschrieben und am Beispiel im nächsten Abschnitt noch verdeutlicht. Der hierbei entstehende hierarchische SPS-Automat soll im folgenden als *Entfaltungsautomat* bezeichnet werden.

Entfaltungs-automat

Die Eingabe bei SPS$_V$-Automaten wird durch die Belegung der (möglicherweise mehreren) Eingangsvariablen ausgedrückt, so daß das Eingabealphabet Σ des Entfaltungsautomaten durch das kartesische Produkt aller möglichen Eingangsbelegungen definiert ist. Die Belegung der lokalen Variablen und der Ausgangsvariablen muß durch die Zustände Q des Entfaltungsautomaten codiert werden, so daß sich die Zustandsmenge aus der Kombination der SPS$_V$-Automaten-Zustände Q_V mit allen Belegungen Θ_{lo}[4] dieser

[4]Der Index soll anzeigen, daß es sich um die Belegungen der lokalen Variablen („local") und der Ausgabevariablen („out") handelt.

Variablen ergibt, d.h. $Q = \{(q,\sigma)|q \in Q_V \wedge \sigma \in \Theta_{lo}\}$. Die Ausgabefunktion kann somit auch leicht festgelegt werden, denn das Ausgabealphabet Ω entsteht wiederum durch das Kreuzprodukt der möglichen Belegungen der Ausgangsvariablen, so daß die Ausgabefunktion ω den Zuständen die mit ihnen assoziierte Belegung der Ausgangsvariablen zuordnet.

Um die Zeitbeschriftungen korrekt abzubilden, müssen für alle ursprünglichen Zustände neue Hierarchieknoten eingeführt werden, denn durch die Verwendung mehrerer Zustände im Entfaltungsautomaten, die nur einen Zustand des SPS_V-Automaten mit verschiedenen Variablenbelegungen repräsentieren, muß die Zeitbeschriftung für die gesamte Menge dieser Zustände gelten. Dieses entspricht genau einer neuen Hierarchiestufe. So wird für jeden ursprünglichen Zustand $q_V \in Q_V$ ein neuer Hierarchieknoten h_{q_V} definiert, der die Zeitbeschriftung erhält, und dem als Kindsknoten die neuen Zustände $\{(q_V,\sigma)|\sigma \in \Theta_{lo}\}$ zugeordnet werden. Seien mit S_V die Hierarchieknoten des SPS_V-Automaten bezeichnet, dann gilt somit für die Hierarchieknoten S des Entfaltungsautomaten $S = S_V \cup \{h_{q_V}|q_V \in Q_V\}$. Die übrige Hierarchie bleibt erhalten, d.h. wenn es vorher eine Kante $(h, q_V) \in E_V$ des Hierarchiebaums gab, mit $h \in S_V$ und $q_V \in Q_V$, so wird diese jetzt ersetzt durch eine Kante (h, h_{q_V}) und es werden die Kanten zu den Kindsknoten hinzugefügt, alle anderen Hierarchiestufen und -kanten bleiben erhalten.

Für eine formale Beschreibung dieser Entfaltung sei hier nochmals auf [Sch97] verwiesen, wobei die dort definierte Entfaltungsfunktion in der beschriebenen Weise um die Hierarchiebehandlung zu ergänzen ist. Hier soll nur deutlich werden, daß eine vernünftige Abbildung der hierarchischen SPS_V-Automaten auf $\mathrm{SPS}_{\mathcal{H}}$-Automaten möglich ist und somit für erstere eine formale Semantik gegeben ist.

2.1.4 Beispiel eines SPS-Automaten

Im Editorsystem Moby/plc können die in [Sch97] als hierarchische SPS_V-Automaten bezeichneten Automaten graphisch eingegeben und editiert werden. Hierbei wurde die Beschreibung der Ein- und Ausgabealphabete in eine programmiersprachenähnliche Form gebracht, so daß die Alphabete über *typisierte Variablen* beschrieben werden, Eingangsvariablen demnach das Eingabealphabet und Ausgangsvariablen entsprechend das Ausgabealphabet beschreiben. Zusätzlich stehen lokale Variablen zur Verfügung, um die Anzahl der zu beschreibenden Zustände klein und übersichtlich halten zu können. Die Ausgabefunktion wird durch *Wertzuweisungen* an Ausgangsvariablen, die als Beschriftung an Transitionen stehen, abgebildet, die in ihrer Syntax sehr an Ausdrücke imperativer Programmiersprachen wie Pascal erinnern, und so dem Benutzer ein vertrautes Bild geben. Die gleichen Ausdrücke können benutzt werden, um Zuweisungen an lokale Variablen vorzunehmen.

typisierte Variablen

Wertzuweisungen

Kommunikation erfolgt bei SPS-Automaten implizit, indem die Ausgangsvariable eines Automaten mit einer typverträglichen Eingangsvariablen eines anderen SPS-Automaten verbunden wird. Der Zeitpunkt, zu dem die Kommunikation der beiden Automaten stattfindet, ist demzufolge der Zeitpunkt, zu dem der Eingang des zweiten Automaten gepollt wird. Alle Wertänderungen an der Ausgangsvariable, die zwischen dem Pollen erfolgen, können vom zweiten SPS-Automaten also nicht wahrgenommen werden.

Kommunikation

Ein Beispiel soll die Arbeitsweise der SPS-Automaten noch einmal verdeutlichen. Abbildung 2.3 zeigt einen SPS-Automaten, der die Steuereinheit für ein Blinklicht realisieren soll. Das Licht kann an oder ausgeschaltet werden, die (einzige) Eingabevariable ist deshalb „schalter" und ist vom Aufzählungstyp „{an, aus}". Den gleichen Typ erhält die Ausgangsvariable „licht", die die Ansteuerung der Glühbirne darstellt.

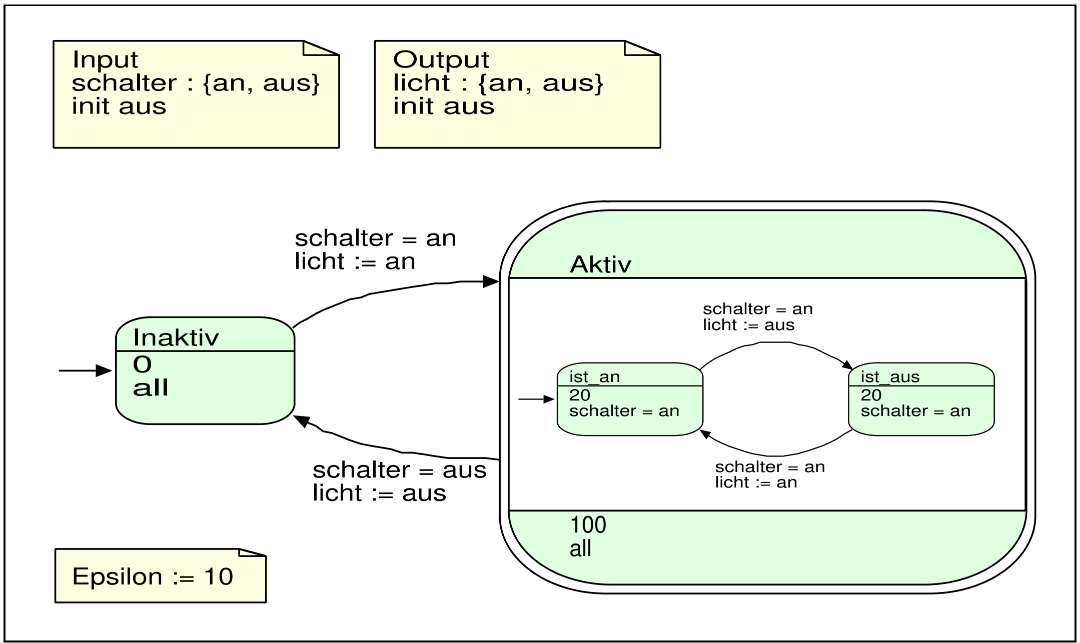

Abbildung 2.3: SPS-Automat für eine Blinklicht-Steuerung

Falls der Schalter für das Blinklicht angeschaltet wird, soll das Licht für mindestens 100 Zeiteinheiten (ZE) blinken. Der Automat wechselt, sobald er auf der Eingabevariable „an" liest, aus dem Zustand „Inaktiv" in den Zustand „ist_an" vom Subautomaten „Aktiv". Ein solcher Subautomat entspricht der Zusammenfassung von Zuständen durch einen Hierarchieknoten, so daß die Zeitbeschriftung eines Subautomaten als Beschriftung des entsprechenden Hierarchieknotens anzusehen ist. Dieser Subautomat kann aufgrund seiner Beschriftung „100, all" durch alle Eingangswerte erst nach 100 ZE verlassen werden. Die Beschriftung nach der Zeiteingabe ist ein boolescher Ausdruck, wobei als Synonym für den Ausdruck „*true*" die Schreibweise „all" eingeführt wurde, da *alle* Belegungen der Eingangsvariablen die Bedingung *true* erfüllen. Innerhalb dieses Subautomaten kann nach frühestens 20 ZE mit dem Eingangswert „an“ zum Pollzeitpunkt in den jeweils anderen Zustand gewechselt werden, wobei der Ausgang „an“ bzw. „aus“ gestellt wird. Der Subautomat kann mit dem Eingangswert „aus“ verlassen werden, wobei natürlich vorher der Timer dieses Subautomaten abgelaufen sein muß.

Wie bereits erwähnt kann ein hierarchischer SPS_V-Automat überführt werden in einen $\text{SPS}_{\mathcal{H}}$-Automat, so daß er auf diesem Wege eine (Duration Calculus) Semantik erhält. Für den hier beschriebenen SPS-Automaten „Steuerung“ würde der zugehörige $\text{SPS}_{\mathcal{H}}$-Automat

$$Steuerung = (Q, S, \Sigma, \delta, q_0, \varepsilon, T, S_t, S_e, \Omega, \omega)$$

wie folgt aussehen:

- $Q = \{(s, \sigma) | s \in \{Inaktiv, ist\_an, ist\_aus\}, \sigma \in \{an, aus\}\}$
- $S = \{h_{Inaktiv}, Aktiv, h_{ist\_an}, h_{ist\_aus}\}$
- $\Sigma = \{an, aus\}$
- δ ergibt sich aus folgender Tabelle:

q	a	$\delta(q,a)$	q	a	$\delta(q,a)$
$(Inaktiv, an)$	an	(ist_an, an)	$(Inaktiv, an)$	aus	$(Inaktiv, aus)$
$(Inaktiv, aus)$	an	(ist_an, an)	$(Inaktiv, aus)$	aus	$(Inaktiv, aus)$
(ist_an, an)	an	(ist_aus, aus)	(ist_an, an)	aus	$(Inaktiv, aus)$
(ist_an, aus)	an	(ist_aus, aus)	(ist_an, aus)	aus	$(Inaktiv, aus)$
(ist_aus, an)	an	(ist_an, an)	(ist_aus, an)	aus	$(Inaktiv, aus)$
(ist_aus, aus)	an	(ist_an, an)	(ist_aus, aus)	aus	$(Inaktiv, aus)$

- $q_0 = (Inaktiv, aus)$
- $\varepsilon = 10$
- $T = (\{r\} \cup S \cup Q, E)$ mit

$$\begin{aligned} E = \{ \; & (r, h_{Inaktiv}),\ (r, Aktiv), \\ & (h_{Inaktiv}, (Inaktiv, an)),\ (h_{Inaktiv}, (Inaktiv, aus)), \\ & (Aktiv, h_{ist\_an}),\ (Aktiv, h_{ist\_aus}), \\ & (h_{ist\_an}, (ist\_an, an)),\ (h_{ist\_an}, (ist\_an, aus)), \\ & (h_{ist\_aus}, (ist\_aus, an)),\ (h_{ist\_aus}, (ist\_aus, aus)) \\ \} & \end{aligned}$$

- Für alle $s \in (S \cup Q)$ gilt:

$$S_t(s) = \begin{cases} 100 & \text{falls} \quad s = Aktiv \\ 20 & \text{falls} \quad s \in \{h_{ist\_an}, h_{ist\_aus}\} \\ 0 & \text{sonst} \end{cases}$$

- Für alle $s \in (S \cup Q)$ gilt:

$$S_e(s) = \begin{cases} \{an\} & \text{falls} \quad s \in \{h_{ist\_an}, h_{ist\_aus}\} \\ \Sigma & \text{sonst} \end{cases}$$

- $\Omega = \{an, aus\}$
- Für alle $(s, \sigma) \in Q$ gilt $\omega((s, \sigma)) = \sigma$

Da diese Tupel-Schreibweise nicht sonderlich übersichtlich ist, insbesondere, wenn größere Automaten betrachtet werden, wurde auch eine graphische Notation entwickelt. Abbildung 2.4 zeigt den SPS$_{\mathcal{H}}$-Automat *Steuerung*, der als semantische Basis für den Beispielautomaten aus Abbildung 2.3 dient, auf diese Weise. Der Startzustand $(Inaktiv, aus)$ entspricht dem SPS$_V$-Automaten-Zustand $Inaktiv$ mit der Belegung aus für die Ausgangsvariable *licht* und wird in der graphischen Darstellung durch einen Pfeil (ohne Ursprungsknoten) gekennzeichnet. Wie diese Abbildung nun zeigt entstehen durch die einfache, direkte Entfaltung, die die Erweiterung der Entfaltungsfunktion aus [Sch97] auf die Behandlung der Hierarchie darstellt, unerreichbare Zustände. Eine Optimierung des Automaten würde also die Zustände $(Inaktiv, an)$, (ist_an, aus) und (ist_aus, an) streichen, so daß wieder ein SPS-Automat mit drei Zuständen entsteht.

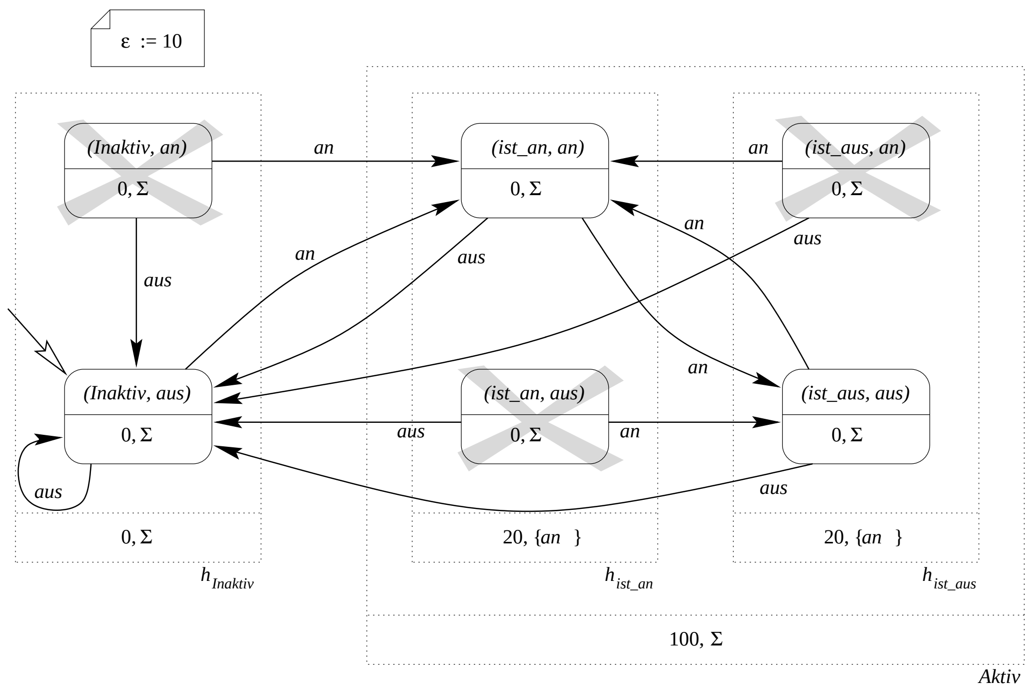

Abbildung 2.4: SPS$_{\mathcal{H}}$-Automat der Blinklicht-Steuerung

Durch diese Optimierung könnte auch der Hierarchiebaum T aus der Automatendefinition, den Abbildung 2.5 graphisch darstellt, vereinfacht werden. Die Superzustände $h_{Inaktiv}$, h_{ist_an} und h_{ist_aus} besitzen nach dem Streichen der unerreichbaren Zustände nur noch jeweils einen Kindsknoten, dessen Verzögerungszeit Null ist. Aus diesem Grund könnte diese Knoten wieder zusammengefaßt werden. Beide Optimierungen würden die Semantik des Automaten allerdings nicht vereinfachen, sie bleibt unverändert.

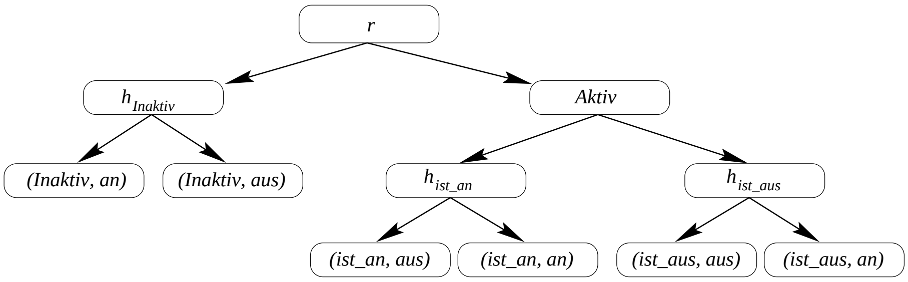

Abbildung 2.5: Hierarchiebaum des $\mathrm{SPS}_{\mathcal{H}}$-Automaten

2.2 Der interaktive SPS-Simulator

Dieser Abschnitt beschreibt die Arbeitsweise des bestehenden SPS-Simulators, wie er im wesentlichen in [Sch97] entstanden ist. Der Schwerpunkt liegt dabei auf der Betrachtung der Komponenten, die für die angestrebte Hintergrundsimulation erweitert oder verändert werden müssen.

Für die unterschiedlichen Anforderungen an Simulatoren sind verschiedene Konzepte entwickelt worden, so daß hier eine für das Problemfeld angemessene Auswahl getroffen werden muß. Die folgenden Abschnitte sollen darlegen, welche Methode aus welchen Gründen gewählt wurde.

2.2.1 Diskrete Ereignis-Simulation

Grundsätzlich können Simulationen in zwei Kategorien eingeteilt werden: die *kontinuierliche* Simulation, die den Modellzustand für jeden beliebigen Simulationszeitpunkt bestimmt, und die *diskrete* Simulation, bei der der Modellzustand nur für diskrete Zeitpunkte neu berechnet wird, für nichtberechnete Zeitpunkte bleibt der Modellzustand undefiniert oder wird interpoliert, i.a. durch Beibehalten des alten Zustands bis zum Zeitpunkt des neu berechneten. Für die Simulation größerer Systeme ist eine diskrete Simulation häufig besser geeignet, da sie den Rechenaufwand in vielen Fällen beschränkt. Dieses liegt darin begründet, daß, um eine kontinuierliche Simulation zu erreichen, Systemzustände häufig durch komplexe, vom Simulator zu bestimmende Funktionen beschrieben werden müssen, die z.B. durch das Lösen von Differentialgleichnungen ermittelt werden. Die diskrete Simulation beschreibt hingegen den Modellzustand durch Paare aus einem Zeitpunkt und einem (diskreten) Wert des Modellzustands.

diskrete Simulation

Die Klasse diskreter Simulationsmodelle kann wieder unterteilt werden in *zeit-diskrete* Modelle, die eine äquidistante Zeiteinteilung vornehmen, und *ereignis-diskrete* Modelle. Letztere machen die Zeiteinteilung abhängig vom Eintreffen von *Ereignissen* (engl. *events*) , so daß i.a. keine äquidistante Zeiteinteilung erfolgt. Ein solches Ereignis wird dadurch ausgemacht, daß der Modellzustand neu zu bestimmen ist.

Da das Zeitverhalten von SPS-Automaten angemessen durch das ereignisdiskrete Modell nachgebildet werden kann, und zudem eine interaktive Simulation hierdurch leichter realisierbar ist, wurde für die Simulation von SPS-Automaten in [Sch97] dieses Modell gewählt. Denn die Zustände von SPS-Automaten lassen sich diskret modellieren und Zustandswechsel finden zu exakt bestimmbaren Zeitpunkten statt, so daß die Ereignisse leicht zu identifizieren sind. [Mit82] stellt drei Möglichkeiten vor, ein solches Simulationsmodell zu implementieren. Bei der *ereignis-orientiert*en Methode wird jedes Ereignis für sich behandelt und im objektorientierten Entwurf als ein

ereignis-orientiert

Objekt modelliert. Jedem solchen Ereignis wird eine Funktion zugeordnet, die die für das Ereignis spezifischen Aktionen vornimmt. Das Auslösen dieser Funktion wird auch *Feuern* (eines Ereignisses) genannt.

Feuern

prozeß-orientiert

Beim *prozeß-orientierte*n Ansatz wird das Systemverhalten durch eine Menge von Prozessen repräsentiert. Solche Prozesse sind bestimmt durch eine Folge von Ereignissen mit zugehörigen Aktionen. Der Unterschied zum ereignis-orientierten Ansatz liegt darin, daß hier nicht einzelne Ereignisse sondern ganze Prozesse in eine Liste nach dem Zeitpunkt ihres nächsten Ereignisses sortiert eingetragen werden. Ein solches Simulationssystem sollte Konstrukte zum Erzeugen, Entfernen und Neusortieren von Prozessen bereitstellen. Es erlaubt so eine bessere Strukturierung der Simulation, benötigt aber auch einen höheren Verwaltungsaufwand um die Prozesse adäquat zu behandeln.

aktivitäts-orientiert

Das *aktivitäts-orientierte* Konzept beruht auf der Idee, die Aktionen der Simulation in Aktivitäten einzuteilen, die zu höchstens einem Ereignis zählen und die jeweils mit einer Bedingung verknüpft sind. Mit fortschreitender Simulationszeit werden jeweils die Aktivitäten ausgeführt, deren Bedingung erfüllt ist.

Die ereignis-orientierte Methode bietet unter den vorgestellten den Vorteil, daß mit ihr die Simulation am einfachsten zu implementieren ist, ohne daß zusätzliche Verwaltungsstrukturen zu entwerfen sind. Gerade auch ein objektorientierter Entwurf dieser Methode fällt leicht, so daß für den SPS-Simulator in [Sch97] dieser Ansatz gewählt wurde. Der nächsten Abschnitte sollen zeigen, wie diese Methode umgesetzt wurde.

2.2.2 Event-Scheduling

Scheduler

Eine zentrale Rolle bei der diskreten Ereignis-Simulation spielt der sogenannte *Scheduler*, der eine Liste von Ereignissen verwaltet und die notwendigen Operationen hierauf zur Verfügung stellt. Für jedes Ereignis wird der Simulationszeitpunkt festgelegt, zu dem es stattfinden soll, so daß der Scheduler alle Ereignisse chronologisch sortieren kann und für einen gegebenen Simulationszeitpunkt alle zu bearbeitenden Ereignisse, bzw. das zeitlich nächst folgende Ereignis kennt.

Wird der Scheduler veranlaßt, einen Simulationsschritt durchzuführen, so bestimmt er das nächste Ereignis. Die Aktionen, die mit diesem Ereignis verbunden sind, werden ausgeführt, indem das Ereignis gefeuert wird. Sollen Simulationsschritte nicht zurückgenommen werden können, kann der Scheduler anschließend das abgearbeitete Ereignis aus der Liste entfernen. Während einer Simulation kann es verschiedenartige Ereignisse geben, die in der objektorientierten Modellierung durch Ableiten von der Ereignis-Klasse realisiert werden, so daß häufig nur die Funktion, die die Aktionen beschreibt, neu implementiert werden muß. Der folgende Abschnitt beschreibt die Ereignisse der SPS-Automaten, wie sie im vorhandenen Simulator implementiert sind.

2.2.3 SPS-Automaten-Ereignisse

Bisher unterscheidet der SPS-Simulator zwei Arten von Ereignissen. In einem SPS-Zyklus werden zunächst die Eingangssignale gelesen, das sogenannte Pollen, das durch ein Eingabeereignis oder *Input-Event* modelliert wird. Jedes Input-Event ist für das Pollen einer Eingangsvariablen zuständig. Ist die Eingangsvariable über einen Kanal (*channel*) mit der Ausgangsvariablen eines anderen SPS-Automaten verbunden, so überträgt der Simulator bei einem Input-Event den momentanen Wert der Ausgangsvariablen in die Eingangsvariable.

Input-Event

Die Berechnung des SPS-Automaten und das Schreiben der Ausgänge erfolgt zusammen in einem Zustandsereignis oder *State-Event*, also durch ein State-Event pro SPS-Automat. Diese Zusammenfassung darf erfolgen, da die Simulation selbst nicht in Echtzeit erfolgt[5] und die Ergebnisse der Berechnung erst durch das Schreiben der Ausgänge sichtbar werden. Durch ein State-Event werden auch alle Timer der betroffenen SPS-Automaten-Zustände neu gesetzt, d.h. die Timer der Zustände werden um die vergangene Zeit herabgesetzt, und neue Ereignisse, Input-Events für alle Eingangsvariablen und ein neues State-Event für den Automaten, erzeugt.

State-Event

Durch die Änderungen am Simulator in der Diplomarbeit wird ein etwas anderes Konzept der Timer eingeführt. Der Zustand merkt sich nicht mehr den aktuellen Wert des Timers, der immer wieder heruntergezählt werden muß (genaugenommen sogar bei jedem Fortschreiten der Simulationszeit), sondern es wird festgehalten, wann der Timer ablaufen wird. Der aktuelle Wert des Timers ergibt sich dann aus der Differenz des „Ablaufzeitpunktes“ und der aktuellen Zeit, falls diese größer als Null ist. Beim Rücksetzen eines Timers wird im Zustand dementsprechend der Timerwert zum aktuellen Zeitpunkt addiert und als neuer „Ablaufzeitpunkt“ eingetragen. Durch diese Umstellung kann zu jedem Zeitpunkt der aktuelle Timerwert ermittelt werden, ohne daß bei jedem Fortschreiten der Simulationzeit eine Aktualisierung notwendig ist.

Die Nachbildung des SPS-Zyklus im Simulator erfolgt nun durch die Überführung in die zeitlich sortierten Input- und State-Events. Bei Initialisierung der Simulation wird für jeden Automaten ein State-Event in die Ereignisliste

SPS-Zyklus im Simulator

[5]Unter einer Simulation in Echtzeit wird hier verstanden, daß die Simulationszeit synchron zur real verstreichenden Zeit läuft. Der implementierte Simulator läßt hingegen die Simulationzeit erst dann (diskret) voranschreiten, wenn alle Ereignisse für den aktuellen Simulationszeitpunkt abgearbeitet sind.

eingetragen. Ein solches State-Event zum Zeitpunkt Null hat eine besondere Bedeutung, denn es stellt nur einen Zyklusbeginn, und nicht gleichzeitig das Ende des vorangegangenen Zyklus dar. Der Anfang eines Zyklus hat als Aktionen zur Folge, daß neue Ereignisse erzeugt werden müssen. Hierzu wird zunächst die Zykluszeit festgelegt, indem ein Aufruf des Zufallszahlengenerators erfolgt, der einen Zeitpunkt zwischen 1 und ε bestimmt, wodurch bereits der Zeitpunkt des folgenden State-Events festgelegt ist. Die Input-Events, die jeweils für eine Eingangsvariable erzeugt werden, werden analog auf das Intervall zwischen Zyklusbeginn und nächstem State-Event verteilt.

Nach dieser Initialisierung kann die Simulationszeit fortschreiten, so daß – für einen Automaten betrachtet – als nächstes ein Input-Event erreicht wird. Die Aktion bei diesem Ereignis ist das Pollen der Eingabe, es wird also der Wert, der an dieser Variable anliegt (i.a. der Wert einer anderen Ausgangsvariable, die mit der Eingangsvariable verbunden ist) in die Variable übernommen.

Beim weiteren Voranschreiten der Simulationszeit werden alle übrigen Input-Events auf die gleiche Weise behandelt[6], bis schließlich das State-Event erreicht wird. Dieses stellt zunächst das Ende eines SPS-Zyklus dar, was bedeutet, daß die berechnete Ausgabe auf die Ausgänge geschrieben werden muß. Diese Berechnung ist zwar noch nicht erfolgt, aber da die Simulationszeit während der Behandlung eines Ereignisses nicht voranschreitet, kann sie auch zu diesem Zeitpunkt erfolgen. Dieses geschieht, indem ermittelt wird, welche Transition des Automaten aktiviert ist. Hierbei wird auch überprüft, ob alle hierfür relevanten Timer in den Hierarchiestufen zwischen dem Ausgangszustand inklusive und dem gemeinsamen Hierarchieknoten mit dem Zielzustand exklusive abgelaufen sind.

Definition 2.3: Sei $\mathcal{A} = (Q, S, \Sigma, \delta, q_0, \varepsilon, T, S_t, S_e, \Omega, \omega)$ ein hierarchischer SPS-Automat mit $T = (N, E)$ und $N = \{r\} \cup S \cup Q$. Dann sei die Menge der Hierarchieknoten zu $s \in S \cup Q$ definiert durch

$$H_{\mathcal{A}} : S \cup Q \to \wp(N)$$

$$H_{\mathcal{A}}(s) = \{h_1, \ldots, h_k | h_1 = r \wedge h_k = s \wedge \forall i = 1, \ldots k-1 : (h_i, h_{i+1}) \in E\}$$

d.h. die Hierarchieknoten zu s sind alle Knoten auf dem Pfad von der Wurzel r zum (Super-) Zustand s.

Die zu betrachtenden Hierarchieknoten für einen Zustandswechsel von q mit der Eingabe a nach $\delta(q, a)$ sind damit definiert durch die Mengenfunktion

$$H^c_{\mathcal{A}} : Q \times \Sigma \to \wp(N)$$

[6]Bei genauer Betrachtung werden Ereignisse im Simulator nicht behandelt, sondern handeln, durch die Verwendung einer objektorientierten Implementierung, selbst.

$$H^c_{\mathcal{A}}(q,a) = \{h \in H_{\mathcal{A}}(q) \setminus H_{\mathcal{A}}(\delta(q,a)) \mid S_t(h) > 0 \land a \in S_e(h)\}$$

□

Die Hierarchieknoten eines Zustandes sind also gegeben durch alle Knoten auf dem Pfad von der Wurzel r des Hierarchiebaumes bis zum betrachteten Zustand. Die zu überprüfenden Timer sind demnach jene, die nur Hierarchieknoten des Ausgangszustandes sind. In der Simulation erfolgt die Überprüfung, ob die Eingabe in der Menge der Zeitbeschriftung liegt, wie dieses nach der Definition von $H^c_{\mathcal{A}}$ erforderlich ist, indem die Beschriftungen mit der aktuellen Belegung der Variablen ausgewertet werden.

Das „Schreiben aller Ausgänge“ erfolgt dann, indem die Aktion, die mit der Transition verknüpft ist, ausgeführt wird und der Zielzustand der Transition neuer aktueller Zustand des Automaten wird. Der neue SPS-Zyklus beginnt dann wieder mit der Erzeugung neuer Ereignisse.

2.2.4 Kompilativer Ansatz

Beim kompilativen Ansatz wird aus der Modellbeschreibung mittels eines Compilers ein vom Computer ausführbares Modell erzeugt. Auf diese Weise wird eine Interpretation nur bei der Erzeugung des Modells vorgenommen, so daß die Ausführungszeit während der Simulation drastisch gesenkt werden kann. Bei herkömmlichen kompilativen Ansätzen wird das gesamte ausführbare Modell bei einer Änderung der Modellbeschreibung neu erzeugt, so daß selbst kleinere Änderungen einen beträchtlichen Zeitaufwand zur Vorbereitung einer erneuten Simulation nach sich ziehen können. Der *interaktiv kompilative* Ansatz, der in [Tap97] beschrieben und vom vorhandenen Simulator genutzt wird, kompensiert den Nachteil dieser Methode, indem jeweils Modellfragmente als ausführbares Programm erzeugt werden und somit bei einer Änderung nur das entsprechende Fragment ersetzt werden muß.

interaktiv kompilativ

Die variablen Aspekte der SPS-Automaten werden durch Bedingungen und Aktionen an Zuständen und Transitionen ausgedrückt, die durch diesen kompilativen Ansatz in ausführbare Programmfragmente transformiert werden, so daß Überprüfung von Bedingungen und Ausführen von Aktionen direkt durch Funktionsaufrufe durchgeführt werden. Die notwendigen Erweiterungen für die Hintergrundsimulation betreffen jedoch die statischen Programmteile für die Simulation der SPS-Automaten, so daß dieses Konzept unverändert bleibt.

2.3 Semantik von SPS-Automaten

Die formale Betrachtung einer Spezifikation ist nur möglich, wenn die verwendete Spezifikationssprache eine wohldefinierte formale Semantik besitzt. Wie im Abschnitt 2.1 zu Beginn bereits erwähnt wurde, ist die Semantik eines SPS-Automaten ursprünglich durch Formeln des Duration Calculus definiert worden. Um die Semantik hier dementsprechend beschreiben zu können, wäre also eine Kurzeinführung in den Duration Calculus nötig, wodurch eine denotationelle Sicht gegeben würde. Allerdings ist eine operationelle Semantik häufig leichter nachvollziehbar und überschaubarer. Dieses ist ein Grund, weshalb hier die bereits erwähnte TA-Semantik dargestellt wird. Ein weiterer Grund, die Semantik der SPS-Automaten durch Timed Automata anzugeben, besteht darin, daß durch ihren operationellen Charakter eine Verbindung zur Simulator-Semantik leichter herzustellen ist, denn die Simulation erfolgt auch auf operationellem Wege.

2.3.1 Timed Automata Semantik

In diesem Abschnitt wird zunächst die in [DFMV98] vorgestellte operationelle Semantik für SPS-Automat beschrieben. Sie berücksichtigt allerdings keine hierarchischen SPS-Automaten, bezieht sich also auf SPS-Automaten nach Definition 2.1. Begonnen wird mit einer Beschreibung der Komponenten des Timed Automaton $\mathcal{T}(\mathcal{A})$, der einem SPS-Automaten $\mathcal{A}$ zugeordnet wird. Die Kombination dieser Komponenten führt dann zur Definition 2.4. Syntax und Semantik der Timed Automata sind in Anhang B zusammengefaßt.

Jede Stelle[7] von $\mathcal{T}(\mathcal{A})$ besteht aus vier Komponenten, ist also ein 4-Tupel (i, a, b, q) mit:

- $i \in \{0, 1, 2, 3\}$ beschreibt den Status der SPS („Programmzähler“), der den Phasen eines SPS-Zyklus entspricht. Die Bedeutung des Statuswertes ist gegeben durch
  - „0“: Anfang des Zyklus, zu dem die Eingabe noch nicht gepollt wurde.
  - „1“: Eingabe wurde gepollt, aber es wurde noch nicht entschieden, wie darauf reagiert werden soll.
  - „2“: Pollen und Testen sind erfolgt, mit der Entscheidung die Eingabe zu ignorieren.
  - „3“: Pollen und Testen sind erfolgt, mit der Entscheidung auf die Eingabe zu reagieren.

[7] Wie in [DFMV98] beziehen sich „Stellen“ auf den Timed Automaton und „Zustände“ auf den SPS-Automaten.

- $a \in \Sigma$ ist das aktuelle Eingabesymbol
- $b \in \Sigma$ ist das zuletzt gelesene (gepollte) Eingabesymbol
- $q \in Q$ ist der aktuelle Zustand des SPS-Automaten

Drei Uhren[8] werden für die Modellierung des Zeitverhaltens benötigt:

- x mißt die Zeit, die das aktuelle Eingabsymbol bereits vorliegt, und wird hier als *Eingabeuhr* bezeichnet. Eingabeuhr
- y mißt die Zeit, die bisher im aktuellen Zustand verweilt wurde, und soll im folgenden durch den Begriff *Verzögerungsuhr* beschrieben werden. Verzögerungsuhr
- z mißt die im aktuellen SPS-Zyklus verstrichene Zeit. Durch eine globale Invariante wird festgelegt, daß diese *Zyklusuhr* den Wert ε des SPS-Automaten nicht überschreiten darf. Indem diese Uhr nur beim Zyklusende zurückgesetzt wird, wird sichergestellt, daß ε die obere Schranke für die Zykluszeit ist. Zyklusuhr

(i, a, b, q)	$\xrightarrow{c,true,\{x\}}$	(i, c, b, q)	falls $c \neq a$	(ta-1)
$(0, a, b, q)$	$\xrightarrow{poll,0<x\wedge 0<z,\emptyset}$	$(1, a, a, q)$		(ta-2)
$(1, a, b, q)$	$\xrightarrow{test,y<S_t(q),\emptyset}$	$(2, a, b, q)$	falls $S_t(q) > 0 \wedge b \in S_e(q)$	(ta-3)
$(1, a, b, q)$	$\xrightarrow{test,y\geq S_t(q),\emptyset}$	$(3, a, b, q)$	falls $S_t(q) > 0 \wedge b \in S_e(q)$	(ta-4)
$(1, a, b, q)$	$\xrightarrow{test,true,\emptyset}$	$(3, a, b, q)$	falls $S_t(q) = 0 \vee b \notin S_e(q)$	(ta-5)
$(2, a, b, q)$	$\xrightarrow{tick,true,\{z\}}$	$(0, a, b, q)$		(ta-6)
$(3, a, b, q)$	$\xrightarrow{tick,true,\{z\}}$	$(0, a, b, q)$	falls $q = \delta(q, b)$	(ta-7)
$(3, a, b, q)$	$\xrightarrow{tick,true,\{y,z\}}$	$(0, a, b, \delta(q, b))$	falls $q \neq \delta(q, b)$	(ta-8)

Tabelle 2.1: Transitionsregeln der Timed Automata Semantik

Die Kanten des Timed Automaton $\mathcal{T}(\mathcal{A})$ sind in Tabelle 2.1 definiert. Eine Transition der Form $s \xrightarrow{e,c,T} s'$ ist dabei so zu lesen, daß ein Übergang von der Stelle s durch ein Ereignis e ausgelöst werden kann, wenn die Bedingung c für die Uhren erfüllt ist. Als Folge wird die neue Stelle s' angenommen, wobei alle Uhren in der Menge T wieder auf Null zurückgesetzt werden.

Im Timed Automata Modell existieren keine Variablen durch die die Eingabe eines SPS-Automaten, die kontinuierlich zur Verfügung steht, nachgebildet werden kann. Deshalb wird die Änderung der Eingabe zu $a \in \Sigma$ durch ein Ereignis a ausgedrückt. Diese Änderung wird festgehalten, indem der Timed

[8]Im Zusammenhang mit Timed Automata wird die Bezeichnung clocks, also Uhren, für Timer benutzt. Ein kleiner Unterschied besteht darin, daß bei Uhren die Zeit positiv gezählt wird, während Timer ablaufen, also von ihrer Startzeit den Wert auf Null herunterzählen.

Automaton in eine entsprechende Stelle wechselt, ausgedrückt durch Transition (ta-1). Das Pollen eines Eingabesymbols soll nur erlaubt werden, wenn die Eingabe eine gewisse Zeit (verschieden von Null) anliegt. Daher wird beim Übernehmen des aktuellen Wertes in Transition (ta-1) die Uhr x zurückgesetzt. Beim Pollen eines Input muß der Wert dieser Uhr größer als Null sein, so daß die gepollte Eingabe dann in der dritten Komponente der neuen Stelle festgehalten wird (vgl. Transition (ta-2)).

Anschließend erfolgt der Test, ob das aktuell gepollte Eingabesymbol ignoriert werden muß, was durch ein „test“-Ereignis ausgelöst wird. Gehört die Eingabe zu den zu ignorierenden Ereignissen ($b \in S_e(q)$), so erfolgt kein Zustandswechsel, falls die Verzögerungszeit des aktuellen Zustandes nicht abgelaufen ist ($y < S_t(b)$), was durch Transition (ta-3) ausgedrückt wird. Ist die Verzögerungszeit abgelaufen, muß ein Zustandswechsel nach der Transitionsfunktion δ des SPS-Automaten erfolgen, was durch den Status „3“ der Timed Automaton Stelle ausgedrückt wird (vgl. Transition (ta-4)). Ebenso muß ein möglicher Zustandswechsel vorgemerkt werden, wenn die Eingabe nicht ignoriert werden soll (ta-5).

Hat sich der Timed Automaton entschieden, die Eingabe zu ignorieren, so wird beim Zyklusende, das durch das Ereignis „tick“ ausgelöst wird, nur die Uhr z für die Zyklusdauer zurückgesetzt (Transition (ta-6)). Wurde das Eingabesymbol hingegen berücksichtigt, wird zusätzlich die Uhr y zurückgesetzt, falls ein neuer Zustand angenommen wurde (Transition (ta-8)). Wird kein neuer Zustand angenommen, so wird ebenfalls nur die Uhr y zurückgesetzt.

Definition 2.4: Sei $\mathcal{A} = (Q, \Sigma, \delta, q_0, \varepsilon, S_t, S_e, \Omega, \omega)$ ein SPS-Automat (nach Definition 2.1). Dann sei der Timed Automaton

$$\mathcal{T}(\mathcal{A}) = (\mathcal{S}, \mathcal{X}, \mathcal{L}, \mathcal{E}, \mathcal{I}, \mathcal{P}, \mu, S_0)$$

zu $\mathcal{A}$ definiert durch:

- $\mathcal{S} = \{0, 1, 2, 3\} \times \Sigma \times \Sigma \times Q$ ist die Menge der Stellen
- $\mathcal{X} = \{x, y, z\}$ ist die Menge der benutzten Uhren (clocks)
- $\mathcal{L} = \Sigma \cup \{poll, test, tick\}$ ist die Menge der Ereignisse
- $\mathcal{E}$ die Menge der Kanten gemäß Tabelle 2.1 mit $i \in \{0, 1, 2, 3\}$, $a, b, c \in \Sigma$ und $q \in Q$
- $\mathcal{I}(s) = z \leq \varepsilon$ als Invariante für jede Stelle $s \in \mathcal{S}$
- $\mathcal{P} = \Sigma \cup Q \cup \Omega$ als Menge von Ausgaben
- $\mu(i, a, b, q) = a \wedge q \wedge \omega(q)$ als Ausgabe für jede Stelle $(i, a, b, q) \in \mathcal{S}$
- $S_0 = \{(0, a, b, q_0) | a, b \in \Sigma\}$ als Menge initialer Stellen

□

Die formale operationelle Semantik eines $\mathrm{SPS}_{\mathcal{H}}$-Automaten ist durch diese Definition noch nicht gegeben. Der Simulator behandelt jedoch hierarchische SPS_V-Automaten, die ihre Semantik auf der Basis der $\mathrm{SPS}_{\mathcal{H}}$-Automaten erhalten, so daß ein Vergleich nur sinnvoll ist, wenn das Hierarchiekonzept bei der TA-Semantik berücksichtigt wird.

Hierzu müssen die Timer von Hierarchieknoten, wie sie durch Definition 2.3 auf Seite 20 bestimmt werden, Berücksichtigung finden. Demzufolge muß die Anzahl der Uhren, die für Timer verwendet werden, auf die Hierarchietiefe angepaßt werden. Diese Hierarchietiefe kann durch die maximale Anzahl der Elemente in der Menge von Hierarchieknoten bestimmt werden, wobei für die Wurzel kein Timer benötigt wird. So wird durch

$$l_{\mathcal{A}}(h) = |H_{\mathcal{A}}(h) \setminus \{r\}|$$

für alle $h \in S \cup Q$ die Hierarchietiefe des (Super-) Zustandes h, durch

$$l(\mathcal{A}) = \max\{l_{\mathcal{A}}(q) | q \in Q\}$$

die Hierarchietiefe des $\mathrm{SPS}_{\mathcal{H}}$-Automaten und damit die Anzahl von Verzögerungsuhren bestimmt. Für jede Hierarchiestufe i wird eine Verzögerungsuhr y_i benötigt, so daß die Zeitbedingung für einen Zustandsübergang aus dem Zustand $q \in Q$ über mehrerer Hierarchieebenen mit der Eingabe $a \in \Sigma$ ausgedrückt werden kann durch die Bedingung

$$C_{\mathcal{A}}(q,a) = \bigwedge_{h \in H^c_{\mathcal{A}}(q,a)} y_{l_{\mathcal{A}}(h)} \geq S_t(q).$$

Diese Bedingung besagt genau, daß alle relevanten Timer abgelaufen sind. Erfolgt nun solch ein Zustandsübergang, so müssen die Verzögerungsuhren der neu betretenen Hierarchieebenen zurückgesetzt werden. Für einen Zustandsübergang von $q \in Q$ mit der Eingabe $a \in \Sigma$ läßt sich diese Menge zurückzusetzender Uhren bestimmen durch

$$R_{\mathcal{A}}(q,a) = \{y_{l_{\mathcal{A}}(h)} | h \in H_{\mathcal{A}}(\delta(q,a)) \setminus H_{\mathcal{A}}(q)\}$$

Mit diesen abkürzenden Schreibweisen müssen die Transitionen des Timed Automaton $\mathcal{T}(\mathcal{A})$ zu einem hierarchischen SPS-Automaten $\mathcal{A}$ dann gemäß Tabelle 2.2 abgeändert werden.

Die Transition (ta-3') besagt, daß die Eingabe ignoriert wird, wenn mindestens ein relevanter Timer nicht abgelaufen ist, und dementsprechend drückt (ta-4') aus, daß alle relevanten Timer bereits abgelaufen sind. Gibt es keine Timer für den Zustandswechsel aus q mit der Eingabe b zu beachten, dann ist die Menge $H^c_{\mathcal{A}}(q,b)$ leer, und somit entspricht (ta-5') der Transition (ta-5) für SPS-Automaten ohne Hierarchie. Transition (ta-8') muß nun noch dafür sorgen, daß alle Timer der neu betretenen Hierarchiestufen zurückgesetzt werden.

$(1,a,b,q)$	$\xrightarrow{test,\neg C_{\mathcal{A}}(q,b),\emptyset}$	$(2,a,b,q)$	falls $H^c_{\mathcal{A}}(q,b) \neq \emptyset$	(ta-3')
$(1,a,b,q)$	$\xrightarrow{test,C_{\mathcal{A}}(q,b),\emptyset}$	$(3,a,b,q)$	falls $H^c_{\mathcal{A}}(q,b) \neq \emptyset$	(ta-4')
$(1,a,b,q)$	$\xrightarrow{test,true,\emptyset}$	$(3,a,b,q)$	falls $H^c_{\mathcal{A}}(q,b) = \emptyset$	(ta-5')
$(3,a,b,q)$	$\xrightarrow{tick,true,\{z\}\cup R_{\mathcal{A}}(q,b)}$	$(0,a,b,\delta(q,b))$	falls $q \neq \delta(q,b)$	(ta-8')

Tabelle 2.2: Transitionsregeln der TA-Semantik für $\mathrm{SPS}_{\mathcal{H}}$-Automaten

Definition 2.5: Sei $\mathcal{A} = (Q, \Sigma, \delta, q_0, \varepsilon, T, S_t, S_e, \Omega, \omega)$ ein hierarchischer SPS-Automat (nach Definition 2.2). Dann sei der Timed Automaton $\mathcal{T}(\mathcal{A}) = (\mathcal{S}, \mathcal{X}, \mathcal{L}, \mathcal{E}, \mathcal{I}, \mathcal{P}, \mu, S_0)$ zu $\mathcal{A}$ definiert durch:

- Für $\mathcal{S}, \mathcal{L}, \mathcal{I}, \mathcal{P}, \mu$ und S_0 gelten die Eigenschaften aus Definition B.2.
- $\mathcal{X} = \{x, y_1, \ldots y_{l(\mathcal{A})}, z\}$ die Uhren (clocks)
- $\mathcal{E}$ die Menge der Kanten gemäß (ta-1), (ta-2), (ta-6) und (ta-7) aus Tabelle 2.1 sowie (ta-3'), (ta-4'), (ta-5') und (ta-8') aus Tabelle 2.2 mit $i \in \{0, 1, 2, 3\}$, $a, b, c \in \Sigma$ und $q \in Q$

$\square$

2.3.2 Abbildung auf die implementierte Semantik

In diesem Abschnitt soll der Zusammenhang der formalen Semantik von SPS-Automaten und der durch die Implementierung des Simulators induzierten operationellen Semantik (Simulator-Semantik) gezeigt werden. Hierbei ist es allerdings kein Anliegen, einen formalen Beweis für die Verträglichkeit der Semantiken zu bringen, sondern lediglich deren Beziehung zu beschreiben. Ein formaler Beweis ist nämlich nicht möglich, da die Simulator-Semantik nicht formal gegeben ist. Dieses geschieht, indem die Transitionen eines Timed Automaton $\mathcal{T}(\mathcal{A})$ den Operationen zugeordnet werden, die der Simulator bei der Simulation des SPS-Automaten $\mathcal{A}$ durchführt.

Wie im Abschnitt 2.2.3 bereits erläutert wurde, wird im SPS-Simulator der Zyklus eines SPS-Automaten lediglich durch zwei Ereignisse nachgebildet, die mit den vier möglichen Werten des Status, die in der TA-Semantik Verwendung finden, in Einklang gebracht werden müssen. Die Lösung liegt darin, die Ereignisse beider Semantiken miteinander zu identifizieren, so daß die Ereignisse der Eingabe $a \in \Sigma$, sowie *poll*, *test* und *tick*, wie sie in der TA-Semantik existieren, auf die Ereignisse des Simulators abzubilden sind.

Die offensichtlichste Verbindung besteht hierbei zwischen dem *poll* Ereignis und dem Input-Event. Feuert der Simulator ein Input-Event, wird der aktuell treibende Wert[9] in die Eingangsvariable übernommen. Die TA-Semantik führt genau diese Aktion in ihrem (ta-2)-Transitionsschritt bei einem *poll* Ereignis durch. Für die Bedingung $0 < x$ der Eingabeuhr wird vereinbart, daß jede Belegung der Variablen für einen von Null verschiedenen Zeitraum anliegt. Diese Annahme darf auch für Variablen gemacht werden, die noch zum selben Simulationszeitpunkt ihren Wert ändern, weil die Simulationszeit nur diskret behandelt wird und somit für diesen Fall die Zeit bis zur Eingabeänderung kürzer ist als die kleinste durch die diskrete Simulation auflösbare Zeitspanne ist.

Die nächst einfachere Verbindung stellt die Assoziation von *tick* Ereignis und State-Event dar. Beim *tick* Ereignis endet der Zyklus des SPS-Automaten in der TA-Semantik, was ebenfalls durch das State-Event im Simulator vollzogen wird, indem der neue aktuelle Zustand für das State-Event des nächsten Zyklus eingetragen wird. Der Simulator hält nämlich in den State-Events fest, welcher Zustand des SPS-Automaten bis zum Eintreten des Ereignisses gültig ist. Ebenso wird die Invariante für die Zyklusuhr $z \leq \varepsilon$ durch die Verteilung neu erzeugter Ereignisse auf das ε-Intervall eingehalten und damit auch das Rücksetzen dieser Uhr nachgebildet. Auch werden bei einem „echten“ Zustandswechsel die betroffenen Timer auf ihren Startwert zurückgesetzt, was genau dem Rücksetzen der y_i in der Transition (ta-8’) entspricht. Allerdings gilt nicht die Umkehrung, daß alle Aktionen des State-Event denen des *tick* Ereignis entsprechen.

Die Aktionen des State-Event, die nicht mit dem *tick* Ereignis des Timed Automaton übereinzubringen sind, sind genau die, die vom *test* Ereignis abgedeckt werden. Die Statusänderung des *test* Ereignis ist die nach außen nicht sichtbare Berechnungsphase im SPS-Zyklus, in der die beobachtbaren Ausgangsvariablen noch nicht geschrieben werden dürfen. Die Prüfung, ob die gepollte Eingabe ignoriert werden muß, also der Test, ob die Verzögerungszeit des aktuellen Zustandes (und ggf. seiner Superzustände) abgelaufen ist, entspricht der Bedingung, daß die Verzögerungsuhr(en) die Zeitbeschriftung des Automatenzustandes bzw. seiner Superzustände überschritten hat. Durch ein Zusammenfassen der Ereignisse *test* und *tick*, was die Semantik des Timed Automaton zuläßt, indem beide Ereignisse immer zum selben Zeitpunkt stattfinden, ist dieses zusammengefaßte Ereignis gleichbedeutend mit dem State-Event des Simulators. Hierbei muß beachtet werden, daß der Simulator nicht mehr alle zeitlichen Abläufe des Timed Automaton nachbilden kann. Solange die Ereignisse *test* und *tick* der TA-Semantik nicht zusammengefaßt werden, ist das Ablaufen eines Timers vor dem Zyklusende möglich, aber

[9] Abschnitt 3.2 geht auf diesen Vorgang und die Begriffsbildung noch genauer ein, da die ursprünglichen Implementation an dieser Stelle nicht ausreichend war. Für die semantische Betrachtung kann zunächst vereinfacht angenommen werden, daß der treibende Wert immer der Wert einer Ausgangsvariable ist, die mit der zu pollenden Variable verbunden ist.

der SPS-Automat kann sich dennoch bereits entschieden haben, die Eingabe ignorieren zu müssen. Der zeitliche Verlauf eines solchen Zyklus wird in Abbildung 2.6 dargestellt, wobei für den aktuellen SPS-Automaten-Zustand q und die aktuelle Eingabe $a \in \Sigma$ als Voraussetzung $a \in S_e(q)$ und $S_t(q) > 0$ gilt. Hieraus folgt nun, daß die Simulator-Semantik lediglich in der TA-Semantik enthalten ist, was für eine Simulation jedoch ausreichend ist, da immer nur ein mögliches Verhalten nachgebildet werden soll. Schließlich kann eine Gleichheit der Semantiken schon deshalb nicht erreicht werden, weil die Behandlung der Zeit im Simulator immer diskret sein muß, die TA-Semantik jedoch auch ein kontinuierliches Zeitmodell benutzt.

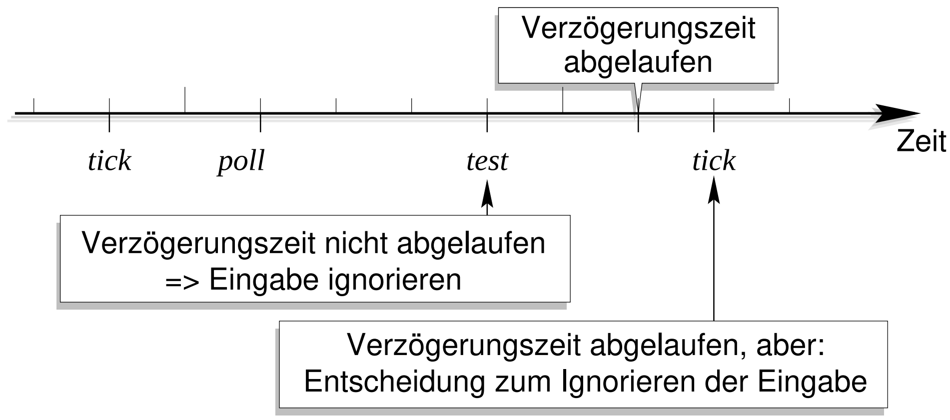

Abbildung 2.6: SPS-Zyklus im Timed Automaton

Bleiben als letzte Ereignisse noch die Eingaben zu betrachten. Bei der Simulation stehen die im Timed Automaton fehlenden Variablen, um die aktuelle Eingabe kontinuierlich festzuhalten, zur Verfügung. So kann auch hier mit einem Hinweis auf Abschnitt 3.2 vereinfacht angenommen werden, daß jede Eingabe für eine Eingangsvariable die aktuelle Belegung einer Ausgangsvariable ist. Ein Ereignis $a \in \Sigma$ des Timed Automaton, entspricht demnach der Änderung der Ausgangsvariable eines SPS-Automaten. Diese Änderung braucht jedoch nicht im betrachteten Automaten festgehalten werden, da dieses schon im Automaten der Ausgangsvariable geschieht. Mit der oben erwähnten Einschränkung zur Zeitauflösung im Simulator braucht auch das Rücksetzen der Eingabeuhr nicht modelliert zu werden. Umgekehrt muß die Eingabe durch ein Ereignis und das Rücksetzen der Eingabeuhr modelliert werden, gerade um die Belegung einer Variablen nachzubilden. Die Unabhängigkeit der Transitionsregel (ta-1) von Status, gepolltem Wert und Zustand des SPS-Automaten zeigt gerade, daß das modellierte Objekt, der treibende Wert der Eingangsvariable, unabhängig vom betrachteten SPS-Automaten Zustandsänderungen vornehmen kann.

Als Ergebnis dieser Betrachtung kann festgehalten werden, daß das operationelle Verhalten des Simulators eine Entsprechung im operationellen Verhalten des Timed Automaton für den simulierten SPS-Automaten hat. Aufgrund dieser Betrachtung kann der Simulator-Semantik eine angemessene Verträglichkeit[10] mit der TA-Semantik bescheinigt werden, und somit verhält sich die Simulation auch konform zur formalen DC-Semantik.

[10]Die Verträglichkeit kann nur angemessen, da die begrenzte Zeitauflösung im Simulator immer eine wenn auch zu vernachlässigende Diskrepanz darstellt.

2.4 Konzepte der MOBY-Klassenbibliothek

Die Implementierung in dieser Diplomarbeit baut auf die MOBY-Klassenbibliothek auf. Im Projekt MOBY wurde diese Klassenbibliothek zunächst in Smalltalk implementiert. In der Diplomarbeit von Andree Seidel [Sei93] wurde sie dann in der Programmiersprache C++ reimplementiert und in diversen Studien- und Diplomarbeiten erweitert. Auf der Basis der MOBY-Klassenbibliothek wurden so verschiedene Editorsysteme für graphenbasierte Projekte entwickelt, u.a. ein Petri-Netz Editor, ein Editor für SDL und ein Editorsystem für SPS-Automaten, das von der Projektgruppe IEKOΣ [BFvG⁺97] entwickelt wurde. Beschrieben ist die MOBY-Klassenbibliothek in [AAFT97], das bei genauerem Interesse herangezogen werden sollte.

Laufzeit-Klasseninformationen

Die Besonderheit an den Klassen der MOBY-Klassenbibliothek besteht darin, daß sie Klasseninformationen zur Laufzeit bereitstellen, d.h. ein Objekt kann mitteilen, zu welcher Klasse es gehört, bzw. ob eine andere Klasse eine Basisklasse dieses Objekts ist. Diese Funktionalität wird in der Basisklasse `MObject` bereitgestellt, so daß sie allen MOBY-Klassen, die alle hiervon abgeleitet sind, zur Verfügung steht. Durch die Verwendung einer Basisklasse für alle Objekte, können Funktionalitäten sehr flexibel gehalten werden. So stehen Container-Klassen in verschiedensten Ausprägungen zur Verfügung, die wiederum verschiedenartigste Objekte aufnehmen können. Beispiele hierfür sind alle von `Collection` abgeleiteten Klassen. Eine solche Kollektion ermöglicht das Sammeln von Objekten beliebigen Typs, das komfortable und auf Wunsch auch typsichere Durchlaufen dieser Kollektion.

Typsicherheit

Die *Typsicherheit* kann hier jedoch häufig erst zur Laufzeit erreicht werden. Durch das Erfragen der Klassenzugehörigkeit wird die Typsicherheit erreicht, dieses Erfragen jedoch findet zur Laufzeit des Programms statt, so daß Typfehler nicht bereits vom Compiler erkannt werden können.

graphische Benutzeroberfläche

Desweiteren werden Programmteile für die Programmierung einer graphischen Benutzeroberfläche weitgehend gekapselt, so daß Programme auf der Basis der MOBY-Klassenbibliothek sowohl mit dem kommerziellen Motif Widget-Set als auch mit dem freien Athena Widget-Set übersetzt und ausgeführt werden können. Zu diesem Zweck bietet die Klassenbibliothek zahlreiche einfache Bedienelemente an, wie Texteingabefelder, Knöpfe, Schalter und Beschriftungen, die durch Containerklassen angeordnet und zu Dialogen zusammengesetzt werden können. So können auf verhältnismäßig einfache Art komplexe Benutzungsoberflächen entstehen, wenn auch zur Zeit noch viel Quellcode hierbei entsteht.

Eine detailliertere Beschreibung soll an dieser Stelle jedoch nicht folgen, da eine auch nur annähernd vollständige Beschreibung den Umfang für diesen Grundlagen-Abschnitt unangemessen vergrößern würde. Das Kapitel zur Implementierung wird bei den betrachteten Klassen zur Realisation des SPS-

Simulators und des Hintergrundsimulators im gegebenen Fall Erläuterungen zu den Basisklassen geben.

Kapitel 3

Entwurf

Dieses Kapitel legt den Entwurf der Software dar, die in dieser Diplomarbeit entstanden ist. Zunächst wird ein grober Überblick über die entstandenen Komponenten und deren Zusammenspiel vermittelt, so daß eine Eindruck vom Gesamtsystem entsteht. Anschließend werden die Details der einzelnen Komponenten erläutert und auf das Zusammenspiel der Teilkomponenten noch etwas genauer eingegangen, um die Grundlage für die Implementierung, wie sie im nächsten Kapitel beschrieben ist, zu schaffen.

3.1 Systemaufbau

Die Entwicklung des Gesamtsystems erfolgt in drei Schritten. In der ersten Phase wird der vorhandene SPS-Simulator um die Schnittstelle zur Protokoll-Übersetzung erweitert. Dieses Schnittstelle soll das Protokollieren von Vorgängen der Ereignisliste erlauben, es soll also das Feuern von Ereignissen mitgeteilt werden. Zudem müssen interaktive Benutzereingaben festgehalten werden. Die andere Richtung der Schnittstelle soll Steuerung der Simulation ermöglichen, so daß das Erzeugen von Ereignissen durch die Protokoll-Übersetzung beeinflußbar wird. Sämtliche Kommunikation über diese Schnittstelle soll im folgenden als *low-level Protokoll* bezeichnet werden. low-level Protokoll

In der zweiten Phase wird zunächst die textuelle Repräsentation des low-level Protokolls und ein entsprechender Parser entwickelt. Die Protokoll-Übersetzung wird dann soweit entwickelt, daß die Simulation durch eine textuelle Protokolldatei eines aufgezeichneten Simulationslaufes gesteuert werden kann. Nachdem dieses geschehen ist, wird das textuelle Protokoll Schritt für Schritt um höhere Sprachkonstrukte mit entsprechenden Erweiterungen von Parser und Protokoll-Übersetzung zum high-level Eingabeprotokoll erweitert. Über Optionen soll es dem Benutzer ermöglicht werden zu bestimmen, welche Informationen für eine Simulation aus einem Eingabeprotokoll gelesen werden sollen und welche Informationen im Simulations-Protokoll festgehalten werden.

Zum Schluß der Entwicklung soll die Visualisierungskomponente entstehen. Diese Komponente muß ein aufgezeichnetes Simulationsprotokoll in seinem zeitlichen Verlauf graphisch darstellen, dabei dem Benutzer die Möglichkeit geben, nicht benötigte Informationen auszublenden und wieder sichtbar zu machen.

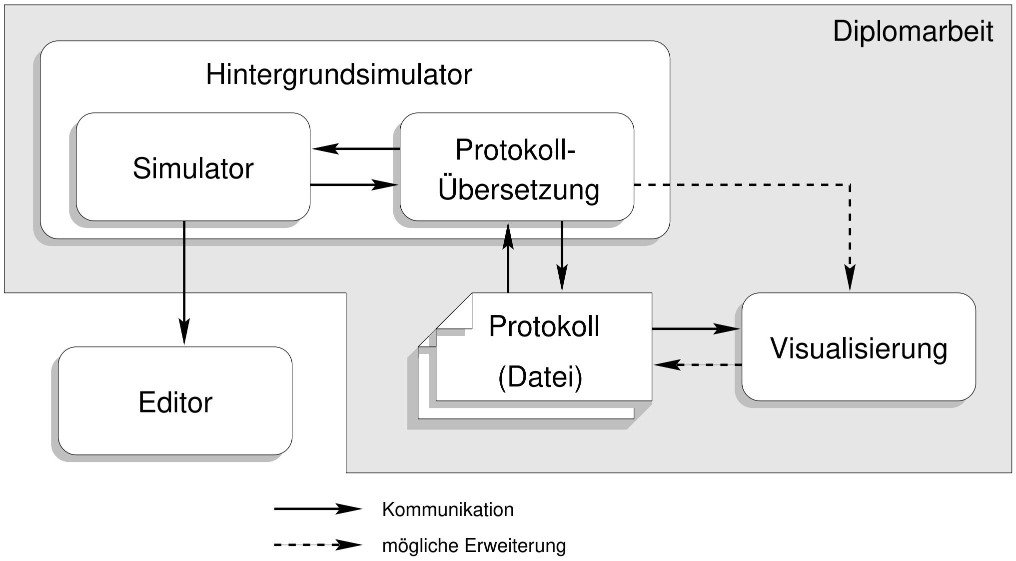

Abbildung 3.1: Systemaufbau

Abbildung 3.1 zeigt eine Übersicht über das so entworfene System. Der erweiterte Simulator bildet zusammen mit der Protokoll-Übersetzung den neuen Hintergrundsimulator. Die bidirektionale Kommunikation zwischen den beiden Hauptkomponenten bildet das low-level Protokoll. Die Kommunikation des Simulators mit dem Editor, die zur Visualisierung des für den aktuellen Simulationszeitpunkt gültigen Systemzustandes genutzt wird, bleibt unverändert. Die Protokoll-Übersetzung liest bzw. schreibt Protokolldateien, die auch durch die Visualisierungskomponente angezeigt werden können und so einen Überblick über den zeitlichen Verlauf der Simulation bieten.

3.2 Erweiterungen im Simulator

Der vorhandene Simulator für SPS-Automaten muß für die Aufgaben, die in dieser Diplomarbeit zu lösen sind, geändert und zum Teil umstrukturiert werden, da die Realisierung des low-level Protokolls als Schnittstelle erfolgt und somit der Simulator Dienste (in Form von Methoden) anbieten und in Anspruch nehmen muß. Bei diesen Arbeiten werden einige Erweiterungen zusammen mit der Umstrukturierung durchgeführt, die nötig oder aber nur wünschenswert sind.

3.2.1 Umstrukturierung des Klassengefüges

Grundsätzlich wird beim Entwurf des Hintergrundsimulators darauf geachtet, daß ursprüngliche und neue Komponenten klar voneinander zu unterscheiden sind. Insbesondere wird angestrebt, daß der interaktive Simulator weiterhin zur Verfügung steht und auch ohne die Komponenten zur Hintergrundsimulation lauffähig bleibt. Durch diese Anforderung und durch andere Erweiterungen, wie z.B. die Simulationszeit, sind jedoch bereits einige Änderungen am ursprünglichen Simulator notwendig.

Der Entwurf des gesamten Klassengefüges für den Simulator wird dahingehend geändert, daß die Simulatorklasse eine zentrale Rolle einnimmt. Da sie bereits für die Verwaltung der anderen Simulatorkomponenten, also deren Erzeugung und Zerstörung, verantwortlich ist, hat sie bereits Zugriff auf diese und so bietet es sich an, den Komponenten untereinander über diese Kernklasse Zugriffe zu erlauben. Durch diese Umstellung wird die Struktur des Simulators klarer und Erweiterungen sind so einfacher realisierbar. Ein Problem war es insbesondere festzustellen, welche Komponente der Simulation für welche Datenstruktur die Verantwortung hat. Dieses Problem trat z.B. für das zu simulierende Automatensystem auf. Wie in [Sch97] erläutert wurde, erhält der Simulator eine Datei mit der Modellbeschreibung, die in eine Datenstruktur, vergleichbar mit der des Editor-Systems[1], übersetzt wird. Dieses Datenstruktur wurde von der Kernklasse erzeugt und mehreren anderen Komponentenklassen als Parameter mitgegeben, so daß unklar war, welche Klasse für die Freigabe des durch die Datenstruktur belegten Speichers[2] zuständig ist. Gewichtiger noch ist der Umstand, daß zur Umstellung auf ein neues Modell alle Komponenten aktualisiert werden müssen. Durch die Umstellung werden solche Komponenten zentral im Simulator verwaltet und bei einem gewünschten Zugriff durch die Benutzung des entsprechenden Dienstes, also durch einen entsprechende Methodenaufruf, auf das aktuelle Objekt zugegriffen.

---

[1] In [BFvG+97] wird der Aufbau der Datenstruktur beschrieben, die im Simulator noch um simulations-spezifische Attribute erweitert wird.

[2] Bekanntlich sind aktuelle Produkte der Moby-Reihe mit der Programmiersprache C++ realisiert worden, so daß von Objekten dynamisch belegter Speicher wieder freigegeben werden muß. Im Gegensatz hierzu erfolgt die Speicherverwaltung z.B. in Java durch die Laufzeitumgebung, so daß bei der Programmierung keine Konstrukte zur Freigabe nötig sind.

3.2.2 Korrektes Verhalten globaler Inputs

Eine Hauptänderung des Simulators war das genauere Nachbilden der Abläufe an den Eingangsvariablen. Die Änderung der Ausgangsvariable eines Automaten hat noch nicht unmittelbar die Änderung einer hiermit verbundenen Eingangsvariablen zur Folge, sondern erst zu deren Pollzeitpunkt, was durch ein Input-Event modelliert wurde. Wenn jedoch eine Eingangsvariable nicht verbunden war, also einen *globalen Input* für das Automatensystem darstellt, so wurde in der Implementierung aus [Sch97] eine Änderung des Eingangswertes direkt in der Variablen sichtbar und das unter Umständen mehrmals während eines Zyklus. So konnte es unter Umständen sogar vorkommen, daß die Belegung der Variablen bei der Auswertung einer Kantenbedingung eine andere gewesen war, als bei den anschließenden Zuweisungsaktionen. Um in diesem Bereich eine genauere Modellierung zu erhalten, so daß die Simulation auch hier der Semantik entspricht, wird das Konzept der *Treiber* eingeführt. Jede Eingangsvariable hat demnach einen Treiber, der entweder die Ausgangsvariable eines anderen Automaten oder aber ein globaler Eingang ist. Diesen Treiber kann man sich als Wertänderungen auf der Leitung vorstellen, mit der die Eingangsvariable verbunden ist, die sie jedoch nur einmal pro Zyklus auswertet.

globaler Input

Treiber

Um den Treiber für einen globalen Input zu materialisieren, werden neue Objekte im Simulator erzeugt, die in einer neuen Komponente verwaltet werden. Diese neue Komponente ist nötig, da die Wertänderungen der „Leitung“ erfaßt werden müssen, ohne den globalen Input selbst zu ändern. Für Eingänge, die nicht globale Inputs sind, existieren diese Objekte bereits, nämlich als Ausgangsvariablen mit denen sie verbunden sind. Die interaktiven Änderungen der Eingänge wirken sich nun nur auf diese neuen Objekte unmittelbar aus und nicht auf die Eingangsvariablen. Letztere werten die Treiber nur genau zu ihrem Pollzeitpunkt aus und übernehmen dann deren Wert.

Abbildung 3.2 macht die Umstellung auf das konsequente Pollen der Eingänge noch einmal graphisch deutlich. Die zweite Eingangsvariable `in2` des Automaten `A2` ist mit dem Ausgang `out` des Automaten `A1` verbunden. Diese Variable ändert ihren Wert sowohl nach der neuen wie nach der alten Methode immer nur zum Pollzeitpunkt, der für beide Variablen synchron durch die gestrichelte Linie angezeigt wird – hier wurde keine Neuerung eingeführt. Die erste Eingangsvariable `in1` ist jedoch mit keiner Ausgangsvariablen verbunden und stellt somit einen globalen Input dar. Änderung an diesem Eingang haben bisher direkten Einfluß auf den Wert der Variable genommen (alt), so daß die Variable auch zwischen zwei Pollzeitpunkten ihren Wert ändern konnte. Durch die Einführung neuer (Treiber-)Objekte für die globalen Inputs ist diese Inkonsistenz abgestellt (neu).

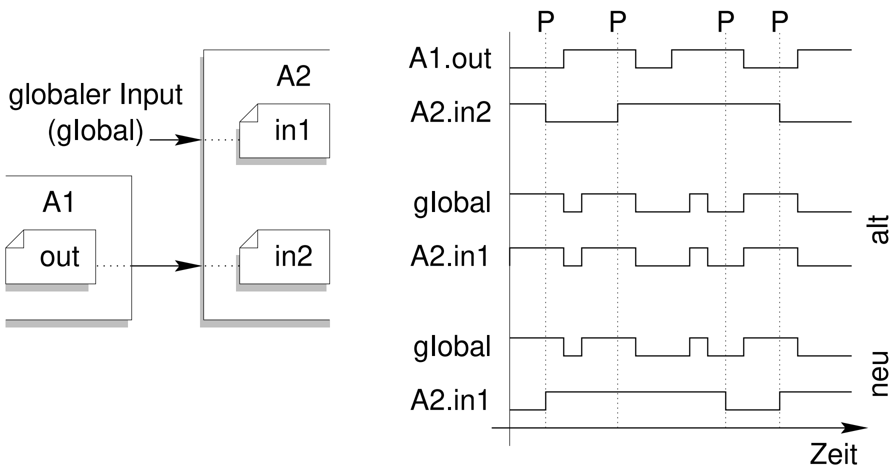

Abbildung 3.2: Behandlung globaler Inputs

3.2.3 Simulationsoptionen

Eine letzte Änderung betrifft die Einführung von *Simulationsoptionen*. Eine neue Komponente zeichnet sich für deren Verwaltung aus. Da auch sie als Instanz des Simulators verwaltet wird, haben auch hierauf alle Komponenten Zugriff. Diese Optionen bieten (bereits für die interaktive Simulation) die Möglichkeit, die Zeitauflösung des Simulators festzulegen. Da der Simulator diskrete Werte zur Modellierung der Zeit benutzt[3], steht nur eine begrenzte Auflösung der Zeit zur Verfügung. Diese Auflösung war bisher durch die Beschreibung im Editor festgelegt, so daß z.B. bei einer Zyklusdauer von zehn Zeiteinheiten auch nur zehn diskrete Zeitpunkte für das Zyklusende möglich sind. Je nach gewähltem Zyklusende stehen entsprechend weniger Zeitpunkte für die Input-Events zur Verfügung. Um die Anzahl dieser möglichen Zeitpunkte für Ereignisse zu erhöhen, kann durch die *Zeitauflösung* in den Optionen bestimmt werden, wie das Verhältnis von Simulations- und Editorzeit ist, so daß eine vorgegebene Anzahl von Zeiteinheiten im Editor durch eine beliebige andere Anzahl von Zeiteinheiten im Simulator modelliert wird.

Simulationsoptionen

Zeitauflösung

Die zweite Option, die bereits für den bisherigen Simulator zur Verfügung steht, ist die Wahl zwischen simultanem und verteiltem Pollen der Eingänge eines Automaten. So wird es ermöglicht zu bestimmen, ob die für die Eingangsvariablen eines Automaten erzeugten Input-Events alle zum selben Zeitpunkt stattfinden sollen, oder ob diese über das Zeitintervall des aktuellen Zyklus unabhängig voneinder verteilt werden. Diese Option ist sinnvoll, da in der formalen Betrachtung der SPS-Automaten nur ein Eingang betrachtet wird. Demnach ist der Eingang eines SPS-Automaten im Editorsystem ein Vektor mit den definierten Eingangsvariablen als Komponenten, dessen

simultanes Pollen

[3]Auf einem digitalen Computer stehen nur diskrete Werte zur Verfügung, so daß eine Zeitmodellierung hier immer nur diskret sein kann.

Wertebereich sich als kartesisches Produkt aus den Wertebereichen (Typen) der Eingangsvariablen ergibt. Bei der Betrachtung der Eingangssignale als eine Eingabe macht es somit keinen Sinn Teilkomponenten zu verschieden Zeitpunkten zu pollen. Da ein verteiltes Pollen dem Verhalten der SPSen, die die Entwicklung der SPS-Automaten ja angeregt haben, eher entspricht, wird auch diese Möglichkeit weiterhin beibehalten.

3.3 Das low-level Protokoll

Im low-level Protokoll müssen alle Simulationsereignisse festgehalten werden, die nicht aus den vorherigen Ereignissen bestimmbar sind. Da sowohl die Input-Events als auch die State-Events zufällig über das Epsilon-Intervall des Automaten verteilt werden, müssen demzufolge beide Arten festgehalten werden. Zusätzlich müssen noch Änderungen der „freien“ Eingangsvariablen protokolliert werden, d.h. Änderungen an Eingangsvariablen, die nicht mit einer Ausgangsvariablen eines SPS-Automaten verbunden sind und deshalb entweder interaktiv vom Benutzer oder automatisch von der Protokoll-Übersetzung geändert werden. Bei dieser Unterscheidung gilt es demnach Ereignisse zu berücksichtigen, die im Scheduler eingetragen waren und solche, für die es keinen Eintrag im Scheduler gibt.

Feuern in drei Phasen: Vorbereitung, Ausführung und Mitteilung

Das Vorgehen, um die erste Art von Ereignissen festzuhalten, kann allgemein für alle ereignisdiskreten Simulationen gelöst werden. Durch die objektorientierte Implementierung ist jedes Ereignis mit einer Methode ausgestattet, die für das Feuern des selbigen zuständig ist. Diese Methode kann benutzt werden, um vor dem Feuern eines Ereignisses dieses zu manipulieren, also zu prüfen, ob es stattfinden soll, ob es geändert werden soll oder wie es stattfinden soll. Die bestehende Implementierung wird zu diesem Zweck verändert, indem der Dienst, der das Feuern des Ereignisses realisiert, in drei Phasen unterteilt wird, die in der bisherigen Implementierung ineinander verwoben waren. Die erste Phase bereitet das Feuern des Ereignisses vor, indem alle zu treffenden Entscheidungen ermittelt werden. Hierunter fällt das Bestimmen des treibenden Wertes einer Eingangsvariablen oder der auszuführenden Transition, sowie das Auslosen der Zeitpunkte für das nächste Lesen der Eingänge und das Zyklusende, die ja zufällig über das Zeitintervall verteilt werden. Diese Phase wird im folgenden als *Vorbereitungsphase* bezeichnet. In der folgenden Phase wird die eigentliche Aktion, bzw. die Aktionen ausgeführt, so daß sie entsprechend als *Ausführungsphase* bezeichnet wird. In der letzten Phase wird das Stattfinden des Ereignisses „berichtet“. Hier werden also alle Komponenten vom Feuern dieses Ereignisses, bzw. von dessen Auswirkungen, in Kenntnis gesetzt, so daß der Begriff *Mitteilungsphase* hierfür verwendet werden soll.

Die Ereignisse, die auf die beschriebene Weise abgeändert werden, bleiben jedoch in ihrer Funktionalität unverändert. Der objektorientierte Entwurf ermöglicht es, die neue Funktionalität durch die Realisierung abgeleiteter Klassen bereitzustellen. Die Grundfunktionalität kann auf diese Weise von den bestehenden Klassen geerbt werden. Auf diese Weise wird bei Input-Events weiterhin der treibende Wert nur anhand des zu simulierenden Automatensystems bestimmt und nur dem Editor das Lesen dieses Wertes mitgeteilt. Ebenso bestimmt ein State-Event weiterhin nur anhand des Automaten, ob eine aktivierte Transition vorliegt und verteilt die Folgeereignisse zufällig über das Epsilon-Intervall. Auch hier wird wieder nur dem Editor die neue Belegung von Variablen und der neue Zustand mitgeteilt.

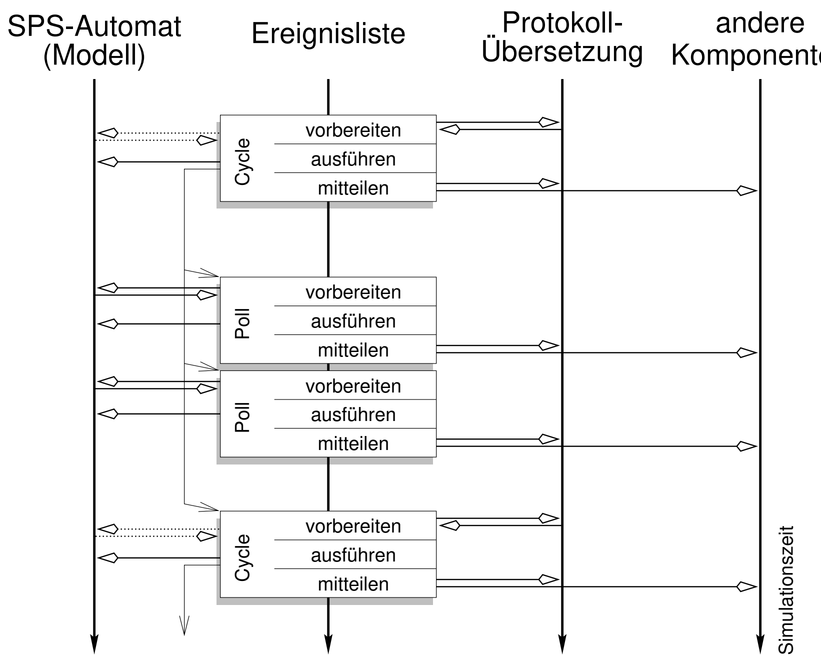

Abbildung 3.3: Phasen beim Feuern von Ereignissen

In den abgeleiteten Klassen wird die Möglichkeit, die Simulation über eine Eingabedatei zu steuern und sie aufzuzeichnen, realisiert. Die neue Klasse *Poll-Event* ersetzt in diesem Fall ein Input-Event. Diese Klasse fügt nur eine Benachrichtigung an die Protokoll-Übersetzung der Mitteilungsphase hinzu, so daß diese Pollzeitpunkte aufgezeichnet werden können. Die Klasse, die das State-Event ersetzt, das *Cycle-Event*, wurde etwas stärker verändert. In der Vorbereitungsphase wird zunächst die Protokoll-Übersetzung herangezogen, um zu ermitteln, wie weit die zu treffenden Entscheidungen durch die Eingabe- oder Steuerungsdatei bereits vorbestimmt sind. Es wird deshalb gefragt, ob die Folgeereignisse, also Zeitpunkte der neuen Poll-Events und des neuen Cycle-Events für den folgenden Zyklus, festgelegt sind und je nach Ergebnis angefordert bzw. auf herkömmliche Weise durch die geerbte Methode erzeugt werden. Zur Vorbereitung auf die Behandlung *nichtdeterministischer*

Poll-Event

Cycle-Event

nichtdeterministische SPS-Automaten

SPS-Automaten wird ebenso mit der zu verwendenden Transition verfahren. Ein solcher SPS-Automat würde es erlauben, daß mehrere Transitionen aktiviert sind und deshalb eine dieser aktivierten Transitionen nichtdeterministisch gewählt werden kann. Bei der Wiederholung einer Simulation müßte dementsprechend die im ersten Lauf gewählte Transition dem Protokoll wieder entnommen werden. In der Mitteilungsphase des Cycle-Events wird die Protokoll-Übersetzung über die geänderten Variablen und das Zyklusende informiert. Auch hier wird die geerbte Methode anschließend aufgerufen, so daß weiterhin der Editor informiert wird. Abbildung 3.3 zeigt diese Unterteilung der Ereignisse in die drei Phasen des Feuerns bei der Abarbeitung eines SPS-Zyklus.

Die zweite Art der zu berücksichtigenden Ereignisse, also solche, für die kein Eintrag in der Ereignisliste, dem Scheduler, existiert werden nicht so direkt behandelt. Hier wird die zentrale Bedeutung der Simulatorklasse wieder deutlich, die eine Methode für Aktionen zur Verfügung stellt, die der Mitteilungsphase beim Feuern von Ereignissen entspricht. Diese Mitteilungsfunktion wird von den Komponenten aufgerufen, die Treiber der Eingangsvariablen geändert haben, und führt in der Klasse des interaktiven Simulators keine Aktionen aus. Ein entsprechende abgeleitete Klasse für den Hintergrundsimulator benachrichtigt hingegen die Protokoll-Übersetzung, so daß sie auch solche Ereignisse aufzeichnen kann.

3.4 Die Protokoll-Übersetzung

Die zentrale Klasse für die Hintergrundsimulation stellt die Protokoll-Übersetzung dar. Alle Simulationsereignisse werden dieser Klasse mitgeteilt, so daß diese gegebenenfalls in der Ausgabedatei festgehalten werden können. Ebenso verwaltet sie das Eingabeprotokoll in einer aufbereiteten Form, so daß die Simulation über diese Eingabedatei gesteuert werden kann. Dem entsprechend lassen sich Methoden und Objekte der Protokoll-Übersetzung in zwei Kategorien einteilen, und zwar in solche, die für die Ausgabe zuständig sind und solche, die die Simulationssteuerung vornehmen und in diesem Sinne die Eingabe des Simulators realisieren.

Alle Ereignisse werden bei der Simulation nach dem Simulationszeitpunkt, zu dem sie feuern sollen, sortiert und in dieser Reihefolge abgearbeitet. Dadurch, daß jedes Ereignis sein Feuern der Protokoll-Übersetzung zu genau dem mit dem Ereignis verknüpften Zeitpunkt mitteilt, fallen bei der Simulation chronologisch geordnete Informationen an, die deshalb am sinnvollsten in dieser Reihenfolge auch festgehalten werden. Auch Änderungen der globalen Inputs durch Benutzerinteraktion fallen während der Simulation in chronologischer Reihenfolge an, da sich jeder Eingriff stets auf den aktuellen Simulationszeitpunkt bezieht. Die so entstehende Folge von Informationen soll im folgenden als *Trace*[4] bezeichnet werden, jede einzelne dementsprechend als (Trace-)Eintrag. Trace

3.4.1 Aufzeichnen einer Simulation

Für die Aufzeichnung eines Simulationslaufes soll festgelegt werden können, welche Ereignisse für den Benutzer von Interesse sind, um so die Anzahl aufzuzeichnender Ereignisse reduzieren zu können. Hierfür werden zu jedem Automaten Informationsobjekte benutzt, die Auskunft darüber geben, was im Ausgabeprotokoll enthalten sein soll. So unterscheidet ein Informationsobjekt für jede Variable des ihm zugeordneten Automaten, ob Wertänderungen und ob Änderungen des Treiberwertes festgehalten werden sollen. Desweiteren legt es fest, ob das zeitliche Verhalten in Form von Informationen darüber, wann Eingänge gelesen wurden und wann ein Zyklus endet, aufgezeichnet werden soll. Auf diese Weise müssen nur solche Ereignisse, deren Aufzeichnung gewünscht ist, in der Ausgabe repräsentiert werden. Diese Informationsobjekte werden in den *Trace-Optionen* zusammen mit weiteren Optionen, die im folgenden erläutert werden, zusammengefaßt. Trace-Optionen

Dadurch, daß Informationen in der Trace in chronologischer Reihenfolge festgehalten werden, häufig aber Ereignisse zum selben Simulationszeitpunkt stattfinden, ist es sinnvoll die Simulationsereignisse aufzuteilen in Zeitinformationen und Ereignisinformationen. Die Einträge der Trace teilen sich demzufolge in zwei Kategorien auf, und zwar in Einträge, die das Fortschreiten der Simulationszeit, und solche, die alle anderen Aktionen repräsentieren.

Um bei der Wiederholung einer Simulation die Trace in Form einer geeigneten Datenstruktur[5] vorliegen zu haben, werden für jede Art aufzuzeichnender Ereignisse Objekte erzeugt, so daß zu jeder Art von Eintrag in der Trace Klassen entstehen, die entsprechend ein Zyklusende, das Pollen eines Inputs, das Ändern einer Variablen oder das Ändern eines Treibers repräsentieren. Jede dieser Klassen kann durch ihre individuelle Methode bestimmen, wie sie in der Ausgabedatei textuell repräsentiert wird.

Jeweils durch die Mitteilungsphase der Ereignisse wird die Protokoll-Übersetzung über das Feuern eines Ereignisse informiert und entscheidet erst dann, ob das Ereignis in der Trace des Ausgabeprotokolls festgehalten werden soll. Die Verantwortung für diese Benachrichtigung liegt bei den Ereignissen. Für festzuhaltende Ereignisse wird dann ggf. ein Zeiteintrag und der eigentliche ereignisspezifische Eintrag generiert und an das Ende der bisherigen Trace

[4]Der Begriff der Trace (engl.: Spur, Einzelschrittfolge) ist auch im deutschsprachigen Raum ein üblicher Ausdruck, der nur schlecht durch eine deutsche Übersetzung umschrieben wird, so daß hier weiterhin der englische Begriff verwendet wird.

[5]Die Betonung liegt hierbei auf Struktur, denn die strukturierte Repräsentation vereinfacht die Interpretation der Trace

angefügt. Dadurch, daß Ereignisse erst beim Erreichen ihres Simulationszeitpunktes in der Trace festgehalten werden und die Simulation erst fortschreitet, wenn kein Ereignis für den aktuellen Simulationszeitpunkt vorliegt, sind alle Ereignisse aller Automaten eines Systems chronologisch geordnet. Hieraus ergibt sich, daß eine aufgezeichnete Trace durch neue Ereignisse nur erweitert, nicht aber verändert wird. Somit ist es möglich, einen solchen Traceanfang jederzeit, auch vor Abschluß der Simulation, bereits in eine Datei zu schreiben, die später ergänzt wird. Der gesicherte Traceanfang kann dann aus dem Hauptspeicher entfernt werden.

3.4.2 Reproduktion einer Simulation

Die zweite Kategorie von Objekten und Methoden der Protokoll-Übersetzung dient der Steuerung der Simulation. Hierzu werden Methoden angeboten mit denen ein Cycle-Event erfragen kann, ob durch das Eingabeprotokoll zu schaltende Transitionen und Folgeereignisse festgelegt sind, und über weitere Methoden können diese angefordert werden. Für das Ändern eines Treibers bei Wiederholung einer Simulation wird im Hintergrundsimulator eine neue Klasse von Ereignissen eingeführt, das *Driver-Event*. Dieses Ereignis entspricht in seiner Wirkung dem interaktiven Eingreifen des Benutzers. Auch hier wird das Feuern des Ereignisses wieder in die drei Phasen unterteilt. Bei der Vorbereitung wird ermittelt, welcher Wert für den Treiber zu setzen ist und welches Objekt als Treiber fungiert, denn es wird lediglich die zu treibende Variable als Referenz festgehalten. So können bei einer wiederholten Simulation eines Teils der Automaten auch solche Eingangsvariablen mit Werten versorgt werden, deren treibende Ausgangsvariablen zu Automaten gehören, die nicht mehr simuliert werden. In der Ausführungsphase eines Driver-Events muß der Wert des Treibers dann gesetzt werden. In der Mitteilungsphase wird dann dieselbe Methode des Simulators benutzt, die auch bei der Benutzerinteraktion aufgerufen wird, um der Protokoll-Übersetzung das Feuern des Ereignisses mitzuteilen. Außerdem müssen Fenster, die diesen Wert darstellen, aktualisiert werden.

Driver-Event

3.5 Simulations-Protokoll und high-level Eingabeprotokoll

Wie schon im vorangegangenem Abschnitt angedeutet, besteht ein Protokoll im wesentlichen aus der Trace, in der die Simulationsereignisse aufgezeichnet werden. Diese Trace besteht aus Objekten, die chronologisch geordnet sind und so einen Simulationslauf oder nur die Belegung von Eingangssignalen beschreiben. Dabei werden Einträge, die das Fortschreiten der Simulationszeit ausdrücken, und solche, die Simulationsereignisse repräsentieren, streng voneinander getrennt. So wird die sequentielle Anordnung der Traceeinträge für deren Interpretation entscheidend. Alle Simulationsereignisse sind dem Zeitpunkt zuzuordnen, der durch den zuletzt vorangegangenen Zeiteintrag in der Trace festgelegt wird. Durch dieses Vorgehen wird die Informationsmenge in der Trace auf die notwendigen Daten reduziert, denn alle Simulationsereignisse eines Zeitpunktes können so festgehalten werden, ohne die Zeitinformationen für alle Ereignisse und damit redundant aufzuzeichnen. Dieses Vorgehen wurde deshalb angestrebt, da auf diese Weise niemals die Datenmenge größer als nötig wird, denn im ungünstigsten Fall bleibt sie gleich (das ist der Fall, wenn für jedes Simulationsereignis ein neuer Zeiteintrag benötigt wird), in jedem anderen Fall wird sie reduziert.

3.5.1 Textuelle Repräsentation

Diese sequentielle Abhängigkeit der Trace-Einträge ist mit der Semantik von imperativen Programmiersprachen vergleichbar. Auch bei imperativen Programmiersprachen kann die Semantik eines Befehls durch die sequentielle Abarbeitung der vorangegangenen Befehle beeinflußt werden, so daß es für die textuelle Repräsentation einer Trace nicht abwegig ist, diese an typische Notationen imperativer Programmiersprachen anzugleichen. Diese Entscheidung kann durch die Beschriftungen, wie sie im Editorsystem benutzt werden, noch untermauert werden, denn auch dort wird eine Pascal-ähnliche Schreibweise für boolesche Ausdrücke und Wertzuweisungen benutzt. Der Benutzer des Editorsystems soll für die manuelle Eingabe eines Simulationsprotokolls, das i.a. die Belegung der globalen Eingänge beschreibt, oder das Lesen eines aufgezeichneten Protokolls mit möglichst keiner weiteren neu zu lernenden Notation konfrontieren werden. Deshalb wurde die gesamte textuelle Repräsentation eines Protokolls an die bestehenden Schreibweisen soweit möglich angelehnt.

Ein weiterer Aspekt beim Entwurf des textuellen Protokolls war die zu erwartende Dateigröße. Um die Simulation eines Automatensystems wiederholen zu können muß für alle Automaten das *Zeitverhalten* festgehalten werden, also alle Poll- und Cycle-Events, sowie Änderungen der treibenden Werte von globalen Eingangsvariablen. Die auf diese Weise anfallende Information soll nicht nur intern eine kompakte Darstellung haben sondern auch in einer Datei möglichst geringen Platzbedarf benötigen. Dennoch sollen die Daten nicht in Binärform abgelegt werden, sondern als Klartext, so daß sie ggf. auch manuell bearbeitet werden können. Aus diesen Überlegung heraus wurden kurze Schlüsselworte bzw. Symbole in der Dateirepräsentation benutzt. Für Verweise auf Variablen reicht es in der textuellen Repräsentation nicht aus, den Variablennamen festzuhalten, wie dieses in den Ausdrücken des Editors üblich ist. Da hier eine globalere Sicht auf ein System von Automaten erfolgt, müssen die Variablen eindeutig in diesem System identifizierbar sein. Dieses

Zeitverhalten

vollständiger Name

wird erreicht, indem Variablen über ihren *vollständigen Namen* angesprochen werden. Der vollständige Name entsteht dadurch, daß dem im Automaten eindeutigen Variablenname der (im System eindeutige) Automatenname durch einen Punkt getrennt vorangestellt wird und so eine eindeutige Zuordnung erreicht wird.

Trace-Eintrag	textuelle Repräsentation	Ereignis
Time-Entry	`AT` <Simulationszeit>	(alle)
Poll-Entry	`POLL` <vollst. Variablenname>	Poll-Event, (Input-Event)
Cycle-Entry	`CYCLE` <Automatenname>	Cycle-Event, (State-Event)
Driver-Entry	<vollst. Variablenname> `<-` <Wert-Ausdruck>	Driver-Event, *Benutzer-Interaktion*
Assign-Entry	<vollst. Variablenname> `:=` <Wert-Ausdruck>	Cycle-Event, Poll-Event, Assign-Event

Tabelle 3.1: Textuelle Repräsentation einfacher Trace-Einträge

Tabelle 3.1 zeigt, wie die einfachen Elemente einer Trace repräsentiert werden. Ein Time-Entry hält das Voranschreiten der Simulationszeit fest, indem der Simulationszeitpunkt angegeben wird, zu dem alle folgenden Trace-Einträge bis zum nächsten Time-Entry stattfinden. Daher kann es von jedem Simulationsereignis erzeugt werden. Beim Cycle-Entry wird nur der Automatenname im Protokoll festgehalten, denn der aktuelle Zustand des Automaten ergibt sich aus dessen bisherige Berechnung und der aktuellen Eingangsbelegung, so daß nur die einzige nichtdeterministische Information, nämlich der Zeitpunkt des Zyklusendes, festgehalten wird.

Beispiel 3.1: Dieses Beispiel zeigt, wie die textuelle Repräsentation des Trace-Ausschnittes aussehen könnte, dessen Ereignisse in Abbildung 3.3 auf Seite 39 dargestellt wurden. Hierzu wird angenommen, daß der Automat mit `A1` benannt wurde und entsprechend der Poll-Ereignisse zwei Eingangsvariablen, `in1` und `in2`, hat.

```
AT 56;
CYCLE A1;
AT 60;
POLL A1.in1;
AT 62;
POLL A1.in2;
AT 65;
CYCLE A1;
```

> Der gezeigte Traceausschnitt entsteht, wenn nur das Zeitverhalten des Automaten aufgezeichnet wird, nicht aber die Zuweisungen an die Eingangsvariablen. Ebenso würde das Protokoll besagen, daß keine Ausgangsvariablen zum Zyklusende manipuliert wurden, bzw. daß diese Informationen nicht aufgezeichnet wurden.

Ein Driver-Entry ist die Repräsentation eines Driver-Events in der Trace. Die Notation, die hier gewählt wurde, ist an die Schreibweise aus VHDL[6] angelehnt, wo durch eine Pfeil-Notation das Einplanen eines Wertes ausgedrückt wird. Die Wertänderung eines Treibers entspricht in etwa diesem Einplanen, denn der Zeitpunkt, zu dem der Wert gelesen wird (bzw. ob er überhaupt gelesen wird) ist nicht bekannt und wird erst durch ein Poll-Event festgelegt. Der Unterschied beider Notationen besteht jedoch darin, daß bei der VHDL-Schreibweise (`<=`) eine Verzögerung für die Übernahme des Wertes angegeben wird und die Variable, bzw. das Signal, zu dem so eingeplanten Zeitpunkt den neuen Wert annimmt. In der hier definierten Protokollsprache drückt der Pfeil `<-` aus, daß der Treiber der Variablen den Wert zum aktuellen Zeitpunkt annimmt, wann die Variable diesen Wert übernimmt, kann jedoch nicht bestimmt werden. Folgt eine weitere Treibereinplanung vor dem nächsten Pollen der Variable, so wird eine solche Werteinplanung ignoriert. Wird ein Protokoll zur Steuerung einer Simulation genutzt, dann sollten diese Wertzuweisungen an Treiber nur für solche Variablen erfolgen, die globale Inputs darstellen, denn sonst wird der Wert einer Ausgangsvariable eines Automaten durch den direkten Zugriff möglicherweise derart geändert, daß das Automatenverhalten nicht mehr seiner Semantik entspricht. Um einen solchen Eingriff in die Semantik nicht versehentlich zu provozieren, werden die Trace-Optionen so vorbelegt, daß Treiberinformationen nur für globale Inputs gelesen werden. Stellt der Benutzer das Lesen der Treiberinformationen für andere Eingangsvariablen ein, die demnach mit Ausgangsvariablen verbunden sind, so informiert eine Warnung über diesen Eingriff.

Assign-Entries repräsentieren Wertänderungen von Variablen, die durch die Aktionen einer Transition oder durch das Lesen eines Treiberwertes ausgelöst werden. Sie dienen dazu, die Belegung einer Variablen im Verlauf einer Simulation beobachten zu können (z.B. durch die Visualisierungskomponente), sollten aber nicht – obwohl der Simulator dieses optional zuläßt – als Eingabe dienen, da sie einen aus semantischer Sicht nicht zulässigen Eingriff in den SPS-Automaten darstellen. Sie werden aber dennoch in einem Eingabeprotokoll zugelassen, um den Benutzer in die Lage zu versetzen, ganz bestimmte Konstellationen zu überprüfen, ohne den Automaten auf „legalem“ Wege in diesen Status versetzen zu müssen. Zu diesem Zweck wird bei der Simulation mit Assign-Entries im Eingabeprotokoll für diese Einträge

[6] **V**ery High Speed Integrated Circuits **H**ardware **D**escription **L**anguage, eine weit verbreitete Hardwarebeschreibungssprache [IEE87, Ash90]

Assign-Event

jeweils ein *Assign-Event* generiert, das in der Vorbereitungsphase den zu setzenden Wert bestimmt, in der Ausführungsphase die Variable mit dem neuen Wert belegt und in der Mitteilungsphase die Protokoll-Übersetzung über diese Wertänderung benachrichtigt.

Beispiel 3.2: Für den folgenden Ausschnitt aus einer Trace wird wieder angenommen, daß der simulierte Automat `A1` heißt. Er soll die Eingangsvariable `in1` vom Typ `bool` und die Ausgangsvariable `out1` vom Typ {`1..3`}. Bis zum Zeitpunkt 31 soll der Treiber der Eingangsvariable den Wert `false` angenommen haben und wurde dann interaktiv vom Benutzer auf den Wert `true` gesetzt. Nachdem der Wert der Eingabe gepollt wurde, löst er eine Zustandswechsel aus, der den Ausgang auf den Wert `3` setzt.

```
AT 28;
POLL A1.in1;
A1.in1 := false;
AT 31;
A1.in1 <- true;
AT 33;
CYCLE A1;
AT 34;
POLL A1.in1;
A1.in1 := true;
AT 35;
CYCLE A1;
A1.out1 := 3;
```

Das Beispiel zeigt auch sehr gut, daß für einen SPS-Automaten immer nur der gepollte Wert entscheidend ist. Obwohl beim Zyklusende zum Zeitpunkt 33 der Treiber der Eingangsvariable bereits den Wert `true` angenommen hat, erfolgt der Zustandsübergang mit der Wertzuweisung erst zum Zeitpunkt 35. Selbst wenn eine Treibereinplanung `A2.in1 <- A1.in1` für den Zeitpunkt 33 in die Trace eingetragen würde, so entspräche dieses der Einplanung `A2.in1 <- false`, da die Eingangsvariable noch nicht den Treiberwert angenommen hat.

3.5.2 Erweiterung des Simulations-Protokolls zum high-level Eingabeprotokoll

Einige Erweiterungen sollen das Simulations-Protokoll derart erweitern, daß es auf effektivere Art als Sprache zur Beschreibung von Eingangsbelegungen durch den Benutzer genutzt werden kann. Hierzu sind im Protokoll bereits einige Konstrukte unterschiedlicher Komplexität vorgesehen.

Zu den einfachen Erweiterungen zählt das Zulassen von Ausdrücken über Werte und Variablen mit denselben Operationen, die bei der Automatenbeschreibung zulässig sind. Diese Ausdrücke werden zu dem Zeitpunkt ausgewertet, zu dem das entsprechende Ereignis stattfindet, so daß für verwendete Variablen die dann gültigen Werte zur Berechnung herangezogen werden.

Kleine Erweiterungen gibt es auch um das Voranschreiten der Zeit auszudrücken. Zum einen werden relative Zeitangaben erlaubt, so daß sich der Simulationszeitpunkt eines „`WAIT` <Zeitspanne>“ -Eintrags durch die Addition der ihm zugeordneten Zeitspanne zum vorher bestimmten Simulationszeitpunkt ergibt. `AT`- und `WAIT`-Einträge können ebenso durch ein Intervall statt einer Konstanten ergänzt werden. Die Semantik dieser Einträge entspricht denen, die man gewinnt, wenn das Intervall durch einen zufälligen Wert aus dem Intervall ersetzt würde.

Schließlich können Variablen zu Vektoren zusammengefaßt werden, die so eine abkürzende Schreibweise erlauben. Wird häufig für eine Menge von Variablen zum gleichen Simulationszeitpunkt eine Treiberänderung eingeplant, so lassen sich diese zu einem Vektor mit frei wählbarem Namen zusammenfassen. Auf diese Weise läßt sich die Folge von Einträgen zu einem und damit übersichtlicherem zusammenfassen.

Beispiel 3.3: Ein kleiner Traceausschnitt soll diese Einträge deutlich machen. Vor der eigentlichen Trace wird der Vektor definiert, der hier die Eingabevariablen `in1` und `in2` des Automaten `A1` zusammenfaßt.

```
VECTOR input := (A1.in1, A1.in2);
```

Um nun für beide Variablen zum selben, aber beliebigen Zeitpunkt im Intervall von 30 bis 40 einen neuen Wert einzuplanen, können folgende Trace-Einträge benutzt werden. Hierbei hängt der Wert der zweiten Variable von den dann aktuellen Werten der lokalen Variable `A2.local1` und der Ausgangsvariable `A3.out1` ab.

```
AT [30, 40];
input <- (false, A2.local1 or ( A3.out1 = 2 ));
```

3.5.3 Übersetzung einer Protokoll-Datei

Um einen aufgezeichneten Simulationslauf wieder ablaufen zu lassen, bzw. um die Steuerung von Eingaben aus einer Protokoll-Datei zu ermöglichen, muß eine solche Datei eingelesen und letztenendes in Simulationsereignisse übersetzt werden. Hierfür stehen prinzipiell zwei offensichtliche Wege zur Verfügung. Zum einen kann die Datei mit einem einfachen Parser gelesen werden, und sobald ein Ereignis im Eingabestrom identifiziert wurde, wird ein entsprechendes Ereignis in die Ereignisliste des Simulators eingetragen. Dieses Vorgehen hat zum Vorteil, daß der Großteil der einzulesenden Trace nicht im Arbeitsspeicher vorhanden sein muß und somit wäre die Größe des Automatensystems und die Länge des Eingabeprotokolls erheblich weniger durch die benutzte Hardware beschränkt. Allerdings ergibt sich das Problem, daß eine Trace mit dem derzeitigen Simulationskonzept nur begrenzt vorausgelesen werden kann. Es entspricht dem Problem bei der zentralen Interpretation, das im Abschnitt 3.5.4 beschrieben ist.

Der andere Weg, der schließlich auch in dieser Arbeit implementiert wurde, besteht darin, die gesamte Eingabe durch einen Compiler in eine interne Repräsentation übersetzen zu lassen. Der Nachteil bei diesem Vorgehen ergibt sich aus dem Vorteil des ersteren, denn alle identifizierten Ereignisse müssen in ihrer internen Darstellung festgehalten werden und belegen somit Speicher. Durch die Verwendung moderner Betriebssysteme, die über virtuellen Speicher einen sehr großen Arbeitsspeicher den Prozessen zur Verfügung stellen, die Verbilligung von Speicher und die relativ kompakten Datenstrukturen, die für die Simulationsereignisse anfallen, wird dieser Nachteil aber etwas entschärft. Ein Vorteil dieser Methode ist es aber, daß syntaktische Fehler bereits zu Beginn entdeckt werden und somit eine unnötige fehlerhafte Simulation vermieden werden kann. Außerdem wird auf diese Weise die verteilte Interpretation der Trace erst ermöglicht, da gelesene Ereignisse nicht unmittelbar verarbeitet werden müssen.

Wert-Ausdrücke

Syntaxbaum

Desweiteren wurde oben schon angedeutet, daß Werte für Treibereinplanungen nicht unmittelbar sondern als *Wert-Ausdrücke* möglich sein sollen. Hier bietet die Implementierung eines Compilers mit Hilfe eines Parsergenerators eine einfache Möglichkeit aus dem Eingabestrom einen entsprechenden *Syntaxbaum*[7] für diese Ausdrücke zu generieren. Maßgeblich für die Entscheidung, diese Ausdrücke durch einen neuen Compiler zu erzeugen, war die Tatsache, daß Variablen erst durch ihren vollständigen Namen eindeutig identifiziert werden. Ausdrücke über Variablen werden vom Compiler aus [Sch97] zwar in C-Quellcode transformiert, dann in ausführbare Programmteile übersetzt, und anschließend dynamisch zum Programm hinzugefügt. Der gesamte Vorgang vom Quellcode zur ausführbaren Funktion wird sogar durch die Dienste einer speziellen Klasse, dem Function-Producer, dem Programmierer bei der Entwicklung neuer Komponenten abgenommen. Die Berücksichtigung der vollständigen Namen wäre aber auch mit einigem Aufwand verbunden und da die zugelassenen Operationen bereits für die Klasse `TValue`, die die Werte der Variablen in der MOBY-Klassenbibliothek repräsentiert, implementiert sind, ist die direkte Auswertung eines Syntaxbaumes hier kaum aufwendiger. Eventuell würde sogar ein direkter Vergleich beider Methoden lohnenswert sein,

[7] Dieser Baum besteht aus den zugelassen Operatoren als innere Knoten und Variablen bzw. Konstanten als Blätter. Den Wert eines Ausdrucks errechnet man, indem man rekursiv den Operator der Wurzel auf die ausgewerteten Teilbäume anwendet.

da der Verwaltungsaufwand bei der Verwendung der dynamisch gebundenen Programmteile ebenfalls nicht unerheblich ist.

Durch die Methode der Kompilation entsteht so aus der textuellen Trace eines einfachen Eingabeprotokolls eine identische Datenstruktur, wie sie bei der Aufzeichnung der Simulation entstanden ist. Ebenso lassen sich einige komplexere Einträge bereits bei der Kompilation in einfachere übersetzen, so z.B. das Expandieren von Vektoren.

Für eine möglichst effiziente Übersetzung einer Protokolldatei werden in *Lexer* und *Parser* des Trace-Compilers einige Optimierungen vorgenommen. Der Lexer, auch Scanner genannt, ist die Komponente zur lexikalischen Analyse des Eingabestroms, der i.a. aus einer Datei gelesen wird, und produziert für die erkannten Sprachelemente sogenannte Token. Diese Token werden durch *reguläre Ausdrücke* beschrieben, so daß der Lexer nicht nur Schlüsselworte und Operatoren erkennt, sondern auch Kommentare, Bezeichner und Strings[8]. Der implementierte Lexer geht bei dieser Erkennung noch ein Stufe weiter, so daß identifizierten Bezeichner mit den Objekten im zu simulierende Automatensystem, das bekannt ist, verknüpft werden. Auf diese Weise ist es möglich, frühzeitig den Typ von Variablen zu erkennen und so die Grammatik für die Wert-Ausdrücke einfach und dennoch mächtig zu halten. Auf diese Weise kann vom Parser, der mit Hilfe des Compiler-Compilers *bison* erzeugt wird, direkt beim Parsen eines Wert-Ausdrucks der gewünschte Syntaxbaum erzeugt werden. In der erweiterten Grammatik, aus der der Parser automatisch erzeugt wird, kann von jeder Regel ein Objekt zurückgegeben werden, so daß der Aufbau des Syntaxbaumes allein durch die sogenannten semantischen Aktionen in dieser Grammatik aufgebaut werden kann.

Lexer und Parser

reguläre Ausdrücke

bison

3.5.4 Interpretation von Traces

Um eine Trace als Eingabesprache zu nutzen, die die Simulation beeinflußt, wird sie zunächst in eine interne Datenstruktur übersetzt. Diese interne Trace entspricht von der Struktur her aber immer noch genau der textuellen Form. Um aus der Trace wieder Simulationsereignisse generieren zu können gibt es prinzipiell zwei Vorgehensweisen. Zum einen könnte die Trace an zentraler Stelle interpretiert werden und die Simulation mit Ereignissen versorgen, zum anderen könnte aber auch jeder SPS-Automat die Trace für sich interpretieren und dabei nur die für ihn relevanten Ereignisse betrachten. Beide Methoden haben ihre speziellen Vorteile, die im folgenden erläutert werden.

Bei der verteilten Interpretation wird für jeden Automaten ein eigener Interpreter benutzt, der die Trace schrittweise durchgeht. Dieses Durchlaufen

verteilte Interpretation

[8] Eine Folge von beliebigen Zeichen, die durch besondere Zeichen, aus dem restlichen Eingabestrom abgegrenzt werden. Auch die Zeichenfolge eines Schlüsselwortes darf in so einer Zeichenkette enthalten sein.

der Trace wird jeweils vom Cycle-Event angestoßen. In der Vorbereitungsphase muß dieses Ereignis seine Folgeereignisse bestimmen. Dazu wird die Protokoll-Übersetzung in Anspruch genommen, die ihrerseits den für den Automaten zuständigen Interpreter heranzieht. Der Interpreter liest die Trace bis zum nächsten Cycle-Event des SPS-Automaten, für den er zuständig ist, höchstens aber für die maximale Zyklusdauer im voraus. Dabei werden alle Einträge für den Automaten – das sind Einträge, die das Pollen, Setzen eines Treibers und direkte Zuweisungen von Variablen dieses Automaten betreffen, sowie das neue Zyklusende – in entsprechende Simulationsereignisse überführt, sofern dieses nach den Trace-Optionen gewünscht ist, und dem aktuellen Cycle-Event als Folgeereignisse mitgeteilt.

Hierbei kann immer nur bis zum nächsten Zyklusende die Trace gelesen werden, da beim Erzeugen eines Cycle-Events der Zustand angegeben werden muß, der bis zum Eintreten dieses Ereignisses der aktuelle Zustand des Automaten sein wird. So wird der aktuelle Automatenzustand immer nur in dem Cycle-Event festgehalten, mit dem der Zyklus, in dem der Zustand angenommen wurde, endet. Sollen aber zwei Cycle-Events vorausgelesen werden, so muß dem zweiten Cycle-Event der bis dahin aktuelle Zustand, also der Folgezustand des ersten vorausgelesenen Cycle-Events, mitgeteilt werden. Dieser berechnet sich aber erst aus der aktuellen Variablenbelegung und dem aktuellen Zustand zu dem Zeitpunkt, zu dem das erste vorausgelesene Cycle-Event bearbeitet wird, so daß für das zweite die nötigen Informationen noch nicht vorhanden sind. Der Folgezustand des ihm vorangegangenem Cycle-Events kann nicht bestimmt werden, da die dann gültigen Variablenwerte noch nicht bekannt sind.

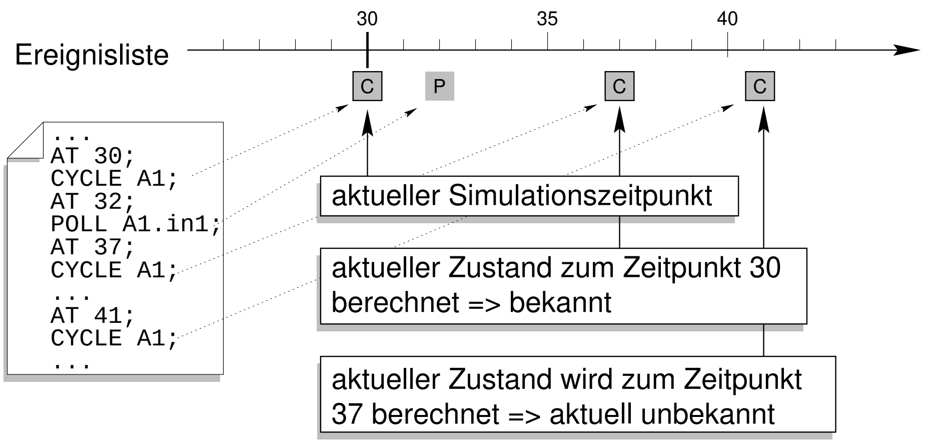

Abbildung 3.4: Problem beim Vorauslesen einer Trace

Abbildung 3.4 stellt diesen Sachverhalt graphisch dar. Die Simulation ist bis zum Zeitpunkt 30 vorangeschritten. Der gültige Zustand für den Zeitpunkt 37, zu dem das nächste Cycle-Event eingeplant wird, ergibt sich aus der aktuellen Variablenbelegung und dem aktuellen Zustand. Somit kann der dann gültige Zustand – der aktuelle Folgezustand – berechnet werden und damit auch das Cycle-Event erzeugt werden, denn diesem kann der dann aktuelle Zustand mitgeteilt werden. Um aber das Cycle-Event für den Zeitpunkt 41 erzeugen zu können, muß der aktuelle Zustand für den Zeitraum nach dem Zeitpunkt 37 bekannt sein. Dieser berechnet sich jedoch unter anderem aus dem Wert der Eingangsvariable, die zum Zeitpunkt 32 gepollt wird, also in der Zukunft. Folglich kann dieses Cycle-Event noch nicht zum Simulationszeitpunkt 30 erzeugt werden, da ihm der dann aktuelle Zustand nicht zur Verfügung steht.

Der Vorteil der verteilten Interpretation liegt nun darin, daß sie unmittelbar mit dem in [Sch97] entwickelten vorhandenen Simulationskonzept realisierbar ist. Der Nachteil hierbei entsteht jedoch, wenn das Eingabe-Protokoll keine exakte, sondern eine unscharfe Beschreibung der Eingangssignale enthält. Die ersten Probleme tauchen auf, wenn ein unscharfer Zeitpunkt in der Eingabetrace beschrieben wird. Wird bei einem Zeiteintrag ein Intervall statt eines Zeitpunktes angegeben, so soll ein beliebiger Wert aus dem Intervall als Zeitangabe gewählt werden. Ein zweiter Automat muß aber diesen Zeiteintrag als selben Zeitpunkt interpretieren. Der folgende Auszug aus einem Protokoll soll dieses verdeutlichen.

```
AT [10, 20];
A1.in1 <- true;
WAIT 1;
A2.in1 <- false;
```

Das Beispiel soll ausdrücken, daß das folgende Ereignis in der Trace zwischen den Simulationszeitpunkten 10 und 20 stattfindet, wobei der Treiber der Variable `in1` des Automaten `A1` seinen Wert auf `true` ändert. Eine Zeiteinheit nach diesem Ereignis ändert der Treiber der Variable `in1` des Automaten `A2` seinen Wert auf `false`. Interpretieren die Automaten `A1` und `A2` den `AT`-Eintrag aber unabhängig, so könnte `A1` hierfür den Zeitpunkt 20 annehmen, zudem der Treiber seiner Eingangsvariable seinen Wert ändert. Der zweite Automat könnte den `AT`-Eintrag aber als Zeitpunkt 10 interpretieren. Er überliest dann den Eintrag für den ersten Automaten und erhöht die Zeit aufgrund des `WAIT`-Eintrags auf 11. So findet die Wertänderung am Treiber der Variable `in1` des Automaten `A2` noch vor der anderen Treiberänderung statt.

Dieses Verhalten wird verhindert, indem solche Einträge sich nach dem ersten Auslosen eines Zeitpunktes bzw. einer Zeitdauer so verhalten als sei ein fester Wert vorgegeben. Beim Neustart der Simulation müssen deshalb diese Einträge wieder zurückgesetzt werden, damit wieder ein neuer zufälliger Zeitpunkt gewählt wird. Auf ähnliche Weise muß mit Einträgen verfahren werden, die nicht direkt einem Automaten zugeordnet werden können. Der erste Automat, der ein solches Ereignis interpretiert, sorgt für das Einfügen eines entsprechenden Ereignisses im Scheduler und markiert den Eintrag als bearbeitet, damit ein anderer Automat nicht wiederholt das Ereignis erzeugt.

Das Problem kann aber nicht mehr auf solch einfache Weise gelöst werden, wenn Schleifen in der Spezifikation der Trace erlaubt werden. Ein Trace-Teil, der einen zufallsbehafteten Zeiteintrag enthält und ständig wiederholt wird, bis die Simulation stoppt, kann hier das Problem verdeutlichen. Wenn zwei Automaten diese Trace unabhängig voneinander interpretieren, sie aber sehr unterschiedliche Zykluszeiten haben, so kann es sein, daß der erste Automat die Schleifeneinträge bereits etliche Male interpretiert hat. Damit müßten für den zufallsbehafteten Zeiteintrag auch schon ebenso oft Werte ausgelost worden sein, die ein zweiter Automat in eben dieser Reihenfolge benutzen müßte. Somit muß für jeden Schleifendurchlauf festgehalten werden, welcher Wert ausgelost wurde, und jeder Automat müßte wissen, wie oft er die Schleife durchlaufen hat, um den richtigen Wert für seinen aktuellen Durchlauf zu erhalten. Dieses hohe Datenaufkommen würde durch eine einmalige Interpretation und damit direkte Übersetzung vermieden werden.

zentrale Interpretation

Bei einer zentralen Interpretation wird bei jeder Anfrage an den Interpreter die Trace für die maximale Zykluszeit vorausgelesen und alle hierbei anfallenden Trace-Einträge in entsprechende Simulator-Ereignisse übersetzt, nicht nur die Einträge für einen Automaten. Automatenspezifische Einträge können dann auf Anfrage bei der Vorbereitungsphase eines Cycle-Events diesem als Folgezustände mitgeteilt werden, wobei alle „nicht abgeholten" Ereignisse gesammelt werden. Dieses Vorgehen erlaubt auch ein direktes Behandeln von Einträgen, die keinem Automaten zugeordnet werden können, da sie direkt in die Ereignisliste eingetragen werden können.

Die Problematik bei diesem Vorgehen liegt aber genau in dem Punkt, weshalb bei der verteilten Interpretation nicht weiter voraus gelesen werden kann als bis zum nächsten Zyklusende für den gerade betrachteten Automaten. Stößt der Interpreter beim Vorauslesen auf einen Eintrag für das Zyklusende eines Automaten, von dem nicht die aktuell bearbeitete Anfrage ausging, so ist der bis zu diesem Zyklusende gültige Automatenzustand nicht bekannt und es kann damit kein Cycle-Event erzeugt werden. Ein späteres Hinzufügen des Zustandes ist auch nicht möglich, da ein Zugriff auf dieses Ereignis erst wieder erfolgt (und auch nur erfolgen kann), wenn das Ereignis feuern soll. Zu diesem Zeitpunkt wird aber bereits der aktuelle Zustand wieder benötigt.

Der Ausweg besteht darin, daß der SPS-Automat die Information über seinen aktuellen Zustand erhält. Ebenso, wie die aktuelle Belegung der Variablen im Automaten selbst festgehalten wird, kann eine Instanzvariable Auskunft über den zur aktuellen Simulationszeit aktiven Zustand geben. Der Simulationsablauf würde so geändert werden, daß ein Cycle-Event nur noch die Information erhält, für welchen Automaten zum Zeitpunkt des Feuerns der

Zyklus beendet wird. Der aktuelle Zustand würde zu diesem Zeitpunkt (in der Vorbereitungsphase) vom Automaten erfragt werden, und der neu ermittelte Folgezustand kann in der Ausführungsphase dem Automaten mitgeteilt werden. Auf diese Weise liegt immer die benötigte Information für das Erzeugen eines Cycle-Events vor, nämlich der Automat, für den der Zyklus endet – jegliche andere Information für das Feuern des Ereignisse wird hingegen erst zu eben jenem Zeitpunkt benötigt und liegt dann im Automaten vor.

Ein andere Möglichkeit wäre es, die Information über den aktuellen Zustand im Protokoll mit festzuhalten. Für die interne Repräsentation der Trace würde auf diese Weise kein zusätzlicher Speicherbedarf entstehen, denn ob eine Referenz auf einen Automaten oder einen Automatenzustand gespeichert wird bleibt sich gleich. In der textuellen Repräsentation würde sich allerdings eine Erweiterung ergeben, da zum Automatennamen auch der Zustandsname festgehalten werden müßte. Dieses Alternative würde eine Protokolldatei somit vergrößern, was aber nicht hauptausschlaggebend für die Ablehnung ist. Das Bestreben in die Protokolldatei ausschließlich benötigte Informationen aufzunehmen, war das Hauptentwurfskriterium und ist damit der Hauptgrund gegen diese Alternative. Obige Lösungsansätze zeigen, daß das Ziel auch ohne diese zusätzlichen Informationen erreicht werden kann.

3.6 Unterbrechungspunkte (Breakpoints)

Das Überprüfen einer Spezifikation oder eines Systementwurfs mit SPS-Automaten kann dem Benutzer für überschaubare Fälle durch die (interaktive) Simulation ermöglicht werden. Bei zu simulierenden Automatensystem werden jedoch häufig schon bei kleineren Applikationen relativ viele Variablen herangezogen, so daß besonders die Beobachtung mehrerer Variablen über den Verlauf der Simulationszeit schnell unübersichtlich und benutzerunfreundlich wird. In diesem Bereich werden häufig sogenannte Breakpoints, also *Unterbrechungspunkte* festgelegt, die dem Benutzer eine automatisierte Beobachtung des Systemzustandes ermöglichen. Durch die Beobachtung aller Simulationsereignisse in der Protokoll-Übersetzung der Hintergrundsimulation wird die Integration von Methoden zur Behandlung von Unterbrechungspunkten auf effektive Weise möglich.

Unterbrechungspunkt

Die Art und Weise, wie Unterbrechungspunkte bestimmt werden, wird in der Regel auf die entsprechende Anwendung angepaßt. Für die Simulation von SPS-Automaten scheint die Betrachtung der Variablen, die in den Automaten definiert und während der Simulation geändert werden, am sinnvollsten, da sie im wesentlichen[9] den Zustand des Systems beschreiben. Aus diesem Grund

[9] Unter Zuhilfenahme der speziellen states-Variable, wie sie in [Sch97] eingeführt wurde, wird durch die Betrachtung der Variablen der Systemzustand vollständig beschrieben.

wird jedem Unterbrechungspunkt ein boolescher Ausdruck über die Variablen des zu simulierenden Automatensystems zugeordnet. Um zeitbehaftete Beobachtungen zu unterstützen kann zusätzlich die Angabe eines Zeitraumes erfolgen, so daß ein Unterbrechungspunkt *aktiviert* wird, wenn seine Bedingung für die gewählte Zeitdauer ohne Unterbrechung erfüllt war. Damit ein praktischer Nutzen durch Unterbrechungspunkte entsteht, müssen sie Aktionen im Simulator auslösen können, so z.B. die Simulation anzuhalten. Gerade wenn mehrere Unterbrechungspunkte definiert sind, reicht diese Aktion häufig nicht aus, deshalb kann zu jedem Unterbrechungspunkt eine Nachricht angegeben werden, die dem Benutzer beim Aktivieren präsentiert wird, so daß dieser Rückschlüsse darauf ziehen kann, wodurch die Unterbrechung ausgelöst wurde.

aktiviert

3.6.1 Einbettung in die Protokoll-Übersetzung

Durch die Einführung der Protokoll-Übersetzung ist im Simulator eine zentrale Instanz vorhanden, die von allen Variablenänderungen Notiz nimmt, und die deshalb für die Verwaltung der Unterbrechungspunkte prädestiniert ist. Aus diesem Grund werden auch die Unterbrechungspunkte zusammen mit anderen Informationen der Protokoll-Übersetzung in der Datenstruktur der Trace-Optionen verwaltet. Wenn nun während der Simulation Variablen ihren Wert ändern, so überprüft die Protokoll-Übersetzung nicht nur, ob diese Änderungen in das Simulations-Protokoll aufgenommen werden, sondern auch, ob sich hierdurch Konsequenzen für die Unterbrechungspunkte ergeben.

Die Effektivität dieser Überprüfungen wird erreicht, indem die Bedingung möglichst selten ausgewertet werden muß. Wie auch schon bei den Ausdrücken im Eingabeprotokoll für Variablen-Treiber Zuweisungen (Werteinplanungen), werden auch für die Bedingungen der Unterbrechungspunkte Syntaxbäume aufgebaut. Bevor nun die Bedingung ausgewertet wird, wird überprüft, ob eine geänderte Variable überhaupt darin benutzt wurde. Erst wenn dieses der Fall ist, wird die Bedingung ausgewertet. Falls die Bedingung nun zum Zeitpunkt der Variablenänderung wahr wird, so muß noch die Zeitrestriktion erfüllt werden. Dazu wird für den entsprechenden Simulationszeitpunkt[10] ein Ereignis in die Ereignisliste des Simulators eingetragen.

Break-Event

Auch dieses Ereignis, ein *Break-Event*, folgt dem Muster der anderen Ereignisse beim Feuern. In der Vorbereitungsphase wird festgestellt, ob der entsprechende Unterbrechungspunkt noch aktiviert ist, also ob seine Bedingung erfüllt ist und ihren Wert seit der festgelegten Zeitspanne nicht mehr geändert hat. Dieses wird festgestellt, indem der Unterbrechungspunkt sich den Zeitpunkt seiner letzten Bedingungsänderung merkt und so durch Vergleich mit

[10] Der mögliche Aktivierungszeitpunkt ergibt sich, wenn zum aktuellen Simulationszeitpunkt die Zeitspanne des Unterbrechungspunktes addiert wird.

der aktuellen Simulationszeit einem Break-Event mitteilen kann, ob er aktiviert ist. In der Ausführungsphase wird dann die zugeordnete Aktion (z.B. des Anhalten des Simulators) veranlaßt, falls eine Aktivierung noch vorlag. Ebenso wird in der Mitteilungsphase die Protokoll-Übersetzung über das Feuern des Ereignisses informiert, sofern eine Aktivierung noch bestand. Die Protokoll-Übersetzung wiederum ist für die Präsentation der Nachricht für den Benutzer verantwortlich. Außerdem hält sie die Aktivierung des Unterbrechungspunktes im Simulations-Protokoll fest.

Das Festhalten aktivierter Unterbrechungspunkte in der Trace der Protokolldatei kann für eine spätere Betrachtung des Simulationslaufs sinnvoll sein. Die Interpretation und das wiederholte Ausführen dieser Einträge scheint jedoch unnötig, und zwar weil die Protokoll-Übersetzung beim Schreiben des Simulations-Protokolls hierin auch die verwendeten Unterbrechungspunkte ablegt. Bei einer wiederholten Simulation sollten also bei korrektem Ablauf mit gleichem Zeitverhalten der Automaten auch Unterbrechungspunkte zu den gleichen Zeitpunkten aktiviert sein. Ist dieses nicht der Fall, so muß davon ausgegangen werden, daß die Simulation nicht exakt wiederholt wird, d.h. es wird ein anderes Zeitverhalten oder es werden andere Eingangssignale benutzt, und damit dürften auch die Unterbrechungen bzw. Meldungen des vorherigen Laufs hinfällig sein.

3.6.2 Unterbrechungen als Trace-Einträge

Eine Erweiterung dieses Konzeptes stellen Zusicherungen dar, die in eine Eingabe-Trace eingefügt werden können. Die Möglichkeit eine Simulation anhand einer Protokolldatei zu wiederholen, erlaubt es auch, dieses Protokoll von einer anderen Quelle erzeugen zu lassen. Im konkrete Anwendungsfall soll durch einen Modelchecker eine mögliche Fehlertrace erzeugt werden, wobei man aber teilweise Abstraktionen in Kauf nimmt, um die Modellbeschreibung klein zu halten und damit den Zeitbedarf des Modelchecker zu verringern. Durch diese Abstraktionen kann jedoch der Fall eintreten, daß eine Trace erzeugt wird, die vom zu Grunde gelegten SPS-Automaten gar nicht erzeugt werden könnte.

Die Erweiterung des Unterbrechungspunkt-Konzeptes besteht nun darin, daß Bedingungen als Einträge in die Trace gebracht werden, zu denen überprüft wird, ob das System einen erwarteten Zustand angenommen hat. Diese Bedingungen entsprechen ebenso denen der Unterbrechungspunkte wie die Aktionen, die ausgelöst werden könnten. Deshalb handelt es sich bei diesen Konsistenzprüfungen um einen Spezialfall der Unterbrechungspunkte, die nur zu genau ihrem Zeitpunkt in der Trace überprüft werden. Folglich können diese speziellen Bedingungen durch die gleichen Break-Events modelliert werden.

3.7 Visualisierung

Die Visualisierung soll es erlauben, eine zuvor aufgezeichnete Simulation in ihrem zeitlichen Verlauf betrachten zu können. Zu diesem Zweck wurden verschiedene Methoden der graphischen Darstellung in Betracht gezogen, insbesondere solche, die bereits mit der Moby-Klassenbibliothek implementiert wurden.

3.7.1 Vorhandene Visualisierungskomponenten

MSC

Zum einen wurde hierbei an die Verwendung der *MSC*-Komponente, die von Xia Gao in ihrer Diplomarbeit [Gao97] im Moby-Kontext entwickelt wurde, gedacht. MSC ist die Abkürzung für Message Sequence Charts, bei denen es sich um eine Trace-Sprache mit graphischer und textueller Darstellung handelt (s. z.B. [RGG96, Mau96]). Diese finden Verwendung, um die Abläufe in einem *SDL*[11]-Diagramm zu visualisieren, so auch im Editor- und Simulationssystem Moby/sdl (s. hierzu auch [Tap95, Arn97]). Der Schwerpunkt von SDL liegt in der Beschreibung kommunikativer Prozesse, so daß diese Spezifikationssprache häufig in Bereichen der Telekommunikation Anwendung findet. Demzufolge ist der kommunikative Aspekt auch bei MSCs besonders ausgeprägt. Die Tatsachen, daß die MSC-Komponente in der Abteilung entwickelt und zudem unter Verwendung der Moby-Klassenbibliothek implementiert wurde, ließe es zu, den Aufwand für die Verwendung in der SPS-Simulation gering zu halten. Das Problem hierbei liegt jedoch gerade in der Ausprägung der Kommunikation, denn diese steht bei den MSCs deutlich im Vordergrund, während sie bei SPS-Automaten eine untergeordnete Rolle einnimmt, da sie bei letzteren nur asynchron erfolgt. Ein Übersetzung von Simulationsläufen bei SPS-Automaten in MSC scheint daher unangebracht.

SDL

Constraint Diagrams

Ein weiterer Ansatzpunkt war die Verwendung von *Constraint Diagrams*, kurz CDs ([Die96]). Diese CDs erlauben besonders gut, zeitliche Veränderungen von sogenannten *Observablen* zu beschreiben, insbesondere lassen sich auch deren zeitliche Abhängigkeiten ausdrücken. So schien eine Verwendung der Editor-Komponente hierzu, die in der Diplomarbeit [Oet98] von Marco Oetken entwickelt wurde, zunächst angebracht. Allerdings besteht hier das Problem, daß CDs nicht ganz das wiedergeben, was durch die Trace eines Simulationsprotokolls ausgedrückt wird. Die Trace-Sprache ist sehr speziell auf SPS-Automaten angepaßt, während die größte Gemeinsamkeit von CDs mit SPS-Automaten lediglich darin besteht, daß auch sie eine formale Semantik durch Formeln des Duration Calculus erhalten haben. CDs dienen eher als Spezifikationssprache, so daß sie unter Beibehaltung der Semantik in SPS-Automaten übersetzt werden können. Eine Nutzung dieser Komponente für

[11] **S**pecification and **D**escription **L**anguage - [BHS91]

die Simulationsvisualisierung, die eine streng deterministische Trace darstellen soll, würde auf eine stark spezialisierte Klassen von CDs hinauslaufen, wobei wieder ein Grundkonzept dieser Spezifikationssprache – die zeitliche Abhängigkeit von Änderungen verschiedener Observablen – in den Hintergrund gestellt würde, so daß auch diese Art der Visualisierung nicht angemessen scheint.

Von der Projektgruppe IDEFIX wurde in [SBS+95, SBS+96] ebenfalls eine Visualisierungskomponente für Simulationsdaten entwickelt. Die darzustellenden Daten fallen hierbei durch die Simulation von *Thor-Netzen*[12] an, und sind in ihrer Art den Daten der SPS-Simulation sehr ähnlich. In Diagrammen wird bei IDEFIX die zeitliche Wertänderung von Netzausdrücken dargestellt, die eine Erweiterung von zu beobachtenden Variablen darstellen, so daß diese Darstellung direkt für die SPS-Visualisierung genutzt werden kann. Leider kann jedoch die Implementierung nicht unverändert übernommen werden. Auch die IDEFIX-Implementierung baut auf die MOBY-Klassenbibliothek auf, doch leider ist die Entwicklung der Bibliothek auf zwei getrennten Wegen weiterverfolgt worden. So ist eine direkte Übernahme der entwickelten Klassen aufgrund von Änderungen in den Basisklassen nicht möglich ist. Dadurch, daß die Quellen an die aktuelle MOBY-Klassenbibliothek angepaßt werden müssen, kann aber auch bereits die Anpassung an die SPS-Simulation erfolgen, so daß dieser Weg noch immer vorteilhafter ist als eine eigene Implementierung.

Thor-Netze

3.7.2 Konzepte der Visualisierung

Aus der IDEFIX-Implementierung werden für die hier zu erstellende Visualisierung zunächst zwei Diagrammtypen übernommen, *Kurven-Diagramme* und sogenannte *Gantt-Diagramme*. Bei den Kurven-Diagrammen kann die zeitliche Belegung von Integer-Variablen dargestellt werden, also Variablen mit Ganzen Zahlen als Wertebereich. Für jede Variable wird dabei eine Kurve in das Diagramm eingezeichnet, wobei der aktuelle Wert über den zeitlichen Verlauf aufgetragen wird. Die Gantt-Diagramme dienen zur Darstellung von booleschen Variablen. Über die Zeit, die die Variable den Wert $true$, annimmt, wird eine durchgezogene Linie gezeichnet, Bereiche ohne diese Linie stellen den Wert $false$ dar.

Kurven- und Gantt-Diagramm

In beiden Diagrammtypen wird bei IDEFIX eine Datenstruktur benötigt, die Koordinatenpaare von Zeit und aktuellem Wert der Variable beinhaltet. Bei dieser Datenstruktur erfolgt bereits eine Anpassung auf die neueren MOBY-Klassen. Für Variablenbelegungen wird bei den aktuellen MOBY-Produkten die Klasse `TValue` benutzt. Im SPS-Simulator werden zwar zur

[12] Thor-Netze sind höhere Petri-Netze, die um ein Hierarchie– und Zeitkonzept erweitert wurden

Zeit nur einfache Typen zugelassen, prinzipiell können aber durch die Verwendung dieser Klasse auch komplexere Datentypen behandelt werden. Deshalb wird in der Datenstruktur der Wert einer betrachteten Variable als ein Objekt dieser Klasse festgehalten. Für die Simulationszeit wurde im SPS-Simulator ein eigener Typ eingeführt, so daß die Datenstruktur an dieser Stelle auf SPS-Automaten angepaßt ist.

Eine solche Datenstruktur von Paaren, bestehend aus Simulationszeitpunkt und Wert, kann nun mit den von IDEFIX entwickelten Komponenten dargestellt werden. Allerdings ist auch hier eine Anpassung erforderlich, denn die Diagramme sind von der Klasse `Graph` abgeleitet, stellen also spezialisierte Graphen dar und somit muß die Darstellung der Diagramme an neue Methoden der Klasse `GraphView` angepaßt werden. Diese Änderungen sind jedoch rein technischer Art, so daß eine Übernahme der Klassen gerechtfertigt bleibt. Die Weiterentwicklungen der Basisklassen seit der IDEFIX-Implementierung lassen allerdings auch komfortablere Darstellungen zu, die in die hier entwickelte Visualisierungskomponente eingearbeitet werden können. Dieses kann aber leider aus zeitlichen Gründen nicht mehr im Rahmen der Diplomarbeit geschehen.

Der letzte Schritt zur Darstellung der Simulationsergebnisse besteht nun darin, aus dem Simulationsprotokoll die oben beschriebene Datenstruktur zu gewinnen. Hier wird wieder der Ansatz verfolgt, der bei der Wiederholung einer Simulation Anwendung findet. Die Protokoll-Datei wird durch den entwickelten Compiler in eine interne Datenstruktur übersetzt. Bei diesem Vorgang wird allerdings das simulierte SPS-Automaten-Modell benötigt, damit Variablen und Automaten richtig identifiziert werden können. Es folgt dann eine Interpretation der Trace, wobei jetzt speziell Wertzuweisungen an Variablen betrachtet werden und dementsprechend die Zeit/Werte-Paare für jede Variable in einer eigenen Datenstruktur festgehalten werden.

Durch dieses Vorgehen sind nur spezielle Protokolle für die Visualisierung geeignet, nämlich solche, bei denen zum einen der zeitliche Ablauf der Ereignisse deterministisch ist – dieses ist allerdings bei einer aufgezeichneten Simulation immer der Fall – und das die Wertänderungen der aus der Anwendungssicht relevanten Variablen enthält.

3.8 Zusammenfassung der Protokollsprachen

Aus den vorangegangenen Abschnitten lassen sich nun drei Anwendungen der entwickelten Protokollsprache ableiten. Zunächst kann ein Protokoll dazu dienen, eine Simulation mit der nötigen Eingabe zu versorgen. Zu diesem Zweck enthält die Protokollsprache nichtdeterministische Elemente, die Möglichkeit, Optionen wie z.B. Unterbrechungspunkte festzulegen und höhere Sprachekonzepte wie z.B. Vektoren. Sprachkonstrukte für das Zeitverhalten sind in einem solchen Protokoll nicht sinnvoll, da dieses erst durch die Simulation ermittelt werden soll – die volle Flexibilität des SPS-Verhaltens soll in einem solchen Fall ausgeschöpft werden. So kann ein Eingabeprotokoll auch benutzt werden, um mehrere verschiedene Szenarien durchzuspielen, da gerade der zeitliche Verlauf der Ereignisse im SPS-Zyklus durch die Semantik nicht fest vorgegeben ist.

Für die Wiederholung einer Simulation dürfen viele dieser Eingabeelemente keine Verwendung mehr finden, um Determinismus zu erreichen. In einem solchen Protokoll muß demnach das exakte Zeitverhalten aller Komponenten festgehalten werden, da nur so eine gleichwertige Simulation garantiert werden kann. Ebenso ist es notwendig, die Änderungen der globalen Inputs festzuhalten, da sie nicht durch andere Ereignisse bestimmt werden. Alle weiteren Sprachkonstrukte können bzw. müssen im *Wiederholungsprotokoll* weggelassen werden, da sie bereits durch das Automatensystem bestimmt werden. Über die Trace-Optionen läßt sich festlegen, daß gerade nur solche Ereignisse, die zur Reproduktion einer Simulation notwendig sind, festgehalten werden – die vom Programm vorgegebene Standardeinstellung dieser Optionen berücksichtigt genau diesen Fall, da dieses voraussichtlich die Hauptaufgabe ist.

Wiederholungs-protokoll

Ein Wiederholungsprotokoll kann wiederum zur Simulationssteuerung herangezogen werden, um so ein Protokoll für die Visualisierungskomponente zu erhalten. Bei der Reproduktion einer Simulation macht es dann keinen Sinn, wieder alle Ereignisse aufzuzeichnen. Vielmehr wird man in diesem Fall lediglich alle Wertänderungen von Variablen in das Protokoll aufnehmen, um deren zeitlichen Verlauf in der Visualisierungskomponente betrachten zu können. Als Erweiterung wäre noch die Aufnahme von Treiberänderungen möglich, um beobachten zu können, wie viele Wertänderungen durch den pollenden Charakter der SPS-Automaten ignoriert werden. Aber auch hier macht nur die Aufnahme von Treibern für solche Eingangsvariablen Sinn, deren treibende Ausgangsvariable nicht aufgezeichnet wird. Eine Erweiterung der Visualisierungskomponente könnte auch das Aufzeichnen der Poll- und Cycle-Events sinnvoll machen. Dazu wären ein neuer Diagrammtyp anzufertigen der evtl. ähnlich wie die Gantt-Diagramme diese Ereignisse darstellt. Die objektorientierte Implementierung müßte den Aufwand für eine solche Erweiterung relativ gering machen.

Kapitel 4

Implementierung

Die Implementierung des Entwurfs aus dem vorangegangen Kapitel erfolgt auf der Grundlage der Moby-Klassenbibliothek, die auch für die bereits vorhandenen Programmteile die Grundlage darstellte. Bei der Moby-Klassenbibliothek handelt es sich um eine C++ Klassenbibliothek, die durch die Diplomarbeit von Andree Seidel ([Sei93]) mit Anlehnung an eine Smalltalk-Bibliothek entstanden, und in diversen Studien- und Diplomarbeiten, sowie Projektgruppen und der Arbeit in der Abteilung zu ihrem heutigen Umfang gewachsen ist. Dieses Kapitel soll insbesondere einige Einzelheiten der Implementierung herausheben und erläutern

4.1 Allgemeines, Hard- und Software

Da die Moby-Klassenbibliothek eine C++-Klassenbibliothek ist, wird als Programmiersprache in dieser Diplomarbeit ebenfalls C++ verwendet. Sprache für die Codierung von Programme der Moby-Reihe ist Englisch, d.h. sämtliche Bezeichner und Kommentare des entwickelten Programm-Codes (Quelltext) sind in englischer Sprache abzufassen. Desweiteren wurde seit der Projektgruppe IEKOΣ ein ausführlicheres Kommentieren der Header-Dateien eingeführt, so daß neue Klassen inzwischen gut kommentiert sind (bzw. sein sollten). Die Kommentare haben eine bestimmte Form einzuhalten, so daß sie durch das Werkzeug doc++ zu HTML-Dateien verarbeitet werden können, und so über HTML-Links durch die dokumentierten Klassen „geblättert" werden kann. Das einzuhaltende Format sind spezielle C++-Kommentare. So kann zu jeder Komponente – Methode oder Attribut – einer Klasse, durch die Zeichenfolge `///` eingeleitet, eine einzeilige Beschreibung der Komponente erfolgen. Ausführlicher Beschreibungen, in denen insbesondere die Bedeutung der Parameter und des Rückgabewertes, sowie Voraussetzungen für die Benutzung einer Methode erläutert werden, beginnen durch die Folge `/**`, sind i.a. mehrzeilig und werden durch `*/` abgeschlossen. Einzeilige Kommentare haben hierbei unmittelbar vor, die mehrzeilige Variante unmittelbar nach der Deklaration der Komponente zu stehen. Zusätzlich wird es ermöglicht durch einen mit `//:` eingeleiteten Kommentar eine neue Gruppierung zu beginnen. In den im Folgenden gezeigten Ausschnitten aus dem Quelltext wird der mehrzeilige Kommentar allerdings weggelassen, da er viel Platz benötigt und (wie bereits erwähnt) in englisch abgefaßt ist.

Für die Implementierung standen die Hardware-Ressourcen der Abteilung Semantik zur Verfügung. Hierbei handelt es sich um ein heterogenes Rechnernetz mit einigen Rechnern der Firma Sun, auf denen das Betriebssystem Solaris genutzt wird, und einigen IBM-kompatiblen PC's, die über das Betriebssystem Linux eingebunden sind. Zur Softwareentwicklung wurde auf beiden Betriebssystemen ein GNU-C/C++-Compiler benutzt, auf den Solaris-Rechner stand zusätzlich der Sun-C/C++-Compiler zur Verfügung.

Im Detail wurden für die Arbeit genutzt:

- IBM-kompatibler PC, AMD K6 mit 200 MHz, 64 MB RAM, Linux Kernel 2.0.36, gcc 2.7.2.1
- 2 IBM-kompatible PC's, Dual Intel Pentium II mit 266 MHz, 128 MB RAM, Linux Kernel 2.0.34, egcs 1.0.3
- Sun Workstation, Dual UltraSPARC-II, 296 MHz, 896 MB RAM, SunOS Release 5.5.1, gcc 2.7.2.3, CC 4.2
- mehrere Sun Workstations, UltraSPARC, 143 – 167 MHz, 256 – 384 MB RAM, SunOS Release 5.5.1, gcc 2.7.2.3, CC 4.2

4.2 Änderungen am Simulator

Zunächst soll auf die Implementierung der Änderungen und Erweiterungen am vorhandenen Simulator eingegangen werden. Diese beziehen sich im wesentlichen auf die Klasse `PLCSimulator`, die die Kernklasse des Simulators darstellt und nun dementsprechend auch eine zentrale Rolle einnimmt.

4.2.1 Zugriff auf Komponenten

Die Umstrukturierung des Simulators soll das Klassengefüge leichter durchschaubar machen und so mögliche Erweiterungen vereinfachen. Aus diesem Grund sollen alle in der Klasse `PLCSimulator` erzeugten Komponenten, auf die von anderen Teilen aus zugegriffen werden muß, durch Methoden verfügbar gemacht werden.

```
//: accessing objects
        /// accessing the design graph of the simulated automata
        MIXDesGraph*      getDesignGraph();

        /// accessing the EventReporter
        PLCEventReporter* getReporter();

        /// accessing the used Scheduler / EventList
        PLCScheduler*     getEventList();

        /// accessing the MessageReceiver
        PLCMsgReceiver*   getMsgReceiver();

        /// return all unconnected input variables
        OrderedCollection& getInterfaceVariables();

        /// the InputObjects (used for polling unconnected inputs)
        PLCInputObjects&  getInputObjects();

        /// accessing the option for the simulation
        PLCSimOptions&    getOptions();
```

Die erste Methode erlaubt den Zugriff auf den *Design-Graph*en. Hierbei handelt es sich um eine Datenstruktur für hierarchische Graphen, die benutzt wird, um das zu simulierende Automatensystem festzuhalten. Die folgenden Methoden erlauben Zugriffe auf die Instanz der Klasse `PLCEventReporter`, die für die Kommunikation mit dem Editor zuständig ist, der Klasse `PLCScheduler`, die Scheduler und Ereignisliste realisiert, und die Instanz der Klasse `PLCMsgReceiver`, die eine Steuerung der Simulation ermöglicht. Design-Graph

Bisher benötigte nur das Steuerungsfenster der Simulation Informationen darüber, welche nichtverbundenen Variablen im Automatensystem existieren, um sie dem Benutzer als Schnittstelle der Umgebung zu präsentieren. Da die Hintergrundsimulation diese Informationen auch benötigt, werden sie nun von der Simulatorklasse ermittelt und für alle Komponenten über die Methode `getInterfaceVariables()` zur Verfügung gestellt.

Das korrekte Verhalten der globalen Inputs wird erreicht, indem für jede Eingangsvariable ein Objekt erzeugt wird, das den von außen anliegenden Wert, den Treiberwert, aufnimmt. Diese Objekte werden in der Klasse `PLCInputObjects` verwaltete und durch die Methode `getInputObjects()` allen Komponenten zur Verfügung gestellt. Ebenso werden die Optionen für die interaktive Simulation in der Klasse `PLCSimOptions` verwaltet und durch die Methode `getOptions()` verfügbar gemacht.

Der Hintergrundsimulator wird durch die Klasse `PLCbgSimulator` realisiert. Um die Grundfunktionalität eines SPS-Simulators zu übernehmen wurde er von der Klasse `PLCSimulator` abgeleitet. Die Liste der global verfügbaren Objekte wird in dieser Klasse erweitert um den Zugriff auf die Protokoll-Übersetzung, deren Kernklasse `PLCProtocolTranslation` darstellt.

```
//: accessing objects
      /// The PLCProtocolTranslation, that controls bgSimulation
      PLCProtocolTranslation* getTranslator();
```

Als eine notwendige Erweiterung am ursprünglichen Simulator müssen Information darüber, wann globale Inputs geändert wurden, übermittelt werden, so daß in der Hintergrundsimulation eine komplette Aufzeichnung eines Simulationslaufs möglich wird. Sowohl interaktiv vom Benutzer, als auch durch Kommunikation über den `PLCMsgReciever` kann der treibende Wert dieser Variablen geändert werden. Da die Protokoll-Übersetzung aber Notiz von solch einer Änderung nehmen muß, wurde die folgende Methode eingefügt.

```
//: communications
      virtual void reportDriverChangedFor (PLCAutVar*);
```

Die Methode erfüllt keinerlei Funktionalität in der interaktiven Simulation. Bei der Hintergrundsimulation wird sie aber überschrieben, so daß die Protokoll-Übersetzung eine Nachricht bekommt. Sie ist virtuell deklariert, damit die Komponenten, die die globalen Inputs beeinflussen, für die Hintergrundsimulation nicht geändert werden müssen. Wird ein `PLCbgSimulator` anstatt eines `PLCSimulator` benutzt, so wird ohne weiteren Eingriff die Protokoll-Übersetzung informiert.

4.2.2 Automatische Wahl der Simulation

Ein Entwurfskriterium war es, eine klare Grenze zwischen der interaktiven Simulation und der Hintergrundsimulation zu erhalten. Aus diesem Grund wurden jegliche Abhängigkeiten der ursprünglichen Klassen von den neuen vermieden. Allerdings soll bei vorhandenen Klassen für Hintergrundsimulation statt einer Instanz der Klasse `PLCSimulator` eine der Klasse `PLCbgSimulator` erzeugt werden. Die MOBY-Klassenbibliothek bietet zu diesem Zweck das dynamische Erzeugen von Klassen anhand ihres Bezeichners über die Methode `Class* Class :: getClassFor (const MString&)` an. Hierbei wird versucht von einer Basisklasse ausgehend eine abgeleitete Klasse mit dem übergebenen Bezeichner zu erzeugen. Bei Erfolg erhält man einen Zeiger auf ein entsprechendes `Class`-Objekt zurück, so daß über die Methode `MObject* Class ::`

`newObj()` eine Instanz der entsprechenden Klasse erzeugt werden kann. Damit dieser Mechanismus greifen kann, muß die Methode `initialObject()` für die gesuchte Klasse implementiert sein. Ist die gesuchte Klasse nicht vorhanden oder nicht von der Basisklasse abgeleitet, so ist der Rückgabewert ein Nullzeiger.

In der Klasse `PLCApplication` wird die benötigte Simulatorinstanz erzeugt. So wird hier zunächst ein `Class`-Objekt für den `PLCSimulator` erzeugt und anhand dieses Objektes versucht den `PLCbgSimulator` durch den Aufruf `getClassFor("PLCbgSimulator")` zu erzeugen. Auf diese Weise ist die Übersetzung der Klasse `PLCApplication` nicht abhängig von der Existenz der Klasse `PLCbgSimulator`, so daß der „normale" Simulator erzeugt werden kann, wenn die Hintergrundsimulation nicht eingebunden wurde. Um diesen Mechanismus nutzen zu können wird demnach für die Klassen `PLCSimulator` und `PLCbgSimulator` die Methode `initialObject()` implementiert, wobei allerdings leider keine Parameter übergeben werden können. Bisher erfolgte eine Parameter-Übergabe von Port und Host, die vom `PLCEventReporter` zum Aufbau der Kommunikationsverbindung mit dem Editor benötigt werden, im Konstruktor des `PLCSimulator`. Dieses wird nun durch eine neue Methode `void PLCEventReporter :: setupPortAndHost (const MString&, const MString&)` bewerkstelligt. Nach dem Erzeugen einer Simulator-Instanz erfolgt die Parameterübergabe durch direkten Zugriff auf den `PLCEventReporter` mittels der Simulator-Methode `getReporter()`.

4.2.3 Treiber für globale Inputs

Die Realisation des korrekten Verhaltens für unverbundene Eingangsvariablen wird durch die Verwendung der Klasse `PLCInputObjects` erreicht. Diese Klasse ordnet jedem globalen Input eine Instanz der Klasse `PLCVarDescription` zu, die in [Sch97] entwickelt wurde, um die aktuelle Belegung und den Typ einer Variablen festzuhalten. Für Zuordnungen dieser Art steht in der Moby-Klassenbibliothek die Klasse `Dictionary` zur Verfügung. Sie erlaubt es zu Schlüsselobjekten – hier vom Typ `PLCAutVar`, also Variablen – Wert-Objekte zu finden bzw. zuzuordnen – hier `PLCVarDescription`. Aus diesem Grund wurde `PLCInputObjects` von `Dictionary` abgeleitet.

Diese Klasse wurde erweitert um Methoden, die den speziellen Anforderungen gerecht werden. Hierbei handelt es sich um Methoden, um die Datenstruktur zu füllen, bzw. zu leeren.

```
//: accessing
        void          init (const Collection&);
        void          empty();
```

Der Methode `init` wird als Parameter eine Kollektion von Variablen übergeben. Diese Menge wird nach Eingangsvariablen durchsucht, um durch Kopieren deren `PLCVarDescription`-Komponente eine Zuordnung von Variable zu anliegendem Wert in die Datenstruktur einfügen zu können. Durch die Methode `empty()` kann dieser dynamisch belegte Speicher wieder frei gegeben werden.

Weitere Methoden erlauben den Test, ob für eine Variable ein Treiber eingetragen ist, und das Setzen bzw. abfragen einer Treiberbelegung.

```
//: testing
        bool          includes(PLCAutVar*);

//: accessing variables
        void          setValue (PLCAutVar*, const TValue&);
        const TValue& getValue (PLCAutVar*);

        PLCVarDescription* getVarDescription (PLCAutVar*);
```

4.2.4 Simulationsoptionen

`PLCSim-Options`

Durch eine Instanz der Klasse `PLCSimOptions` werden im Simulator die Optionen zur Verfügung gestellt. Die Klasse bietet derzeit für zwei Optionen Methoden und Instanzvariablen, und zwar für das Verhalten beim Pollen und für die Zeitauflösung

```
//: input polling
        void setSimultaneousInputPolling (bool);
        bool simultaneousInputPolling () const;
```

Diese Methoden erlauben das Setzen und Abfragen der Poll-Option. Es wird in einer booleschen Variable festgehalten, ob alle Eingangsvariablen eines Automaten zum gleichen Zeitpunkt gepollt werden sollen, also deren Treiberwert quasi simultan[1] ausgelesen werden soll.

Für die Behandlung der Zeitauflösung wurden zum Setzen und Erfragen der aktuellen Einstellung noch Methoden zur Umrechnung hinzugefügt. Die Zeitauflösung wird hierbei als Bruch verwaltet, so daß sich die Simulationszeit aus der Multiplikation der Zeitangaben im Editor mit diesem Bruch ergibt.

[1] Da der Simulator sequentiell arbeitet, können Aktionen nicht nebenläufig durchgeführt werden.

```
//: time resolution
        void setTimeResolution (unsigned int num = 1,
                                unsigned int den = 1);
        void getTimeResolution (unsigned int& num,
                                unsigned int& den);

        PLCSimTime simTimeFor (unsigned int edTime) const;

        double  editorTimeFor (PLCSimTime simTime) const;
        MString editorTimeStrFor (PLCSimTime simTime,
                                  int accuracy = -1) const;
```

Im Allgemeinen werden in der Simulation mehr Zeitschritte benutzt, als im Editor beschrieben wurden, d.h. die Zeitauflösung ist größer als Eins, so daß bei der Rückrechnung von Simulationszeit in Editorzeit ein Simulationsschritt den Bruchteil einer Editorzeiteinheit darstellt. Aus diesem Grund wurde bei einer solchen Konvertierung als Rückgabewert eine Fließkommazahl gewählt. Ebenso wurde eine Konvertierung in Text vorgesehen, bei der die Anzahl der Nachkommastellen angegeben werden kann. Wird hierfür ein negativer Wert benutzt, so wird eine durch die Zeitauflösung festgelegte Anzahl von Nachkommestellen benutzt.

Diese Zahl der Nachkommastellen wird beim Setzen der Zeitauflösung durch die Methode `calculateDecimalAccuracy()` berechnet, bei der durch iteratives erhöhen der Nachkommastellen bis zu einem Maximalwert versucht wird, alle Simulationszeitpunkte in der Darstellung als Editorzeit unterscheidbar zu machen.

```
protected:
//: auxiliury functions
        void calculateDecimalAccuracy();
```

4.3 Ereignisse der SPS-Simulation

In diesem Abschnitt werden Änderungen und Erweiterungen der Ereignisse bei der Implementierung der Hintergrundsimulation beschrieben. Dieses soll in drei Schritten erfolgen. Zunächst wird die Umstrukturierung der vorhandenen Ereignisse zur Vorbereitung der Hintergrundsimulation dargestellt, dann die Umstrukturierung der anderen beteiligten Simulatorkomponenten. Schließlich wird auf die neuen Ereignisse, die zur Umsetzung der Unterbrechungspunkte und der Reproduktion einer Simulation notwendig sind, eingegangen.

4.3.1 Struktur in den Ereignissen

Für die Hintergrundsimulation ist eine Unterteilung des Feuerns in drei Phasen erforderlich. Dieses wird bereits in der Basisklasse `Event` aller Ereignisse implementiert.

```
//: fire event
        virtual bool prepareFire ();
        virtual bool doFire ();
        virtual void reportFire ();
```

Die erste Methode ist für die Umsetzung der Vorbereitungsphase zuständig, die Ausführungsphase wird durch die Methode `doFire()` implementiert und durch die dritte Methode die Mitteilungsphase. Diese drei Methoden sind virtuell deklariert, so daß die Methode `performFire`, die die vorangegangenen Methoden zum Feuern in der ursprünglichen Schnittstelle zusammenfaßt, stets die aktuellen Methoden der abgeleiteten Klassen benutzt, ohne daß sie selbst reimplementiert werden müßte. Der boolesche Rückgabewert der Methode `performFire` besagt, ob die Simulation durch das Feuern dieses Ereignisses gestoppt werden soll. Dieses ist genau dann der Fall, wenn entweder der Rückgabewert der Vorbereitungs- oder der Ausführungsphase `TRUE` ist.

```
        virtual bool performFire ();
```

Entsprechend wurde die ursprüngliche Methode `performFire()` der Klassen `PLCInputEvent` und `PLCStateEvent` aufgeteilt. Das `PLCInputEvent` ermittelt in der Vorbereitungsphase den Treiber für die zu behandelnde Variable und speichert diesen in der Instanzvariable `PLCVarDescription* _driver`, von der in der Ausführungsphase der aktuelle Wert in die Variable übernommen wird. Die Methode `reportFire()` übernimmt die Benachrichtigung des `PLCEventReporter`. Das `PLCStateEvent` ermittelt in der Vorbereitungsphase die aktivierte Transition, die es in der Instanzvariable `PLCSimStateTrans* _nextTransition` festhält. Falls keine Transition aktiviert ist, wird ein Nullzeiger abgelegt. Desweiteren werden die Folgeereignisse wie bisher in der Instanzvariable `SimpleCollection _childEvents` abgelegt.

```
//: event creation
        void                createNewEventsFor (PLCSimAutState*);
        virtual Event*      createEventFor (PLCSimTime time,
                                            MObject* object);
```

Es wurde allerdings eine virtuelle Methode hinzugefügt, die anhand eines Modellobjektes ein Simulationsereignis hierfür erzeugt, so daß für abgeleitete Klassen nur diese Methode überschrieben werden muß, um aktuellere Ereignisse erzeugen zu lassen.

In der Ausführungsphase wird die Aktion der aktivierten Transition und ein Update der Timer durchgeführt. Hierbei wurde die im Entwurf erläuterte Änderung implementiert, so daß jeder Zustand, der Instanz der Klasse `PLCSimAutState` ist, über ein Attribut festhält, wann sein Timer ablaufen wird. Erfolgte ein Zustandswechsel, so wird für alle beteiligten Zustände dieser Wert auf den Zeitwert gesetzt, der sich aus der Addition des (in Simulationszeit umgerechneten) Timerwertes zum aktuellen Simulationszeitpunkt ergibt. Dementsprechend errechnet die Methode `PLCSimTime PLCSimAutState :: getTimeToWait(PLCSimTime simTime)` ihren Rückgabewert aus der Subtraktion des aktuellen Zeitpunktes vom festgehalten Ablaufzeitpunkt.

In der Mitteilungsphase informiert das `PLCStateEvent` wie zuvor den Editor mittels des `PLCEventReporter` über die Variablenänderungen und die benutzte Transition.

4.3.2 Anpassung der Simulatorkomponenten

Wenn man vom Start einer Simulation ausgehend alle Schritte betrachtet, so ist die erste Änderung, die für eine flexiblere Simulation nötig ist, eine Anpassung der initial erzeugten Ereignisse in der Ereignisliste. Aus diesem Grund wird im `PLCScheduler`, analog zur Erweiterung des `PLCStateEvent`, eine virtuelle Methode hinzugefügt, die bestimmt, welches Ereignis für die Behandlung eines bestimmten Objektes in die Ereignisliste eingetragen werden soll.

```
//: auxiliary functions
        virtual Event* initialEventFor (PLCSimTime time,
                                        MObject* object);
```

Für die bisherige Simulation liefert diese Methode ein `PLCStateEvent` für Zustände und für Variablen ein `PLCInputEvent`, so daß das Verhalten gegenüber dem ursprünglichen Scheduler beibehalten wird.

Für die Hintergrundsimulation wird dementsprechend eine Instanz der Klasse `PLCbgScheduler` benutzt, die von `PLCScheduler` abgeleitete Klasse abgeleitet wurde, wobei genau zwei virtuelle Methoden überschrieben werden. `PLCbg-Scheduler`

```
public:
//: accessing
        virtual void   init ();

protected:
//: auxiliary functions
        virtual Event* initialEventFor (PLCSimTime time,
                                        MObject* object);
```

Die Methode `initialEventFor` liefert hier die neuen Ereignisse für den übergebenen Simulationszeitpunkt zurück, `PLCPollEvent` für Variablen und `PLCCycleEvent` für Zustände – beide Ereignisse werden weiter unten beschrieben. Somit kann die Ereignisliste mit den neuen Ereignissen für Hintergrundsimulation initialisiert werden. Die Initialisierung des Schedulers wird trotzdem auch überschrieben, da bei einem Neustart der Simulation durch den „Restart"-Knopf im Kontrollfenster derzeit nur der Scheduler über eben diese Aktion benachrichtigt wird. Um in der Klasse des Kontrollfensters Abhängigkeiten zur Hintergrundsimulation zu vermeiden kann hier keine Benachrichtigung der Protokoll-Übersetzung erfolgen, so daß dieses vom `PLCbgScheduler` in seiner `init`-Methode vorgenommen wird, nachdem die geerbte Methode zur Initialisierung aufgerufen wurde. Auf längere Sicht wäre eine Lösung, bei der die Simulatorklasse solche Ereignisse weitergibt, zu empfehlen.

4.3.3 Neue Ereignisse

Wie durch das Entwurfskapitel bereits bekannt und im vorangegangenen Abschnitt auch angedeutet, sind in der Hintergrundsimulation für das Pollen und das Zyklusende neue Ereignisse entstanden.

`PLCPollEvent`

Die Klasse `PLCPollEvent` wurde von `PLCInputEvent` abgeleitet und ersetzt dieses bei Verwendung des `PLCbgSimulator`. Die Änderung gegenüber dem Input-Event betrifft die Methode `reportFire`, in der lediglich zusätzlich zum `PLCEventReporter` die Protokoll-Übersetzung benachrichtigt wird.

`PLCCycle-Event`

Die Änderungen am Ereignis für das Zyklusende, `PLCCycleEvent`, dessen Basisklasse `PLCStateEvent` ist, sind gravierender. In der Vorbereitungsphase muß das Ereignis die Protokoll-Übersetzung befragen, ob Folgeereignisse durch das Protokoll vorgegeben werden, um so das Zeitverhalten nachbilden zu können. Desweiteren kann es sein, daß aus dem Protokoll auch Ereignisse übernommen werden müssen, die nicht das Zeitverhalten betreffen. Durch die Verwendung der verteilten Trace-Interpretation können derzeit keine Ereignisse aus dem Protokoll direkt in die Ereignisliste eingefügt werden, so daß z.B. auch Änderungen eines globalen Inputs als Folgeereignisse eines Cycle-Events behandelt werden. Soll oder kann das Zeitverhalten nicht aus dem Protokoll übernommen werden, so werden nur genau diese anderen Ereignisse[2] in die Liste der Folgeereignisse eingefügt. Die Poll-Events und das Cycle-Event werden in diesem Fall durch die geerbte Methode neu erzeugt. Hierbei kommt zum Tragen, daß auf die virtuelle Methode `createEventFor` zurückgegriffen wird, so daß statt Input- und State-Event, auch tatsächlich Poll- und Cycle-Event erzeugt werden.

Bei der Übernahme von Poll- und Cycle-Events muß dann noch überprüft werden, ob alle notwendigen Ereignisse tatsächlich dem Protokoll entnommen

[2] Ereignisse, die nicht das Zeitverhalten festlegen

werden konnten. Dieses ist dann notwendig, wenn das gelesene Protokoll vor dem Ende des aktuellen SPS-Zyklus abbricht, so daß hier nur noch fehlende Ereignisse ergänzt werden, und zwar so, daß sie zeitlich nach dem Ende der Aufzeichnung eingeordnet werden.

Zur Vorbereitung auf nichtdeterministische SPS-Automaten wird auch erfragt ob die zu wählende Transition durch das Protokoll bestimmt wird. Eine solche Erweiterung der SPS-Automaten würde nämlich zulassen, daß mehrere Transitionen gleichzeitig aktiviert sind und so eine (gewichtete) zufällige Auswahl getroffen werden kann. Auch hier wird wieder das Prinzip verfolgt, daß die bisherige Methode zur Bestimmung der Transition benutzt wird, wenn diese nicht von der Protokoll-Übersetzung übermittelt wird.

Die Ausführungsphase des `PLCCycleEvent` bleibt hingegen unverändert und in der Mitteilungsphase wird wie schon beim `PLCPollEvent` lediglich die Benachrichtigung der Protokoll-Übersetzung hinzugenommen.

Die Einführung weiterer Ereignisse ergibt sich aus den Anforderungen an die Hintergrundsimulation. Zum einen ist es erforderlich aufgezeichnete Treiberänderungen bei einer wiederholten Simulation erneut durchzuführen. Dieses wird durch ein `PLCDriverEvent` ausgelöst wobei durch das Ableiten von `PLCInputEvent` das Auffinden des Treibers in der Vorbereitungsphase geerbt wird. Die Zuweisung eines Wertes an das treibende Objekt[3] erfolgt in der Ausführungsphase, wobei ein Ausdruck ausgewertet wird. Solch ein Ausdruck liegt in Form einer `PLCTExpression` vor – das Präfix ergibt sich aus **PLC** und **T**race. Solch ein Ausdruck wird vom Trace-Compiler aufgebaut und wird im Abschnitt 4.4.1 erläutert. Wichtig ist an dieser Stelle zunächst nur, daß der Ausdruck zum Zeitpunkt, zu dem das Ereignis feuert, ausgewertet wird und damit die dann aktuellen Variablenbelegungen von Bedeutung sind. Die Mitteilungsphase ruft wiederum die Methode `reportDriverChangedFor` des Simulators auf, so daß sich das Ereignis genauso verhält, als sei eine Benutzereingabe erfolgt.

`PLCDriver-Event`

Das `PLCAssignEvent` wird ebenfalls von der Klasse `PLCInputEvent` abgeleitet. Es sorgt dafür, daß Variablenzuweisungen, die im Protokoll aufgezeichnet sind, wieder in die Simulation aufgenommen werden können. Wie bereits beim Entwurf erläutert, sollten solche Ereignisse, um konsistent mit der Semantik zu bleiben, eigentlich nicht erzeugt werden – aus den genannten Gründen wurden sie aber dennoch implementiert. Solch ein Ereignis hat keine Vorbereitungsphase – d.h. die Methode `prepareFire` liefert lediglich den Wert `FALSE` als Rückgabe, so daß kein Halt des Simulators ausgelöst wird. In der Ausführungsphase wird hier wie beim `PLCDriverEvent` ein Ausdruck ausgewertet, jedoch wird das Ergebnis direkt in die zu behandelnde Variable

`PLCAssign-Event`

[3] Das treibende Objekt ist stehts Instanz der Klasse `PLCVarDescription`, die ihrerseits Komponente der `PLCInputObjects` oder einer Ausgangsvariable ist.

geschrieben. Dementsprechend verhält sich das Ereignis in der Mitteilungsphase so, als sei durch eine Transition die Variable geändert worden.

PLCBreak-Event

Das schließlich letzte neue Ereignis ist ein `PLCBreakEvent`, das direkt von der Basisklasse `PLCEvent` abgeleitet ist. Diese Ereignis zeichnet sich für das Auslösen eines Unterbrechungspunktes verantwortlich. Auch die Überprüfung des Systemzustandes wird durch solch ein Ereignis ausgelöst, da dieses nach den Überlegungen in Abschnitt 3.6.2 ein Spezialfall von Unterbrechungspunkten ist. Die Implementierung des `PLCBreakEvent` wird aber erst im Abschnitt 4.5.3 erläutert, da sie, ohne die Umsetzung der Unterbrechungspunkte zu kennen, nicht nachvollziehbar ist.

4.4 Der Trace-Compiler

PLCTrace-Compiler

Der Trace-Compiler spielt eine zentrale Rolle bei der Reproduktion einer Simulation. Er wird durch die Klasse `PLCTraceCompiler` implementiert und erzeugt aus einer Protokoll-Datei die interne Darstellung eines Simulationsprotokolls. Diese interne Darstellung besteht aus zwei Teilen, der Trace als Instanz der Klasse `PLCSimTrace` sowie einem Teil der Trace- und Simulationsoptionen.

4.4.1 Ausdrücke in der Trace

PLCT-Expression

Um komplexere Ausdrücke im Eingabeprotokoll und damit auch in der internen Darstellung einer Trace zu ermöglichen, wurde eine Datenstruktur entwickelt, die solche beliebig komplexen Ausdrücke als Syntaxbaum speichert. Die Klasse `PLCTExpression` realisiert diesen interpretierbaren Syntaxbaum auf effiziente Weise, indem jede Instanz der Klasse einen Knoten im Syntaxbaum eines Ausdrucks repräsentiert. Bei den Knoten kann es sich um ein- und zweistellige Operatoren handeln, wie sie bereits im Editor für Ausdrücke erlaubt werden, sowie um nullstellige Knoten, die demnach Variablen bzw. Konstanten repräsentieren. Der Typ einer `PLCTExpression` wird durch Parameter des Konstruktors festgelegt.

```
PLCTExpression (Type0, TValue*);
PLCTExpression (Type0, PLCAutVar*);
PLCTExpression (Type1, PLCTExpression*);
PLCTExpression (Type2, PLCTExpression*, PLCTExpression*);
PLCTExpression (const PLCTExpression&);
```

Typen eines Knotens sind in die drei zuvor genannten Kategorien, null-, ein- und zweistellige Operatoren, unterteilt und benötigen dementsprechend weitere Parameter. So erhalten nullstellige Operatoren den konstanten Wert bzw. die Variable als Parameter. Ein einstelliger Operator muß seinen Typ und seinen Parameter kennen, der wiederum Knoten eines Syntaxbaumes, also eine `PLCTExpression`, ist. Ein zweistelliger Operator hat dementsprechend zwei solche Parameter.

```
        enum Type0 {notType0,
                    boolConst, boolVar, intConst, intVar,
                    enumConst, enumVar};
        enum Type1 {notType1, notOp, negativOp};
        enum Type2 {notType2, andOp, orOp, equalOp, not_equalOp,
                    lessOp, less_equalOp, greaterOp,
                    greater_equalOp, minusOp, plusOp,
                    moduloOp, divOp, multOp};
```

Ein Syntaxbaum muß nach dieser Definition Bottom-up aufgebaut werden, d.h. es sind zunächst die Blätter zu erzeugen, die dann den Operatorknoten als Konstruktorparameter übergeben werden und diese können wiederum weiteren Operatoren als Parameter dienen. Anhand der Grammatik, dessen Sprache der Parser des Trace-Compilers erkennt, soll beispielhaft gezeigt werden, wie der Aufbau eines Syntaxbaumes erfolgt. Der Compiler-Compiler bison benutzt eine attributierte Grammatik, wobei jedem Terminal- und Nichtterminalsymbol ein typisiertes Attribut zugeordnet ist. Dieses Attribut hat den Typ `PLCTExpression*` bei den Regeln zur Übersetzung von Ausdrücken. In semantischen Aktionen, die als C++-Befehle in geschweiften Klammern wie Nichtterminale in die Grammatik geschrieben werden, kann auf diese Attribute zugegriffen werden. Hierbei steht der Ausdruck `$$` für das resultierende Attribut und `$`i für das Attribut der i-ten Komponente der Regel. Weitere Informationen zum Aufbau einer bison-Grammatik liefert die Online-Hilfe (aufzurufen durch: `info bison`, bzw. jedes andere Programm zur Darstellung von GNU-info Seiten)

```
intVarExpr
        : FullIntVariableName
          { $$ = new PLCTExpression(PLCTExpression::intVar,
                                     (PLCAutVar*)$1->reference());
          }
        | NUMBER
          { $$ = new PLCTExpression(PLCTExpression::intConst,
                                     new TValue(
                                       (ValueType)$1->intVal()) );
          }
        ;
```

Diese Regeln sind ein Beispiel für das Erzeugen von Blättern des Syntaxbaumes. Die erste Regel entdeckt eine Variable im Eingabestrom, so daß ein nullstelliger Operator vom Typ `intVar` erzeugt wird. Ein solcher Knoten erhält als weiteren Parameter eine Referenz auf die entsprechende Variable. Analog wird beim Erkennen einer Zahl ein Knoten mit einer Integer-Konstante erzeugt.

Durch das benutzen des Compiler-Compilers können diese Blätter des Syntaxbaumes mit geringestem Aufwand durch Operatoren kombiniert werden. Da die Attribute der beteiligten Nichtterminale bereits eine Instanz der Klasse `PLCTExpression` liefern, können diese direkt dem neu erzeugten Operator-Knoten als Konstruktorparameter übergeben werden, was hier am Beispiel für Additions- und Subtraktionsoperator gezeigt wird. Hierbei wird deutlich, daß der Parsergenerator bereits dafür sorgt, den Syntaxbaum Bottum-up aufzubauen.

```
intExpr
        : intTerm
        | intExpr PLUS intTerm
            { $$ = new PLCTExpression(PLCTExpression::plusOp,
                                      $1, $3); }
        | intExpr MINUS intTerm
            { $$ =new PLCTExpression(PLCTExpression::minusOp,
                                      $1, $3); }
        ;
```

Da diese Ausdrücke häufig ausgewertet werden müssen, ist eine möglichtst hohe Effektivität anzustreben, so daß Optimierungen stattfinden sollten. Die Reduktion von Ausdrücken mit Konstanten wird bereits in den Konstruktoren der Klasse `PLCTExpression` vorgenommen. So wird z.B. beim Erzeugen eines Additionsoperators mit Knoten für die Konstanten `2` und `3`, hieraus der ausgewertet Ausdruck generiert, also ein Knoten für die Konstante `5`.

Um sich unnötiges Auswerten eines Ausdrucks ersparen zu können, erlaubt `PLCTExpression` auch das Durchsuchen des Syntaxbaumes nach dem Vorkommen von Variablen. Die Methode `getVariables` füllt die als Parameter übergebene Menge mit den Variablen, auf die im Ausdruck zugegriffen wird. Durch die Benutzung einer Instanz der Klasse `Set` wird hierbei automatisch dafür gesorgt, daß jede Variable nur einmal enthalten ist. Bei Operatoren erfolgt dieses Ermitteln verwendeter Variablen, indem diese Methode für die Parameter aufgerufen wird, bei einem Variablenknoten wird entsprechend lediglich die Variable der Menge hinzugefügt.

Die hier beschriebenen Ausdrücke werden zum einen benutzt, um Werte für Treibereinplanungen und direkte Zuweisungen in der Trace benutzen zu können, zum anderen auch, um die Bedingungen von Unterbrechungspunkten anzugeben. Um diese Unterbrechungspunkte auch in einem eigenen Editor bearbeiten zu können, müssen die Ausdrücke auch eigenständig übersetzt werden können. Der Compiler stellt zu diesem Zweck einen eigenen Dienst bereit.

```
        PLCTExpression*          compileExpr(const MString&);
        PLCTExpression*          compileBoolExpr(const MString&);
```

Hierbei wird der übergebene Text[4] in einen Syntaxbaum übersetzt, wobei die zweite Methode nur boolesche Ausdrücke zuläßt und sonst einen Übersetzungsfehler erzeugt. Falls hierbei Fehler auftreten, wird ein Nullzeiger zurückgeliefert und die Fehler bei der Übersetzung können über die Methode `getAllErrors()` erfragt werden.

4.4.2 Übersetzung einer Protokolldatei

Das Prinzip für die Übersetzung eines Ausdrucks wird im wesentlichen auch für die Übersetzung eines ganzen Protokolls benutzt. Die Regeln der Grammatik werden attributiert, so daß nur wenige Informationen in Instanzvariablen des Compilers festgehalten werden müssen.

```
//: instance variables
  //results
        PLCSimTrace*             _theTrace;
        OrderedCollection        _breakpoints;
        PLCSimOptions*           _simOptions;
```

Diese Instanzvariablen werden vom Compiler durch das Lesen der Protokolldatei belegt. Für jeden korrekt erkannten Unterbrechungspunkt wird ein `PLCBreakpoint`-Object der Menge `_breakpoints` hinzugefügt. Beim erkennen einer Simulationsoption wird überprüft, ob ein Objekt für `_simOptions` bereits existiert, dieses ggf. erzeugt und die erkannte Option dort eingetragen. Schließlich wird bei jeder korrekten Übersetzung ein Objekt der Klasse `PLCSimTrace` erzeugt. Das Lesen der textuellen Repräsentationen von Trace-Einträgen erzeugt Objekte der von `PLCTraceEntry` abgeleiteten Klassen. `PLCTrace-Entry`

Das Übersetzen eines Protokolls wird durch die `compile()`-Methode der Protokoll-Übersetzung angestoßen und liefert bei syntaktisch korrekter Eingabe als Ergebnis einen Zeiger auf die gelesene Trace des Protokolls. Bei einem Fehlversuch wird ein Nullzeiger zurückgeliefert, worauf wie bei der Übersetzung eines einzelnen Ausdrucks mit der Methode `getAllErrors()` die gefundenen Fehler abgefragt werden können. Um ein Protokoll übersetzen zu können, muß der Trace-Compiler Variablen- und Automatennamen kennen. Aus diesem Grund erhält er als Konstruktorparameter einen Zeiger auf den

[4] Zur Behandlung von Texten stellt die MOBY-Klassenbibliothek die Klasse `MSting` bereit.

`PLCSimulator`, und somit Zugriff auf den Design-Graph des zu simulierenden Automatensystems.

Im Fall einer erfolgreichen Compilation liefert ein Aufruf der Methode `getBreakpoints()` eine Liste der im Protokoll definierten Unterbrechungspunkte. Ein Aufruf von `getOptions()` liefert einen Zeiger auf ein Objekt der Klasse `PLCSimOptions`, sofern Simulationsoptionen im Protokoll definiert wurden, im anderen Fall einen Nullzeiger – konsistent zu den anderen Zugriffsmethoden.

4.5 Die Protokoll-Übersetzung

`PLCProtocol-Translation`

Die Protokoll-Übersetzung, die die Kernkomponente der Hintergrundsimulation darstellt, wird in der Klasse `PLCProtocolTranslation` realisiert. Sie stellt das Bindeglied zwischen Simulator und Protokolldatei dar, so daß sie eine ähnlich zentrale Rolle einnimmt, wie die Simulator-Klasse.

4.5.1 Die Schnittstelle zum Simulator

Durch eine ganze Reihe von Methoden erlaubt die Protokoll-Übersetzung dem Simulator das Mitteilen von Ereignissen. Diese werden hauptsächlich von den Ereignissen in Anspruch genommen, um ihr Feuern mitzuteilen. Die Methode `driverChanged` hingegen wird vom `PLCbgSimulator` in der Methode `reportDriverChangedFor` benutzt, um Änderungen an treibenden Objekten der Protokoll-Übersetzung mitzuteilen.

```
//:simulator interface
        void              cycleEndedFor ( PLCSimAutGraph* event,
                                          PLCSimTime time);
        void              variablesChanged ( PLCSimStateTrans*,
                                             PLCSimTime time);
        void              variableChanged ( PLCAutVar*,
                                            PLCSimTime time);
        void              inputPolledFor ( PLCAutVar*,
                                           PLCSimTime time);
        void              driverChanged ( PLCAutVar*,
                                          PLCSimTime time);
```

`PLCTrace-Options`

Die Methoden erfragen allesamt die Trace-Optionen, die durch eine Instanz der Klasse `PLCTraceOptions` in der Protokoll-Übersetzung festgehalten werden und durch die Methode `getTraceOptions()` auch von außen zugreifbar sind, ob eine Aufzeichnung der Simulation überhaupt erfolgt, und falls ja, ob die aktuelle Information in der Trace des Simulationsprotokolls überhaupt erwünscht ist. Letztere wird durch eine Instanz der Klasse `PLCSimTrace` im Attribut `_outTrace` aufgebaut.

Ebenso wie das Berichten des Feuerns liegt es in der Verantwortung des `PLCCycleEvent`, sich nötige Informationen aus dem Protokoll zu erfragen. Hierzu gibt es jeweils eine Methode, die Auskunft darüber gibt, ob eine Information durch die Trace vorgegeben wird und eine um ggf. diese zu erhalten.

```
//: controlling simulation (playback)
        bool              determinesTransFor(PLCSimAutState*);

        PLCSimStateTrans* getTransFor ( PLCSimTime actualTime,
                                        PLCSimAutState*,
                                        bool& stop);
```

Über diese Methoden kann ein Cycle-Event feststellen, ob eine Transition durch das Eingabeprotokoll vorgegeben wird. Da es sich hier nur um eine Vorbereitung für die Erweiterung des Simulators handelt liefert die Methode derzeit noch stets den Wert `FALSE`, bzw. einen Nullzeiger zurück, d.h. diese Informationen werden nicht der Trace entnommen.

```
        bool              createsEventsFor( PLCSimTime actualTime,
                                            PLCSimAutState*);

        bool              getEventsFor ( PLCSimTime actualTime,
                                         PLCSimAutState*,
                                         Collection& childEvents);
        bool              getNonTimingEventsFor
                                       ( PLCSimTime actualTime,
                                         PLCSimAutState*,
                                         Collection& childEvents);
```

Ein Cycle-Event erfragt, ob Poll- und Cycle-Events für den nächsten Zyklus durch das Eingabeprotokoll vorgegeben werden. Auch diese Methode konsultiert zunächst die Trace-Optionen, denn wenn diese festlegen, daß bestimmte Informationen nicht der Trace entnommen werden sollen, so müssen sie auch nicht darin gesucht werden. Die beiden `get`-Methoden liefern entsprechend Folgeereignisse, mit bzw. ohne die Ereignisse für das Zeitverhalten. Aber auch bei den anderen Ereignissen wird vor deren Erzeugung aus den Trace-Einträgen überprüft, ob sie gemäß der Trace-Optionen überhaupt übernommen werden dürfen.

4.5.2 Details der Trace-Optionen

PLCTrace-Options

Bei der Klasse PLCTraceOptions handelt es sich um ein Verzeichnis für alle Informationen, die bei der Hintergrundsimulation von Belang sind. Diesem Verzeichnischarakter entsprechend wurde sie auch von der Klasse Dictionary abgeleitet und um weitere Komponenten angereichert. Der Dictionary-Teil speichert zu jedem Automaten[5] seine speziellen Optionen in einer Instanz der Klasse PLCSimAutInfo auf die durch die folgenden Methoden der Zugriff erleichtert wird.

PLCSimAut-Info

```
//: accessing
        PLCSimAutInfo* newInfoFor ( PLCSimAutGraph* );
        bool           insertInfoFor ( PLCSimAutGraph*,
                                       PLCSimAutInfo* );
        PLCSimAutInfo* getInfoFor ( PLCSimAutGraph* );
```

Die allgemeineren Optionen werden in Instanzvariablen dieser Klasse festgehalten, wobei jeweils über weitere Methoden Zugriff hierauf gewährt wird.

```
//: instance variables
        bool _useInputTrace;
        bool _createOutputTrace;

        Filename _inputFile;
        Filename _outputFile;

        OrderedCollection _breakpoints;
```

Ähnlich hierzu verfügt die Klasse PLCSimAutInfo über Zugriffsmethoden, die die einzelnen Optionen für einen Automaten festlegen. So kann hierin festgelegt werden, ob das Zeitverhalten gelesen werden soll, und für jede Variable, ob Treiberänderungen sowie direkte Wertänderungen aus der Trace gelesen werden sollen. Die gleiche Auswahl steht für die Aufzeichnung einer Simulation zur Verfügung.

Außerdem ist diese Klasse für die verteilte Interpretation der Eingabetrace verantwortlich. Für die Interpretation einer Trace wird die Information benötigt, wie weit die Trace bereits gelesen wurde. Dieses wird durch die Klasse PLCTraceContext gelöst, die als Stack implementiert ist, um auch geschachtelte Traces behandeln zu können. Diese Möglichkeit wird bei höheren Sprachkonstrukten, wie Schleifen oder bedingten Trace-Einträgen besonders interessant und kann demzufolge derzeit noch nicht ausgenutzt werden.

Der PLCTraceContext legt zur Interpretation auf einem Stack alle „betretenen“ Trace-Einträge ab, so daß für jede Schachtelungstiefe und den aktu-

[5] Demnach sind die zu simulierenden Automaten die Schlüssel im Dictionary.

ellen Trace-Eintrag, der nicht weiter verschachtelt ist, ein Objekt auf diesen Stack gelegt wird. Wird nun der nächste Trace-Eintrag über `nextEntry` angefordert, so findet die Methode als oberstes Element des Stack den zuletzt gelesenen Eintrag und entfernt ihn. Der dann obere Eintrag ist die Instanz, in der der Folgeeintrag zu dem gerade entfernten Objekt gesucht werden muß. Gibt es keinen Nachfolger, so muß auch dieser Eintag wieder als aktuell gelesener Eintrag vom Stack genommen werden und die Suche eine Ebene tiefer fortgesetzt werden. Trifft dieses auf die unterste Ebene zu, so wurde die Trace bis zu ihrem Ende gelesen. Wurde hingegen ein Folgeeintrag gefunden, so muß überprüft werden, ob es sich um einen Eintrag handelt der wiederum Einträge enthält. Ist dieses der Fall, so wird der gefundene Eintrag auf den Stack gelegt und mit dessen ersten Element die Suche fortgesetzt.

4.5.3 Unterbrechungspunkte

`PLC-Breakpoint`

Ein `PLCBreakpoint` im Hintergrundsimulator besteht aus vier Komponenten. Zunächst muß eine Bedingung angegeben werden, die den Unterbrechungspunkt auslösen soll. Für diese Bedingung wird auf die bereits erwähnten Ausdrücke, die in Abschnitt 4.4.1 erklärt wurden, zurückgegriffen. Desweiteren soll optional eine Zeitspanne angegeben werden können, so daß die Bedingung über einen solchen Zeitraum erfüllt sein muß. Hierzu reicht ein Attribut vom Typ `PLCSimTime`, denn das Weglassen der Option ist dem Setzen dieses Parameters auf den Wert 0 gleich.

Ebenso wird mit der Nachricht verfahren, die ein Unterbrechungspunkt auslösen kann. Handelt es sich bei der Instanz des `MString` um eine leere Zeichenkette, so wird keine Nachricht erzeugt. Schließlich wird noch eine Aktion angegeben, deren mögliche Ausprägungen durch einen Aufzählungstyp angegeben werden.

```
/// create a breakpoint and set all information
PLCBreakpoint (PLCTExpression* expression,
               PLCSimTime      holdTime,
               const MString&  message,
               BreakAction     action);
```

Die Aufgabe der Protokoll-Übersetzung ist es nun zu überprüfen, wann ein Unterbrechungspunkt aktiviert ist, wobei weitere Methoden und Attribute des `PLCBreakpoint` Unterstützung leisten. Bei jeder Variablenänderung testet die Protokoll-Übersetzung, ob die geänderte Variable in einer Bedingung der Unterbrechungspunkte benutzt wird. Dieses kann durch einen Test, ob die Variable in der Menge der benutzten Variablen eines Ausdrucks enthalten ist, erreicht werden. Die Instanzen der Klasse `PLCTExpression` können diese Informationen bekanntlich über die Methode `getVariables` zur Verfügung stellen, so daß die Protokoll-Übersetzung über die Methode `bool PLCBreakpoint :: dependsOn (PLCAutVar*)` für jeden Unterbrechungspunkt feststellen kann, ob er von der geänderten Variable abhängt.

Wenn ein Unterbrechungspunkt nun von einer Variablen abhängt, so muß seine Bedingung überprüft werden. Die Methode `evaluate()` führt diesen Test durch, wobei sie die aktuelle Simulationszeit als Parameter erhält, damit im `PLCBreakpoint` die Zeit der letzten Wertänderung der Bedingung festgehalten werden kann. Wurde die Bedingung zu `TRUE` ausgewertet, dann ist der Unterbrechungspunkt möglicherweise nach verstreichen der Verzögerungszeit aktiviert. Deshalb wird dieser Zeitpunkt der möglichen Aktivierung mit der Methode `activateTime(PLCSimTime actualTime)` erfragt und für den erhaltenen Zeitpunkt ein `PLCBreakEvent` in die Ereignisliste eingetragen.

`PLCBreak-Event`

Die Aufgabe der Klasse `PLCBreakEvent` besteht nun darin zu prüfen, ob der Unterbrechungspunkt zum Zeitpunkt des Ereignisses noch aktiviert ist. Deshalb erhält eine Instanz dieser Klasse als Parameter einen Zeiger auf den zu überprüfenden `PLCBreakpoint`. In der Vorbereitungsphase dieses Ereignisses wird durch den Aufruf der Methode `bool PLCBreakpoint :: isActive (PLCSimTime actualTime)` der Unterbrechungspunkt befragt, ob er noch aktiviert ist. Letzterer stellt dieses dadurch fest, daß seine Bedingung derzeit erfüllt ist und sich seit der geforderten Zeitspanne auch nicht mehr geändert hat.

In der Ausführungsphase des `PLCBreakEvent` wird nun bei noch aktiviertem Unterbrechungspunkt die hiermit verbundene Aktion ausgelöst. In der Mitteilungsphase wird, wie üblich, die Protokoll-Übersetzung über das Feuern des Ereignisses informiert. Diese präsentiert für ein `PLCBreakEvent` die Nachricht des Unterbrechungspunktes und hält das Auslösen der Unterbrechung im Protokoll fest, indem ein Objekt der Klasse `PLCBreakEntry` in die Trace eingetragen wird.

Das Überprüfen von Zusicherungen in der Trace, die durch ein Objekt der Klasse `PLCCheckEntry` im internen Protokoll repräsentiert werden, wurde im Entwurfskapitel als Spezialfall der Unterbrechungspunkte beschrieben. Deshalb werden auch `PLCBreakpoint`-Objekte benutzt, um die Überprüfung der Zusicherungen vorzunehmen. Eine Zusicherung in der Trace einer Protokolldatei hat die folgende Form:

```
AWAIT <Bedingung> ELSE ( REPORT <String> <Aktion>
                       | REPORT <String> | <Aktion> ) ;
```

Eine solche Zusicherung soll garantieren, das zu dem Zeitpunkt, für den der Eintrag in der Trace steht, die Bedingung erfüllt ist. Falls dem nicht so ist, soll die Nachricht ausgegeben und die Aktion ausgeführt werden. Dieses entspricht einem Unterbrechungspunkt mit negierter Bedingung, dessen einzuhaltende Zeitspanne 0 ist und der die gleiche Nachricht und Aktion erhält.

Vom Trace-Compiler wird beim Erkennen eines `PLCCheckEntry` genau solch ein `PLCBreakpoint` erzeugt und der internen Repräsentation als Parameter übergeben. Solch ein Unterbrechungspunkt darf nicht in die Liste der Unterbrechungspunkt eingetragen werden, da er sonst ständig überprüft würde, dieses jedoch nur zu dem eingetragenen Zeitpunkt erfolgen soll. Deshalb wird er weiterhin als Komponente eines Trace-Eintrags, der Zusicherung, verwaltet.

Bei der Übersetzung des `PLCCheckEntry` in ein Ereignis der Simulation wird nun ein `PLCBreakEvent` erzeugt, das als zu behandelnden Unterbrechungspunkt den des `PLCCheckEntry` erhält. Dieses Ereignis verhält sich wie bereits oben beschrieben, wobei hier der zu behandelnde Unterbrechungspunkt genau dann aktiviert ist, wenn der erwartete Systemzustand, der in der Bedingung der Zusicherung beschrieben wird, nicht angenommen wurde.

4.6 Die Visualisierungskomponente

Wie bereits im Entwurf erwähnt, wurden für die Visualisierungskomponente die Klassen der Projektgruppe IDEFIX als Basis benutzt, um sie in angepaßter Form im Hintergrundsimulator benutzen zu können.

4.6.1 Umsetzung der IDEFIX-Klassen

Die IDEFIX-Implementierung verwendet Klassen des Thor-Netz-Editors zur Verwaltung der verschiedenen Diagramme. Da solch eine Anbindung im Simulator nicht erwünscht ist, wurde eine neue Verwaltungskomponente geschaffen, der `GDiagramManager`. Dieser stellt einen Dialog zur Verfügung, indem das darzustellende Protokoll gewählt und dann Diagramme geöffnet werden können.

`GDiagram-Manager`

```
//: view
        void     createSystemView();
        void     open ();
        void     close ();
```

Bevor Simulationsdaten in Diagrammen dargestellt werden können, muß ein Protokoll eingelesen werden. Die Aktivierung des entsprechenden Knopfes löst die Funktion `selectTrace` aus, die einen Dateiauswahldialog öffnet und anschließend die Übersetzung der Protokolldatei in die benötigte Datenstruktur vornimmt. Dieses erfolgt in zwei Schritten.

```
//: simulation data
        void compile (PLCSimTrace*);
        bool compile (const Filename&);
```

Die erste Methode dient dazu, aus einer Trace die benötigten Daten in Form von Instanzen der Klasse `GExpressionData` zu erzeugen. Sie wird von der zweiten Methode aufgerufen, nachdem diese mit dem Trace-Compiler aus der Protokolldatei eine `PLCSimTrace` erzeugen konnte. Die Methode `getData` kann dann von den Diagrammen benutzt werden, um festzustellen, welche Daten zur Visualisierung zur Verfügung stehen.

```
        Collection& getData();
```

`PLCT-Translation`

Das Erzeugen der Daten erfolgt durch die Klasse `PLCTTranslation`. Diese geht eine Trace Schritt für Schritt durch und sucht nach Daten, die zur Visualisierung geeignet sind. Dieses sind zur Zeit Driver- und Assign-Entries. Dabei wird für jede Variable ein Objekt der Klasse `GExpressionData` erzeugt und hierin für ein entsprechendes Ereignis eine sogenannte Koordinate, also ein Paar aus Simulationszeit und Variablen-, bzw. Treiberwert eingetragen.

Die Klasse `GScaledObject` bereitet diese Daten noch für die Diagramme auf, indem es z.B. festhält, was Maximal- und Minimalwert sowie die maximale Simulationszeit sind. Sie ist zudem für die Darstellung der Daten in einem Diagramm verantwortlich, also für den Teil der Darstellung, der durch die behandelten Daten erzeugt wird. Dazu verfügt sie über eine Methode für jeden Diagrammtyp, die von der entsprechenden Klasse aufrufen wird. Diese Methoden wurden bis auf eine Typanpassung unverändert aus der IDEFIX-Implementierung übernommen.

```
//: diagrams
        void newGanttDiagram ();
        void newCurveDiagram ();
```

Nach dem erfolgreichen Lesen der Daten können über diese Methoden Diagramme des entsprechenden Typs erzeugt werden. Alle Diagramme werden hierbei in eine Liste eingetragen, die durch die Instanzvariable `_diagrams` als Object der Klasse `SimpleCollection` realisiert wird, so daß beim Schließen des Diagramm-Managers auch alle Diagramme geschlossene werden.

Model-View-Controller

In der MOBY-Klassenbibliothek wurde das *Model-View-Controller* Konzept aus der Smalltalk-Bibliothek übernommen (d.h. durch C++ Klassen nachgebildet), so daß zwischen Daten (Model), Darstellungskomponenten (View), und Bedienelementen (Controller), unterschieden wird. Dieses Konzept erlaubt z.B. die Darstellung von nur einem Modell in mehreren Fenstern (Views), oder die Benutzung unterschiedlicher Controller für eine Fensterklasse.

Die eigentlichen Diagramme mit der Basisklasse `GDiagram`, die bis auf die Umstellung von Thor-Netz auf die Verwaltungsinstanz `GDiagramManager` übernommen werden konnten, sind von der Klasse `Graph` abgeleitet. Entsprechend des Model-View-Controller Konzeptes gibt es hierzu die Klassen

`GDiagramView` und `GDiagramController`. Für Kurvendiagramme der Klasse `GCurve` wurden entsprechend die Klassen `GCurveView` und `GCurveController` übernommen. Die Anpassung des Controllers beschränkte sich hierbei auf die Übersetzung des Klassennamens. Das View konnte jedoch nicht so einfach übernommen werden. Die Klasse `GraphView` stellt inzwischen eine Zeichenfläche zur Verfügung, die sich von der Fenstergröße unterscheiden kann. Außerdem kann die Ansicht des „Zeichenblattes“ skaliert werden. Die Darstellungsimplementierung der IDEFIX-Klassen geht jedoch von einer festen Skalierung und einer Zeichenfläche, die der Fenstergröße entspricht, aus. Da keine geerbte Funktionalität des `GraphView` genutzt wurde, ist die Klasse `GDiagramView` von `CanvasView` abgeleitet und auf die neue Methoden zur automatischen Bestimmung der Fenstergröße angepaßt worden.

4.6.2 Einbindung in den Simulator-Prozeß

Der Grund, weshalb die Visualisierungskomponente in den Simulator-Prozeß eingebettet wurde, ist die Tatsache, daß sie für die Übersetzung einer Protokolldatei das zugrundeliegende Automatenmodell kennen muß, das sie so vom Simulator mitbenutzen kann. Die zentrale Komponente der Visualisierung, eine Instanz der Klasse `GDiagramManager`, wird deshalb vom `PLCbgSimulator` nach einem erfolgreichen Einlesen der Modellbeschreibung initialisiert. Der Dialog (das View) des `GDiagramManager` ist zu diesem Zweck als `SystemView`[6] implementiert worden, so daß bereits während der Simulation vorher erzeugte Protokolle betrachtet werden können. Zu diesem Zweck mußte auch das Kontrollfenster der Simulation als `SystemView` realisiert werden.

[6] Fenster werden in der MOBY-Klassenbibliothek durch die Klassen `SystemView` und `PopUpView` erzeugt, wobei ein `PopUpView` allerdings die Benutzerinteraktion für alle anderen Fenster blockiert, bis es geschlossen wurde.

Kapitel 5

Beispielsitzung

Durch dieses Kapitel soll ein Einblick gegeben werden, wie der Hintergrundsimulator benutzt wird, insbesondere wie die Optionen bei der Simulation genutzt werden. Bei dieser Einführung soll auf den SPS-Automaten aus Abschnitt 2.1.4 des Grundlagenkapitels zurückgegriffen werden.

5.1 Der SPS-Automat „Blinklichtsteuerung“

Für die Demonstration des Hintergrundsimulators soll der SPS-Automat zur Ansteuerung eines Blinklichts, wie er im Grundlagenkapitel in Abbildung 2.3 auf Seite 13 gezeigt wurde, benutzt werden. Dieser SPS-Automat ist kein umfassendes Beispiel für die Spezifikationsmöglichkeiten durch SPS-Automaten. Er wird hier aber dennoch benutzt, da er durch das Grundlagenkapitel bereits bekannt ist und die wesentlichen Aspekte der Hintergrundsimulation an ihm dargestellt werden können. Allerdings werden noch ein paar Modifikationen vorgenommen.

Zum einen wird der Typ der Variablen vereinfacht, so daß statt des Aufzählungstyps boolesche Variablen verwendet werden, wobei der Wert `an` durch den booleschen Wert `true` repräsentiert. Desweiteren soll der aktuelle Automatenzustand in den Ausdrücken des Protokolls verwendbar sein, so daß die spezielle Variable `states` zusätzlich definiert wird. Eine Übersetzung aller Bezeichner ins Englische ergibt schließlich den Automaten, der in Abbildung 5.1 dargestellt ist.

5.2 Vorbereitung

Bevor die Simulation des SPS-Automaten angestoßen wird sollen zunächst einige Testfälle festgelegt werden. Zu diesem Zweck wird eine Protokolldatei erstellt, die nur die Belegung des globalen Inputs beschreibt.

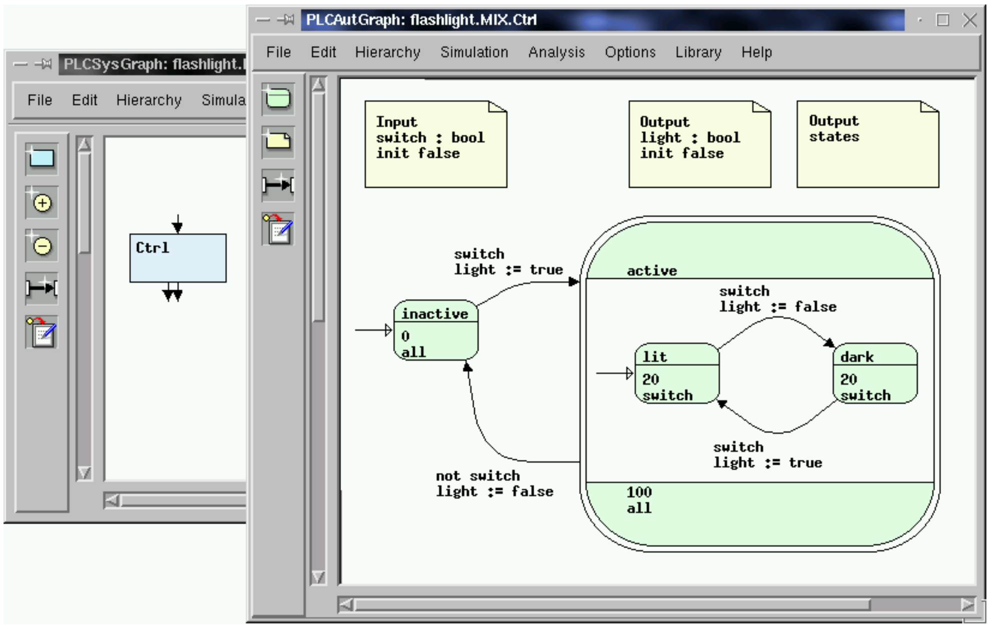

Abbildung 5.1: Simulierter SPS-Automat

Als erstes soll überprüft werden, ob der Automat überhaupt mit dem Blinken beginnt. Dazu muß der Schalter aktiviert werden, so daß für den Treiber von `Ctrl.switch` der Wert `true` eingeplant wird. Danach soll der Schalter zunächst für 120 Zeiteinheiten (ZE) eingeschaltet bleiben, so daß auch sicher im Anschluß durch „Ausschalten“ aus dem Subautomaten, der für 100 ZE nicht verlassen werden kann, gesprungen wird.

```
// initial ist der Schalter aus.
AT 0;
Ctrl.switch <- false;

// Testfall 1: für 120 ZE Treiber von switch = true
AT 50;
Ctrl.switch <- true;
WAIT 120;
Ctrl.switch <- false;
```

Anschließend soll das Verhalten des Automaten beobachtet werden, wenn während der 100 ZE, während denen der Subautomat nicht verlassen werden kann, der Schalter kurzfristig ausgestellt wird. Dazu wird der Schalter aktiviert, 40 ZE gewartet und dann deaktiviert, wieder 40 ZE gewartet und wieder aktiviert und das für weitere 40 ZE. Danach soll das Blinklicht für eine Dauer von 120 ZE ausgeschaltet bleiben. Dementsprechend sieht der entsprechende Traceausschnitt wie folgt aus:

```
  //Testfall 2: während das Licht blinkt für 40 ZE noch während
  //            der ersten 100 ZE im Subautomat ausschalten
  AT 300;
  Ctrl.switch <- true;
  WAIT 40;
  Ctrl.switch <- false;
  WAIT 40;
  Ctrl.switch <- true;
  WAIT 40;
  Ctrl.switch <- false;
  WAIT 120;
```

Desweiteren soll der Benutzer durch einen Unterbrechungspunkt informiert werden, wenn das Licht länger als 25 ZE angeblieben ist. Dieses kann durchaus vorkommen, nämlich wenn das letzte Zyklusende erst unmittelbar vor dem Ablaufen des Timers stattfand und für das nächste Zyklusende annähernd die 10 ZE Zyklusdauer ausgeschöpft werden. Hierbei handelt es sich um korrektes Systemverhalten, so daß dementsprechend auch kein Halt der Simulation veranlaßt wird.

```
// Breakpoint 1: Information "Licht lange an"
BREAKPOINT Ctrl.light
  SINCE 25
  REPORT "Blinklicht seit 25 Zeiteinheiten an!"
```

Schließlich soll noch überprüft werden, ob das Licht auch in dem vorgesehenen Rahmen ausgeschaltet wird. Wird die Blinklichtsteuerung ausgeschaltet, während der Automat im Superzustand `active` ist, so kann im ungünstigsten Fall erst dann in den Zustand `inactive` gewechselt werden, wenn die Verzögerungszeit plus die Zyklusdauer verstrichen sind, also 100 ZE + ε = 110 ZE. Die Simulation soll gestoppt werden, wenn nach dieser Zeitspanne bei ausgeschaltetem Blinklicht der Zustand `inactive` noch nicht erreicht wurde. Hier muß nicht die maximale Zyklusdauer zweifach addiert werden, weil zur Überprüfung bereits der gepollte Wert herangezogen wird. Demnach hat der Automat nur noch die einfache Zyklusdauer nach Ablaufen des Timers zur Verfügung, um das Zyklusende festzulegen. Wenn die Bedingung des Unterbrechungspunktes zu diesem Zeitpunkt noch erfüllt ist, so muß noch immer der Wert `false` für die Eingabe gepollt worden sein. Ist dann aber auch der Zustandswechsel noch nicht erfolgt, so liegt ein Fehlverhalten des Systems[1] vor.

[1] Da die Semantik des SPS-Automat dieses nicht zuläßt, liegt beim Auslösen des Unterbrechungspunktes ein Fehler in der Simulatorimplementierung vor.

```
// Breakpoint 2: kritischer Systemfehler
BREAKPOINT (not Ctrl.switch and (Ctrl.states # 'Ctrl.inactive))
  SINCE 111
  REPORT "'inactive' nach 111 Zeiteinheiten noch nicht erreicht!"
  STOP;
```

Faßt man die oben gezeigten Protokollteile zusammen, wobei die Traceausschnitte dem `TRACE`-Abschnitt zugeordnet werden, so ist das Eingabeprotokoll bereits fertig. Die im folgenden gezeigte Zusammenfassung stellt die bereits gezeigten Teile nur noch durch ihren initialen Kommentar dar. Nachdem dieses Eingabeprotokoll in einer Datei gespeichert wurde, kann die Simulation gestartet werden.

```
// Breakpoint 1: Information "Licht lange an"
...

// Breakpoint 2: kritischer Systemfehler
...

TRACE {
  // initial ist der Schalter aus.
  ...

  // Testfall 1: für 120 ZE Treiber von switch = true
  ...

  // Testfall 2: ...
  ...
}
```

5.3 Start der Simulation

Wie bisher kann die Simulation aus dem Editor über den Menüpunkt „Simulate“ des Menüs „Simulation“ gestartet werden. Nachdem die Sicherheitsabfrage bestätigt wurde, wird der Simulator initialisiert. Er übersetzt zunächst das Modell in die interne Datenstruktur und erzeugt anschließend die Funktionen, die für Bedingungen und Aktionen im Automatenmodell nach dem interaktiv kompilativen Ansatz benötigt werden. Schließlich wird die Hintergrundsimulation aktiviert, indem die Protokoll-Übersetzung den Dialog für die Simulations- und Trace-Optionen öffnet.

Der Dialog besteht aus drei Seiten, die durch die Laschen am oberen Rand umgeschaltet werden können. Zu Beginn wird die Seite „General“ gezeigt, die

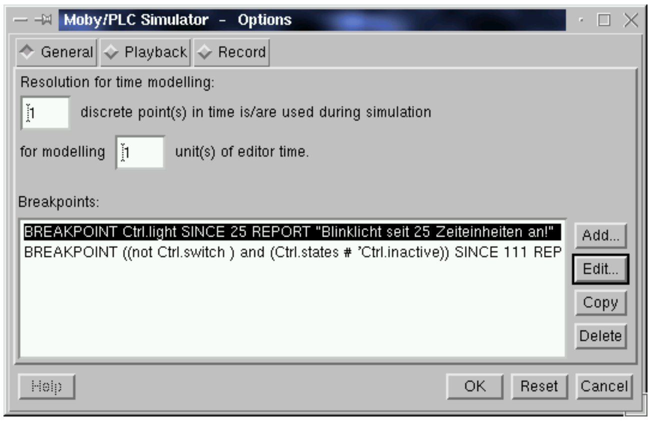

Abbildung 5.2: Optionen der Simulation – „General“

in Abbildung 5.2 zu sehen ist. Auf der oberen Hälfte dieser Seite kann die Zeitauflösung für die Simulation eingestellt werden.

Darunter findet sich eine Liste, in der die definierten Unterbrechungspunkte aufgeführt sind. Diese ist zu Beginn leer und kann zum einen interaktiv durch die Knöpfe an ihrer rechten Seite bearbeitet werden. Desweiteren wird sie aber auch beim Einlesen einer Protokolldatei gefüllt, wenn darin neue Unterbrechungspunkte definiert werden.

Durch das Umschalten auf die Seite „Playback“ erhält man die in Abbildung 5.3 dargestellten Optionen für das Einlesen eines Protokolls. Zunächst muß hier durch den Schalter am oberen Rand das Lesen einer Datei aktiviert werden. In der Zeile darunter wird der Dateiname des Protokolls angegeben, das durch Betätigen des „Select“-Knopfes gewählt werden kann. Eine Protokolldatei wird bereits bei dieser Auswahl in die interne Datenstruktur übersetzt, so daß Syntaxfehler unmittelbar nach der Wahl des Protokolls angezeigt werden.

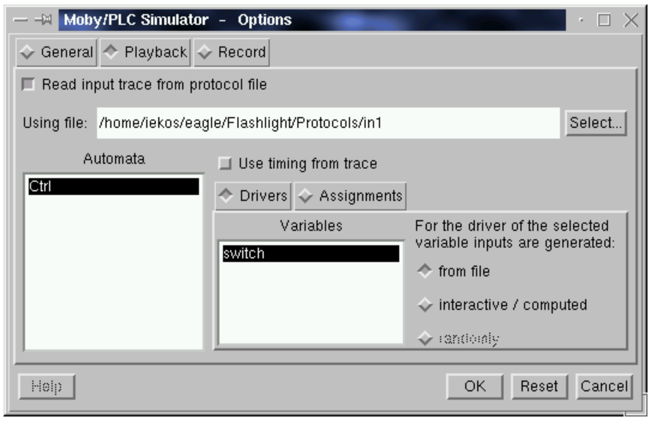

Abbildung 5.3: Optionen der Simulation – „Playback“

Im Bereich unter dieser Zeile ist auf der linken Seite des Fensters die Liste der SPS-Automaten, die simuliert werden sollen, zu finden. Alle Angaben rechts von dieser Liste beziehen sich auf den hier ausgewählten Automaten. Dabei handelt es sich um die Option, die angibt, ob das Zeitverhalten des Automaten aus der Protokolldatei übernommen werden soll, und um Optionen zu den Variablen des Automaten. Letztere werden wieder durch ein zweiseitiges Unterfenster gegliedert. Auf der ersten Seite „Drivers“ kann eingestellt werden, welche Eingabevariablen durch das Protokoll getrieben werden sollen. Die Vorgabe erlaubt das Lesen aus der Datei nur für die Variablen, die globale Inputs darstellen. Auf der zweiten Seite „Assignments“ kann eingestellt werden, daß direkte Zuweisungen für bestimmte Variablen dem Protokoll entnommen werden dürfen. Da hierdurch Verletzungen der Semantik möglich sind, ist die Voreinstellung für alle Variablen, daß Zuweisungen nicht dem Protokoll entnommen werden. Diese Option kann nur aktiviert werden, wenn der weitere Dialog aus Abbildung 5.4, der auf diesen Umstand hinweist, bestätigt wurde.

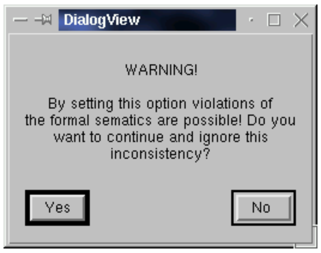

Abbildung 5.4: Sicherheitsabfrage zur Übernahme von Variablenzuweisungen

Für die Simulation des Beispielautomaten können fast alle Optionen unverändert bleiben. Es wird lediglich das Lesen des Zeitverhaltens aus der Trace deaktiviert, denn dieses wird durch das Eingabeprotokoll nicht beschrieben. Wird vergessen, diese Option zu deaktivieren, so wird bei der Simulation bereits für den ersten SPS-Zyklus ein Meldung erscheinen, daß keine Zeitinformationen der Trace entnommen werden konnten, und die Möglichkeit geboten, diese Option nachträglich auszuschalten. Das Lesen der Treiberänderungen für den Eingang `switch` bleibt aktiviert.

Die dritte Seite, „Record“, der Optionen ist nach dem gleichen Schema wie die „Playback“-Seite aufgebaut. Oben auf dieser Seite, die in Abbildung 5.5 gezeigt wird, kann gewählt werden, daß eine Protokollaufzeichnung erfolgt, und durch Betätigen des „Select“-Knopfes wird ein Dateiname hierfür gewählt.

Im unteren Bereich wird festgelegt, welche automatenspezifischen Optionen im Protokoll festzuhalten sind. Zum einen kann das Zeitverhalten protokolliert werden, wodurch Poll- und Cycle-Events des SPS-Automaten aufgezeichnet werden. In dieser Beispielsitzung soll das Simulationsprotokoll sowohl

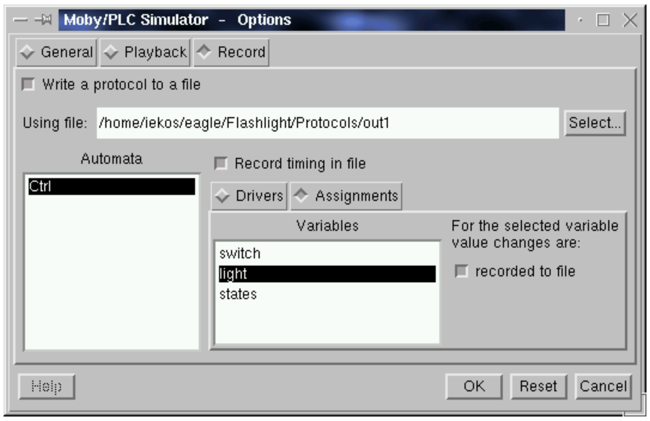

Abbildung 5.5: Optionen der Simulation – „Record“

zur Wiederholung der Simulation, als auch zur Darstellung der Simulation in der Visualisierungskomponente benutzt werden, so daß für den einzigen simulierten SPS-Automaten `Ctrl` diese Option eingeschaltet wird, bzw. bleibt, da sie bereits in der Vorgabe aktiviert ist.

Insbesondere um vom Benutzer getätigte Eingaben an den globalen Inputs festzuhalten, kann auf der „Driver“-Seite für jede Eingangsvariable festgehalten werden, ob eine Änderung ihres Treibers ins Protokoll aufzunehmen ist. Die Standard-Einstellung sieht dieses für die globalen Inputs vor. Soll jedoch ein Protokoll für die wiederholte Simulation einer Teilkomponente erstellt werden, so kann dieses auch für andere Variablen sinnvoll sein. Da in der Beispielsitzung nicht mehr interaktiv in die Simulation eingegriffen werden soll und die Treiberinformationen für die einzige Eingangsvariable bereits durch das Eingabeprotokoll festgelegt sind, bräuchten diese nicht mehr ins Simulationsprotokoll aufgenommen werden. Um aber eine vollständige Aufzeichnung (in einer Datei) zu erhalten, wird die Option dennoch aktiviert.

Die Aufzeichnung von Zuweisungen an Variablen ist gerade für eine nachträgliche Betrachtung in der Visualisierungskomponente wichtig. Zur Wiederholung einer Simulation werden sie hingegen nicht benötigt, da sie sich aus den Eingaben an globalen Inputs und dem Zeitverhalten errechnen lassen. Wie bereits erwähnt, soll aber die Aufzeichnung für beide Anwendungsfälle benutzt werden. Die Vorgabe sieht für alle unverbundenen Ausgangsvariablen (globale Outputs) die Aufzeichnung vor. Weil aber nur die „echte“ Ausgabe, also die Variable `light`, und nicht die spezielle Variables `states` im nachhinein dargestellt werden soll, wird die Option für `states` deaktiviert.

5.4 Durchführung der Simulation

Nachdem alle Optionen für die Simulation eingestellt sind, kann der Dialog mit „OK“ beendet werden, worauf das Kontrollfenster der Simulation erscheint. Dieses Fenster wurde in dieser Diplomarbeit um den Eintrag „Input Polling“ im Menü „Options“ erweitert. Da der betrachtete SPS-Automat aber nur eine Eingangsvariable besitzt, ist diese Einstellung irrelevant. Desweiteren wird in der Anzeige der Simulationszeit diese auch in Editor-Zeiteinheiten umgerechnet dargestellt. Alle weiteren Komponenten des Kontrollfensters wurden in [Sch97] bzw. durch die Arbeit der Abteilung entwickelt.

Die Simulation kann nun durch Betätigen des „Run“-Knopfes gestartet werden. Da alle Eingaben bereits vorher festgelegt wurden und die gesamte Simulation aufgezeichnet wird, kann die Geschwindigkeit noch erhöht werden, um schneller zu Ergebnissen zu kommen. Bei der Simulation wurde nun der erste Unterbrechungspunkt zunächst zum Zeitpunkt 85 ausgelöst, und vom Simulator durch die Meldung in Abbildung 5.6 signalisiert. Bei einer erneuten Simulation mit der gleichen Eingabe muß dieser Unterbrechungspunkt nicht zum gleichen Zeitpunkt stattfinden[2], da das Zeitverhalten des Simulators nicht im Protokoll festgelegt wird.

Abbildung 5.6: Meldung des ersten Unterbrechungspunktes

Zum Simulationszeitpunkt 420 findet die letzte Treiberänderung durch das Eingabeprotokoll statt, so daß kurze Zeit später die Simulation angehalten werden kann. Dieses könnte auch automatisch geschehen, indem ein geeigneter Eintrag in die Trace geschrieben wird.

```
AWAIT false ELSE REPORT "genug simuliert" STOP;
```

Die Bedingung `false` einer solchen Zusicherung in der Trace kann von keinem Systemzustand erfüllt werden, so daß die Aktion in jedem Fall ausgeführt wird.

[2]Es ist sogar recht unwahrscheinliche, daß der Simulator bei einem weiteren Lauf ein identisches Zeitverhalten erzeugt.

5.5 Visualisierung des Simulationsprotokolls

Ein Neustart oder Beenden des Simulators sichert die aufgezeichnete Trace, sowie die benutzten Unterbrechungspunkte in der zuvor festgelegten Datei[3]. Diese kann nun von der Visualisierungskomponente benutzt werden, um Diagramme darzustellen, die zeitabhängig die Belegung der Variablen, bzw. der Treiber zeigen. Die Verwaltungsinstanz für diese Diagramme ist der Diagramm-Manager, dessen Fenster in Abbildung 5.7 gezeigt wird. Nachdem über den „Select"-Knopf eine Protokolldatei erfolgreich eingelesen wurde, können über die entsprechenden Schaltflächen beliebig viele Diagramme erzeugt werden.

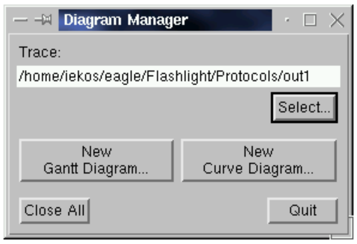

Abbildung 5.7: Fenster des Diagramm-Managers

Abbildung 5.8 zeigt in einem Gantt-Diagramm den gesamten Verlauf der aufgezeichneten Variablen. Eine solche Übersicht wird erzeugt, indem die Zeitachse des Diagramms über das Kontext-Menü skaliert wird (vgl. Abbildung 5.9). Dem vollständigen Variablennamen wird bei der Darstellung ein „`=`" vorangestellt, wenn direkt der Variablenwert visualisiert wird, und die Zeichenfolge „`->`", wenn der Treiber der Variable gemeint ist[4].

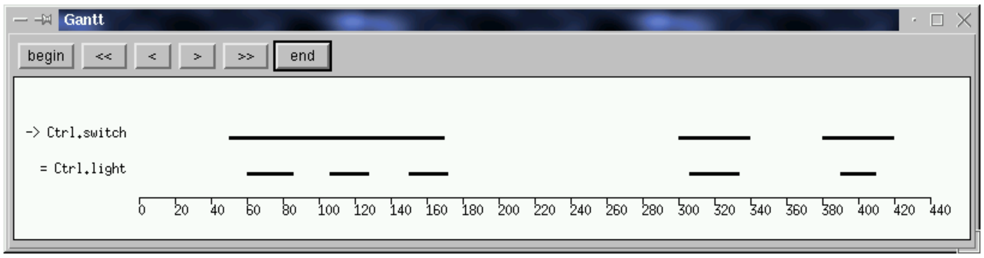

Abbildung 5.8: Übersicht über die Simulation

Der erste Testfall soll nun etwas genauer betrachtet werden. Für den Zeitpunkt 85 wurde durch die Meldung des ersten Unterbrechungspunktes signalisiert, daß die Ansteuerung für die Glühbirne bereits seit 25 Zeiteinheiten

[3]Falls keine Dateiname gewählt wurde, die Trace aber nicht leer ist, so wird noch die Gelegenheit gegeben, den Dateinamen nachträglich zu bestimmen.

[4]Der Pfeil soll immer auf die Variable zeigen, die den Wert pollt, deshalb ist er hier andersherum als bei den Trace-Einträgen.

aktiviert ist. Um dieses anhand des Diagramms nachzuvollziehen, wird über das Kontextmenü eine geeignete Auflösung für die Zeitachse gewählt (hier z.B. 10) und dann über die Knöpfe oberhalb der Darstellungsfläche zu der gewünschten Zeit gesprungen. Wird so die Zeit ab dem Zeitpunkt 50 dargestellt, dann kann man im Diagramm erkennen, daß seit dem Zeitpunkt 60 tatsächlich genau bis zum Zeitpunkt 85 die Variable `light` den Wert `true` angenommen hat. Abbildung 5.9 zeigt genau den beschriebenen Simulationszeitraum.

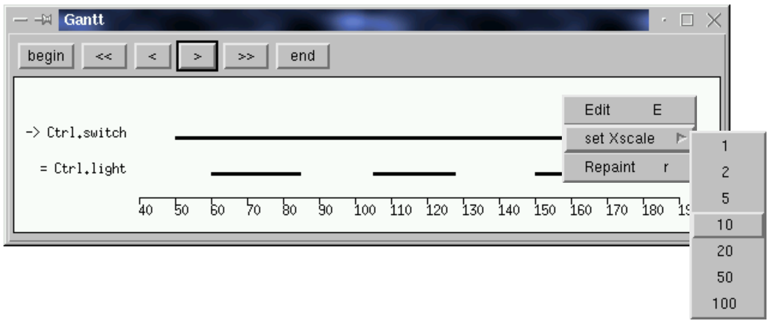

Abbildung 5.9: Der erste Testfall und das Kontextmenü

Für eine Darstellung des zweiten Testfalls wurde in Abbildung 5.10 ein Kurvendiagramm gewählt. Dieser Diagrammtyp dient zur Darstellung von ganzzahligen Werten über den betrachteten Simulationszeitraum. Für die Darstellung von booleschen Werten wurde deshalb eine Abbildung vorgenommen, bei der der Wert `false` wie 0 und der Wert `true` wie 1 behandelt wird.

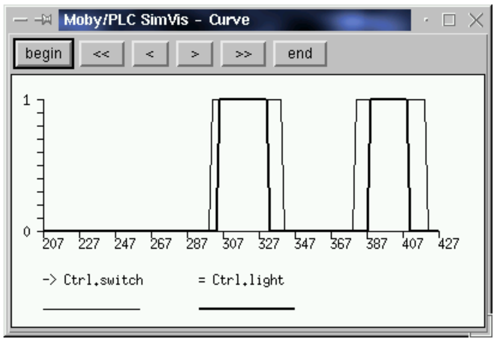

Abbildung 5.10: Verhalten beim zweiten Testfall

In der Darstellung der Simulation wird deutlich, daß mit dem Ausschalten auch das Blinken stoppt, die genauen internen Vorgänge werden jedoch nicht aus dem Diagramm ersichtlich, und somit kann das Verhalten anhand des Bildes nicht gedeutet werden. Für dieses Problem wurde das Protokoll so erzeugt, daß es sich auch zur Wiederholung der Simulation eignet.

Beim Neustart der Simulation wird nun das Simulationsprotokoll als Eingabeprotokoll benutzt. Um eine genaue Wiederholung zu erhalten, muß im

Optionen-Dialog das Lesen des Zeitverhaltens und des treibenden Wertes für den Eingang aktiviert werden. Sind diese Einstellungen erfolgt, dann kann über den „Time"-Knopf im Kontrollfenster der Simulation bis zum Zeitpunkt 300 simuliert werden. Hierbei tritt dann tatsächlich – wie erwartet – die Meldung des ersten Unterbrechungspunktes zu genau dem gleichen Zeitpunkt wie zuvor auf.

Ist die Simulation bis zum Zeitpunkt 300 vorangeschritten, kann die sie Schritt für Schritt nachvollzogen werden. Hierbei lassen sich dann alle Vorgänge, wie bisher bei der interaktiven Simulation, auch im Editor beobachten. Dadurch wird deutlich, daß der Automat den Subautomaten `active` zum Zeitpunkt 340, zu dem der Treiber den Wert `false` annimmt, nicht verläßt und dieses auch nicht kann, da die 100 Zeiteinheiten des Timers, der für jede Eingabe zu beachten ist, noch nicht abgelaufen sind. Da die Eingabe aber auch nicht `true` ist, verweilt der SPS-Automat im Zustand `dark`, bis die Eingabe ihren Wert wieder ändert. Beim zweiten Setzen der Eingabe auf den Wert `false` hat der Automat den Subautomaten seit 120 Zeiteinheiten nicht mehr verlassen. Somit ist der Timer abgelaufen und der SPS-Automat kann beim nächsten Zyklusende in den Zustand `inactive` wechseln.

Kapitel 6

Abschließende Anmerkungen

Den Abschluß des Hauptteils dieser Diplomarbeit soll eine Betrachtung des Erreichten bilden. Hier soll noch einmal zusammengefaßt werden, was im Rahmen der Diplomarbeit realisiert werden konnte. Ebenso werden Anregungen gegeben, die zeigen sollen, wie das Entstandene noch komplettiert bzw. erweitert werden kann.

6.1 Ergebnisse der Diplomarbeit

6.1.1 Erreichtes

Diese Diplomarbeit hat erstmals in der Beschreibung der Grundlagen im Abschnitt 2.3 dargestellt, warum die Implementierung des vorhanden Simulators sinnvoll ist. Sinnvoll heißt hier, daß das Verhalten der SPS-Automaten, die im Editor eingegeben werden, während der Simulation verträglich ist mit der formalen Semantik. Dazu wurde zum einen die Entfaltungsfunktion aus [Sch97] informell auf hierarchische SPS_V-Automaten erweitert, zum anderen wurde die TA-Semantik aus [DFMV98] auf hierarchische SPS-Automaten ($SPS_{\mathcal{H}}$-Automaten) übertragen.

semantische Betrachtung

Als weiteres Ziel konnte erreicht werden, daß ein kompletter Simulationslauf mit allen nötigten Informationen für eine identische Wiederholung aufgezeichnet werden kann. Die so aufgezeichneten Daten werden in eine textuelle Repräsentation gebracht, so daß sie mit jedem Programm zum anzeigen von Texten betrachtet werden können.

Aufzeichnung einer Simulation

Die so gewonnene Protokolldatei kann wiederum dazu benutzt werden, um eine erneute, identische Simulation zu erhalten. Weil der Simulator nach wie vor den aktuellen Systemzustand im Editor visualisiert, kann dieses dazu dienen, sich einen kritischen Abschnitt der Simulation noch einmal anzuschauen. Die Steuerung einer Simulation muß jedoch nicht vollständig durch das eingelesene Protokoll festgelegt sein. Gerade wenn es darum geht zu testen, ob das

Reproduktion einer Simulation

System auf einen bestimmten Eingabestrom immer[1] richtig reagiert, ist ein Protokoll von Nutzen, in dem lediglich dieser Eingabestrom beschrieben ist, das jedoch nicht das Zeitverhalten der simulierten Automaten vorschreibt. Einige Möglichkeiten wurden in der Diplomarbeit entwickelt, um solch eine Eingabebeschreibung einfach halten zu können. Außerdem bleiben bei der Wiederholung einer Simulation sämtliche interaktive Eingriffsmöglichkeiten erhalten.

Unterbrechungspunkte

Zur Kontrolle des Ablaufs und damit zur Steuerung einer Simulation dienen aber auch die neu hinzugekommen Unterbrechungspunkte. Sie erlauben es dem Benutzer in bestimmten Ausnahmefällen Nachrichten zu erhalten, bzw. in die Simulation einzugreifen, so daß ein Fehlverhalten nicht übersehen wird, bzw. schon frühzeitig erkannt werden kann.

Visualisierungskomponente

Eine weitere Kontrollmöglichkeit bietet schließlich die Visualisierungskomponente. Gerade zeitliche Zusammenhänge, sowie das Verhältnis der Zeitintervalle zwischen verschiedenen Ereignissen werden durch diese Komponente besonders gut dargestellt.

6.1.2 Nicht Erreichtes

„Undo"-Funktion

Die ursprüngliche Hoffnung, daß durch Hintergrundsimulation auch eine „Undo"-Funktion entsteht, also eine Möglichkeit, Simulationsschritte wieder zurücknehmen zu können, konnte nicht erfüllt werden. Dieser Umstand hängt damit zusammen, daß aufgezeichnete Ereignisse in einer anderen Komponente, der Protokoll-Übersetzung, und in einer anderen Form, den Trace-Einträgen, verwaltet werden. Für die Rücknahme von Simulationsschritten müßten demnach solche Einträge wieder in Ereignisse zurückübersetzt werden. Hierbei entsteht jedoch ein Problem, wenn für ein zurückübersetztes Cycle-Event dessen Folgeereignisse noch in der Ereignisliste vorhanden sind. Auf diese Ereignisse könnte nur direkt zugegriffen werden, wenn sie auch noch anderweitig (als in der Ereignisliste) vermerkt worden wären.

Das Fehlen einer solchen Funktionalität wird aber durch die Reproduzierbarkeit einer Simulation teilweise wieder ausgeglichen. So kann das Protokoll einer Simulation auf den Zeitraum gekürzt werden, bis zu dem noch keine Rücknahme der Simulationsschritte erwünscht ist. Dient dieses Protokoll als Eingabeprotokoll, so kann nach dem Ende der Trace in der Datei die Simulation wieder interaktiv fortgeführt werden. Desweiteren liegt die Vermutung nahe, daß die Rücknahme von Simulationsschritten durch die Möglichkeit, die Eingabe a priori beschreiben zu können, kaum noch nötig ist.

[1] Die Sicherheit, daß sich das System immer richtig verhält wird man i.a. erst durch den Einsatz von Modelchecking erreichen, aber man kann durch Simulation häufig bereits schneller Fehler erkennen.

6.2 Mögliche Erweiterungen

Während der Entwicklung der Hintergrundsimulation und der dazu benötigten Komponenten, sind einige Ideen für Erweiterungen aufgekommen, die im folgenden kurz dargestellt werden sollen.

6.2.1 Die Protokollsprache

Gerade durch den Einsatz eines Parsers kann die Sprache für Eingabeprotokolle sehr mächtig und flexibel gehalten werden. In der Diplomarbeit wurden einige Sprachkonstrukte umgesetzt, die diese Möglichkeiten nutzen. Es wurden jedoch nicht alle ursprünglichen geplanten bzw. angedachten Erweiterungen dieser Sprache realisiert, wenn auch für einige bereits Vorarbeiten geleistet wurden.

Eine dieser Erweiterungen, für die während der Diplomarbeit bereits Anstrengungen unternommen wurden, ist ein Bausteinkonzept. Durch eine solche Erweiterung soll eine Menge von Trace-Einträgen zu einer Art Prozedur zusammengefaßt werden können, so daß die manuelle Erstellung eines Eingabeprotokolls erleichtert wird. In einer weiteren Steigerung würden solche Prozeduren Parameterübergabe, wie man sie aus imperativen Programmiersprachen kennt, erlauben, so daß eine noch höhere Flexibilität erreicht würde.

Ebenso kann aus dem Entwurf entnommen werden, daß auch Überlegungen getätigt wurden, die Schleifenkonstrukte in der Sprache erlauben würden, so daß möglicherweise unendliche lange Traces beschreibbar sind. Gerade weil die SPS-Automaten eines Automatensystems völlig unabhängig voneinander arbeiten, könnte die Eingabsprache auch diesem Umstand Rechnung tragen, indem ein Sprachkonstrukt das Interleaving mehrerer Traces erlaubt, also das chronologisch konsistente Mischen von Einträgen aus mehreren Traces. Bedingte Einträge, bzw. Traces und die Definition lokaler Variablen würden die Sprache des Eingabeprotokolls schließlich mit einer imperativen Programmiersprache vergleichbar machen.

Auch die Angabe von Optionen für die Simulation könnte durch die Protokollsprache erfolgen. So wie dieses bereits für die Simulationsoptionen, die Zeitauflösung und das Pollverhalten, erfolgt, könnten auf gleiche Art und Weise die vielfältigen Trace-Optionen festgelegt werden. Auf diese Weise könnte bereits beim Schreiben einer Protokolldatei vermerkt werden, welche Informationen der Trace entnommen werden können, so daß die Vorbelegung der Optionen besser an die Eingabe angepaßt wäre.

Für eine Weiterentwicklung der Protokollsprache sollte jedoch das Anwendungsgebiet bekannt sein, um nicht Aufwand für Funktionalität zu investieren, die im nachhinein nicht genutzt wird. Wird die Simulation intensiv zur

Überprüfung eines entworfenen Systems genutzt und ist hierbei die Umgebungsmodellierung durch SPS-Automaten nicht angemessen, dann kann eine komplexe Sprache die Formulierung der für den Test erforderlichen Eingabe – i.a. die Belegung der globalen Inputs, möglicherweise in Abhängigkeit von Ausgaben – sehr erleichtern.

Wird aber eher eine Systemüberprüfung durch Modelchecking angestrebt, so wird die Eingabesprache häufig nur genutzt, um einen vom Modelchecker entdeckten fehlerhaften Simulationslauf im Automatenmodell reproduzieren zu können. In diesem Fall wäre die vorhandenen Eingabesprache ausreichend, bzw. es wären andere Anforderungen an Spracherweiterungen zu stellen. So wären Sprachkonstrukte denkbar, die für Variablen, die durch Abstraktionen beim Modelchecking außer Acht bleiben, zufällige Belegungen, bzw. Belegungen anhand verschiedener Strategien erzeugen.

6.2.2 Unterbrechungspunkte

Werden häufig Unterbrechungspunkte zur Simulationskontrolle genutzt, so wären in diesem Bereich ebenfalls noch Erweiterungen denkbar. Zunächst wäre eine erweiterte Möglichkeit, den zeitlichen Aspekt in den Bedingungen zu berücksichtigen, denkbar. Eventuelle würde die Wiederverwendbarkeit eines Unterbrechungspunktes in der Bedingung eines anderen bereits in diese Richtung beitragen, d.h. ein Unterbrechungspunkt wäre selbst auch wieder eine Bedingung, die erfüllt ist, wenn der Breakpoint aktiviert ist.

Derzeit ist es z.B. nicht möglich, einen Ausdruck zu formulieren, der besagt, daß Bedingung A bereits 10 Zeiteinheiten und Bedingung B seit 5 Zeiteinheiten erfüllt ist, um einen Unterbrechungspunkt auszulösen. Durch die Wiederverwendbarkeit eines Unterbrechungspunktes könnte dieses ausgedrückt werden.

```
BREAKPOINT bp_A := A SINCE 5;
BREAKPOINT bp_B := (bp_A and B) SINCE 5 STOP;
```

In diesem Beispiel ist `bp_B` erfüllt, wenn sowohl die Bedingung B seit 5 Zeiteinheiten gilt, als auch der Unterbrechungspunkt `bp_A` seit 5 Zeiteinheiten aktiviert ist. Letzters ist genau dann der Fall, wenn Bedingung A bereits seit 5 Zeiteinheiten erfüllt ist, so daß zusammengenommen Bedingung A bereits seit 10 Zeiteinheiten gilt.

6.2.3 Visualisierung

Obwohl ursprünglich die Visualisierungskomponente eine gleichwertige Bedeutung wie das Aufzeichnen bzw. die Reproduktion einer Simulation haben sollte, konnte ihr gegen Ende der Bearbeitungszeit für diese Diplomarbeit nicht die gleiche Aufmerksamkeit geschenkt werden. Das heißt nicht, daß die Visualisierungskomponente nicht funktionsfähig ist, sondern daß die ursprüngliche Vorstellung davon ausging besonders in diesem Bereich Erweiterungen bringen zu können. Stattdessen wurde lediglich eine Anpassung der bestehenden Klassen durchgeführt, was aber auch schon einen nicht zu unterschätzenden Arbeitsaufwand zur Folge hatte – als Vergleich sei hier auf die Probleme bei einer Portierung der MOBY-Klassenbibliothek nach Java, die in [Bar98] beschrieben werden, hingewiesen.

Gerade auf SPS-Automaten spezialisierte Darstellungsformen wären hier als erstes anzustreben, so daß insbesondere das Zeitverhalten eines SPS-Automaten graphisch dargestellt werden kann. Desweiteren könnte eine Erweiterung, die konzeptionell den Netzausdrücken der IDEFIX-Implementierung entspricht, sinnvoll sein, wodurch es erlaubt würde auch den Wert eines Ausdrucks über die aufgezeichneten Variablen in seinem zeitlichen Verlauf darzustellen.

Anhang A

SPS_V-Automaten

Im folgenden wird die Syntax der in [Sch97] definierten SPS_V-Automaten vorgestellt, deren Semantik durch die Abbildung auf $\mathrm{SPS}_{\mathcal{H}}$-Automat, also hierarchische SPS-Automaten, definiert wird. Die sogenannte Entfaltungsfunktion, die diese Abbildung leistet, wird in [Sch97] formal definiert. Eine informelle Beschreibung, die zudem Hierarchie bei SPS_V-Automaten berücksichtigt, findet sich im Abschnitt 2.1.3 ab Seite 11.

SPS_V-Automaten benutzen zur Beschreibung des Ein- und Ausgabealphabets, sowie zur Erweiterung der Zustandsmenge Variablen, die dementsprechend in Eingabevariablen V_i, Ausgabevariablen V_o und lokale Variablen V_l unterteilt sind. Diese Variablen $V = V_i \cup V_l \cup V_o$ sind getypt, wobei zwischen booleschen Variablen, die den Typ $\mathbb{B} = \{true, false\}$ haben, Integer-Variablen, die als Typ eine endliche Teilmenge von $\mathbb{N}$ haben, und Variablen mit Aufzählungstyp, deren Typ eine endliche Menge von Bezeichnern ist und die deshalb als Spezialfall von Integer-Variablen angesehen werden[1], unterschieden wird. Der Typ einer Variablen v wird durch die Funktion $\vartheta(v)$ ermittelt, d.h. $\vartheta : V \to \wp(\mathbb{N}) \cup \{\mathbb{B}\}$. Die Belegung einer Menge V von Variablen ist eine Abbildung $\gamma : V \to \mathbb{N} \cup \mathbb{B}$, die jeder Variable einen Wert, ein Element ihres Typs, zuordnet. Demnach ergeben sich alle möglichen Belegungen einer endlichen Menge V von Variablen zu

$$\Theta(V) = \{\gamma : V \to \mathbb{N} \cup \mathbb{B} \mid \forall v \in V : \gamma(v) \in \vartheta(v)\}.$$

Als Abkürzende Schreibweise wird $\Theta_x \overset{df}{=} \Theta(V_x)$ für $x \in \{i, l, o\}$ und $\Theta_{lo} \overset{df}{=} \Theta(V_l \cup V_o)$ für die Belegung der lokalen und der Ausgangsvariablen benutzt. Da ein SPS-Automat deterministisch reagieren soll, müssen die Variablen mit einer initialen Anfangsbelegung $\gamma_0 \in \Theta_{lo}$ für die lokalen (V_l) und Ausgangsvariablen (V_o) versehen werden.

[1]Für die Bezeichner eines Aufzählungstyps gibt es immer eine Bijektion auf eine Teilmenge natürlicher Zahlen, die nicht für Integer-Variablen benutzt werden, da alle Typen endliche Mengen sind.

Da bei SPS_V-Automaten immer nur eine endliche Menge von Variablen zugelassen wird, die ihrerseits eine endliche Menge als Typ besitzen, so kann eine Belegung der Variablen $V = \{v_1, \ldots, v_k\}$ auch als Vektor $(\gamma(v_1), \ldots \gamma(v_k))$ geschrieben werden. Entsprechend stellt sich die Menge der möglichen Belegungen als Kreuzprodukt der Typen dar, d.h. $\Theta(V) = \vartheta(v_1) \times \cdots \times \vartheta(v_k)$.

Die Wahl des Zustandsüberganges wird nach der Transitionsrelation ϕ beeinflußt durch die Belegung der Variablen. Ebenso beeinflußt ein erfolgter Zustandsübergang wiederum die Belegung der Variablen. Hierzu werden Ausdrücke aus den booleschen Operatoren $\{and, or, xor, not\}$, den ganzzahligen Operatoren $\{+, -, \times, /, \%\}$, den Vergleichsoperatoren $\{=, \neq, <, \leq, \geq, >\}$, den Klammeroperatoren $\{(,)\}$, beliebigen Variablen aus V und Konstanten gebildet. Die Menge aller dieser Ausdrücke sei mit $TERM(V)$ bezeichnet. Die Funktion ϑ wird auch auf diese Ausdrücke erweitert, so daß $\vartheta(\tau)$ den Typ eines solchen Terms liefert, d.h $\vartheta : V \cup TERM(V) \rightarrow \wp(\mathbb{N}) \cup \{\mathbb{B}\}$. Als Bedingungen für Zustandsübergänge dienen Ausdrücke $\tau \in TERM(V)$, die einen booleschen Wert liefern, also für die $\vartheta(\tau) = \mathbb{B}$ gilt. Die Menge diese Bedingungen wird mit $TERM_\beta(V)$ bezeichnet.

$$TERM_\beta(V) = \{\tau \in TERM(V) \mid \vartheta(\tau) = \mathbb{B}\}$$

Wertzuweisungen

Wertzuweisungen werden an Zustandsübergängen ausgeführt. Sie bestehen aus einer lokalen oder Ausgangsvariablen $x \in V_l \cup V_o$, dem Zuweisungsoperator $:=$ und einem Wertterm $\tau \in TERM(V)$, wobei die Variable und der Ausdruck typverträglich sind, d.h. $\vartheta(\tau) \subseteq \vartheta(x)$. Die Menge der Wertzuweisungen $x := \tau$ wird bezeichnet mit $TERM_\alpha(V_l \cup V_o, V)$. An den Zustandsübergängen können mehreren lokalen und Ausgangsvariablen auf diese Weise neue Werte zugewiesen werden:

$$\phi \subseteq Q \times TERM_\beta(V) \times (TERM_\alpha(V_l \cup V_o, V))^\star \times Q$$

deterministisch

Eine Transitionsrelation ist *deterministisch*, wenn für jeden Zustand des SPS_V-Automaten und jede Belegung der Variablen die Bedingung von höchstens einer Transition wahr wird. Mathematisch formuliert bedeutet dieses:

$$\begin{aligned} \forall q \in Q, \gamma \in \Theta(V) : (\exists e \in \phi : e = (q, \tau_\beta, \tau_\alpha, q') \wedge \tau_\beta[\gamma] = true) \\ \Longrightarrow \forall e' \in \phi, e' = (q, \tau'_\beta, \tau'_\alpha, q''), e' \neq e : \tau'_\beta[\gamma] = false \end{aligned}$$

SPS_V-Automat

Definition A.1: Ein (deterministischer) *SPS_V-Automat* ist ein Tupel

$$\mathcal{A} = (Q, V, \phi, q_0, \gamma_0, \varepsilon, \iota)$$

mit folgenden Eigenschaften:

- Q ist eine endliche Menge von Zuständen.

- $V = V_i \cup V_l \cup V_o$ ist eine endliche Menge von Variablen, die paarweise disjunkt in die Menge V_i der Eingabevariablen, die Menge V_l der lokale Variablen und in die Menge V_o der Ausgangsvariablen zerfällt.
- $\phi \subseteq Q \times TERM_\beta(V) \times (TERM_\alpha(V_l \cup V_o, V))^\star \times Q$ ist eine deterministische Transitionsrelation.
- $q_0 \in Q$ ist der Anfangszustand.
- $\gamma_0 \in \Theta_{lo}$, d.h. $\gamma_o : (V_l \cup V_o) \to \mathbb{N} \cup \mathbb{B}$ mit $\gamma_0(v) \in \vartheta(v)$ für alle $v \in V_l \cup V_o$, ist die initiale Anfangsbelegung.
- $\varepsilon \in \mathbb{R}$ mit $\varepsilon > 0$ gibt die obere Schranke für die Zyklusdauer an.
- $\iota : Q \to TERM_\beta(V_i) \times \mathbb{R}_{\geq 0}$ ist eine Beschriftungsfunktion, die jedem Zustand $q \in Q$ mit $\iota(q) = (b_q, t_q)$ eine Verzögerungszeit t_q zuordnet, in der die Belegungen der Eingangsvariablen, die die Bedingung b_q erfüllen, ignoriert werden.

□

Auf diese Definition wird das von den SPS$_\mathcal{H}$-Automaten bekannte Hierarchiekonzept übertragen, um so die im Editorsystem benutzten hierarchischen SPS$_V$-Automaten zu erhalten. Hierzu wird die Menge von Superzuständen und der Hierarchiebaum dem Automaten hinzugefügt und die Beschriftungsfunktion der Zustände auf die Superzustände erweitert.

Definition A.2: Ein *hierarchischer SPS$_V$-Automat* ist ein Tupel

hierarchischer SPS$_V$-Automat

$$\mathcal{A} = (Q, S, V, \phi, q_0, \gamma_0, \varepsilon, T, \iota)$$

mit folgenden Eigenschaften:

- für $Q, V, \phi, q_0, \gamma_0$ und ε gelten die Eigenschaften aus Definition A.1.
- S ist endliche Menge von *Superzuständen*, die auch als Hierarchieknoten bezeichnet werden.
- $T = (\{r\} \cup S \cup Q, E)$ ist ein (Hierarchie-) Baum mit dem Wurzelknoten $r \notin (S \cup Q)$, den Elementen von S als innere Knoten und mit den Elementen von Q als Blätter.
- $\iota : Q \cup S \to TERM_\beta(V_i) \times \mathbb{R}_{\geq 0}$ ist eine Beschriftungsfunktion, die jedem (Super-) Zustand $s \in Q \cup S$ mit $\iota(s) = (b_s, t_s)$ eine Verzögerungszeit t_s zuordnet, in der die Belegungen der Eingangsvariablen, die die Bedingung b_s erfüllen, ignoriert werden.

□

Anhang B

Syntax und Semantik von Timed Automata

Timed Automata sind ein automaten-basiertes mathematisches Modell für Realzeitsysteme. Die hier dargestellte Syntax und Semantik wurde auch bereits in [DFMV98] als Grundlage für die semantische Betrachtung von SPS-Automaten benutzt und entstammt [MY96].

Definition B.1: Ein Tupel $\mathcal{T} = (\mathcal{S}, \mathcal{X}, \mathcal{L}, \mathcal{E}, \mathcal{I}, \mathcal{P}, \mu, S_0)$ ist ein *Timed Automaton*, falls gilt:

Timed Automaton

- $\mathcal{S}$ ist eine endliche Menge von *Stellen*
- $\mathcal{X}$ ist eine endliche Menge von Variablen mit reellem Wertebereich, die *Uhren* (clocks) genannt werden und deren Wert gleichförmig mit der Zeit zunimmt.
- $\mathcal{L}$ ist eine endliche Menge von *Ereignissen*.
- $\mathcal{E}$ ist eine endliche Menge von *Kanten* der Form $e = (s, L, \phi, \rho, s')$ mit $s, s' \in \mathcal{S}$, $L \in \mathcal{L}$, ϕ ist eine *Zeitbedingung*, erzeugt von der Grammatik

  $$\phi ::= x + c \leq d \mid c \leq x + d \mid x + c \leq y + d \mid \neg\phi \mid \phi_1 \wedge \phi_2$$

  mit $x, y \in \mathcal{X}$ und $c, d \in \mathbb{R}$, und $\rho \subseteq \mathcal{X}$ ist die Menge der Uhren, die durch die Transition auf 0 zurückgesetzt werden.
- $\mathcal{I}$ ordnet jeder Stelle eine Bedingung zu, die als *Invariante* dieser Stelle fungiert.
- $\mathcal{P}$ ist eine endliche Menge von *Ausgaben*.
- μ ordnet jeder Stelle eine Menge von Ausgaben aus $\mathcal{P}$ zu.
- $S_0 \subseteq \mathcal{S}$ ist die Menge *initialer Stellen*

□

Üblicherweise werden nur natürliche Zahlen als Konstanten in den Zeitbedingungen zugelassen, aber um den SPS-Automaten Timed Automata zuordnen zu können, erlaubt diese Definition reelle Werte. Der Preis, der hierfür bezahlt werden muß, ist, daß mit einem solchen Modell kein Modelchecking mehr möglich ist. Solange jedoch der SPS-Automat nur diskrete Verzögerungszeiten und diskrete Zykluszeiten benutzt, verwendet auch die korrespondierende TA-Semantik nur diskrete Konstanten.

Lauf

Definition B.2: Ein *Lauf* eines Timed Automaton $\mathcal{T}$ ist eine unendliche Folge $r = ((s_i, v_i, t_i))_{i \in \mathbb{N}}$ wobei für jedes $i \in \mathbb{N}$ gilt:

- $s_i \in \mathcal{S}$ ist eine Stelle,
- $v_i \in \mathcal{X} \longrightarrow \mathbb{R}_{\geq 0}$ ist eine Belegung der Uhren,
- $t_i \in \mathbb{R}_{\geq 0}$ ist eine Zeitstempel

und r erfüllt die folgenden Eigenschaften:

- Die erste Stelle ist in der Menge initialer Stellen S_0 enthalten: $s_0 \in S_0$
- Zu Beginn haben alle Uhren den Wert 0: $\forall x \in \mathcal{X} : v_0(x) = 0$
- Die Zeit beginnt mit 0: $t_0 = 0$
- Die Folge von Zeitstempeln ist monoton und divergiert: $\forall i \in \mathbb{N} : t_i \leq t_{i+1}$ und $\lim_{i \to \infty} t_i = \infty$
- Für alle $i \in \mathbb{N}$ ist die Invariante $\mathcal{I}(s_i)$ während des gesamten Zeitintervalls $[t_i, t_{i+1}]$ erfüllt:

$$\forall 0 \leq t \leq (t_{i+1} - t_i) : \mathcal{I}(s_i)(v_i + t)$$

  mit $(v_i + t)(x) \stackrel{df}{=} v_i(x) + t$ für alle $x \in \mathcal{X}$ und $\mathcal{I}(s)(v)$ beschreibt die Auswertung der Bedingung $\mathcal{I}(s)$ unter der Belegung v der Uhren.
- Für alle $i \in \mathbb{N}$ existiert eine Kante $e = (s_i, L, \phi, \rho, s_{i+1})$ so daß:
  - die Zeitbedingung ϕ zum Zeitpunkt t_{i+1} erfüllt ist: $\phi(v_i + t_{i+1} - t_i)$, und
  - die Belegung der Uhren v_i wird gemäß ρ aktualisiert:

$$\forall x \in \mathcal{X} : v_{i+1}(x) = \begin{cases} 0 & \text{falls } x \in \rho \\ v_i(x) + t_{i+1} - t_i & \text{falls } x \notin \rho \end{cases}$$

Mit $\mathcal{R}(\mathcal{T})$ wird die Menge der möglichen Läufe eines Timed Automaton $\mathcal{T}$ bezeichnet. □

Anhang C

Grammatik der Protokollsprache

C.1 Start

<Start>	::=	<DefinitionList> <TracePart>
	|	<TracePart>

C.2 Definitionen

<DefinitionList>	::=	<DefinitionList> <Definition>
	|	<Definition>
<Definition>	::=	<BreakpointDef> ";"
	|	<OptionDef> ";"
	|	<VectorDef> ";"
<BreakpointDef>	::=	"`BREAKPOINT`" <Condition>
		"`SINCE`" <TimeExpr> <BreakAction>
	|	"`BREAKPOINT`" <Condition> <BreakAction>
<BreakAction>	::=	"`STOP`"
	|	"`RESTART`"
	|	"`REPORT`" *STRING* "`STOP`"
	|	"`REPORT`" *STRING* "`RESTART`"
	|	"`REPORT`" *STRING*
<OptionDef>	::=	"`OPTION`" "`TimeResolution`" "`:=`" *NUMBER*
	|	"`OPTION`" "`TimeResolution`" "`:=`"
		NUMBER "`/`" *NUMBER*
	|	"`OPTION`" "`InputPolling`" "`:=`" *IDENT*

<VectorDef>	::=	“VECTOR” *IDENT* “:=” “(” VariableList “)”
<VariableList>	::=	<VariableList> “,” <FullVariableName>
	|	<FullVariableName>
<Condition>	::=	<boolExpr>

C.3 Die Trace

<TracePart>	::=	“TRACE” <TraceStatement>
<Trace>	::=	<Trace> <TraceStatement>
	|	<TraceStatement>
<TraceStatement>	::=	“{” <Trace> “}”
	|	<TimeStatement> “;”
	|	<DriverEntry> “;”
	|	<AssignEntry> “;”
	|	<CycleEntry> “;”
	|	<PollEntry> “;”
	|	<BreakEntry> “;”
	|	<CheckEntry> “;”
	|	“;”
<TimeStatement>	::=	“WAIT” <TimeExpr>
	|	“WAIT” <TimeInterval>
	|	“AT” <TimeExpr>
	|	“AT” <TimeInterval>
<TimeExpr>	::=	*NUMBER*
<TimeInterval>	::=	“[” <TimeExpr> “,” <TimeExpr> “]”
<DriverEntry>	::=	<VectorName> “<-” <VectorValue>
	|	<FullVariableName> “<-” <Value>
<AssignEntry>	::=	<VectorName> “:=” <VectorValue>
	|	<FullVariableName> “:=” <Value>
<VectorValue>	::=	“(” <VectorValueList> “)”
	|	<VectorName>

<VectorValueList>	::=	<VectorValueList> “,” <Value>
	|	<Value>
<CycleEntry>	::=	“CYCLE” <AutomatonName>
<PollEntry>	::=	“POLL” <VectorName>
	|	“POLL” <FullVariableName>
<BreakEntry>	::=	<BreakAction>
<CheckEntry>	::=	“AWAIT” <Condition> “ELSE” <BreakAction>

C.4 Ausdrücke

<Value>	::=	<boolExpr>
	|	<intExpr>
	|	<enumVarExpr>
<boolExpr>	::=	<boolExpr> “or” <boolAndExpr>
	|	<boolAndExpr>
<boolAndExpr>	::=	<boolAndExpr> “and” <boolNotExpr>
	|	<boolNotExpr>
<boolNotExpr>	::=	“not” <boolNotExpr>
	|	“(” <boolExpr> “)”
	|	<boolVarExpr>
	|	<compareExpr>
<compareExpr>	::=	<boolVarExpr> “=” <boolVarExpr>
	|	<boolVarExpr> “#” <boolVarExpr>
	|	<enumVarExpr> “=” <enumVarExpr>
	|	<enumVarExpr> “#” <enumVarExpr>
	|	<intExpr> “=” <intExpr>
	|	<intExpr> “#” <intExpr>
	|	<intExpr> “>” <intExpr>
	|	<intExpr> “>=” <intExpr>
	|	<intExpr> “<” <intExpr>
	|	<intExpr> “<=” <intExpr>
<boolVarExpr>	::=	<FullBoolVariableName>
	|	“true”
	|	“false”

<enumVarExpr>	::=	<FullEnumVariableName>
	|	“'” *IDENT* “.” *IDENT*
	|	*IDENT*
<intExpr>	::=	<intTerm>
	|	<intExpr> “+” <intTerm>
	|	<intExpr> “-” <intTerm>
<intTerm>	::=	<intFact>
	|	<intTerm> “*” <intFact>
	|	<intTerm> “/” <intFact>
	|	<intTerm> “%” <intFact>
<intFact>	::=	“(” <intExpr> “)”
	|	“-” <intFact>
	|	<intVarExpr>
<intVarExpr>	::=	<FullIntVariableName>
	|	*NUMBER*

C.5 Bezeichner

<FullVariableName>	::=	<FullBoolVariableName>
	|	<FullIntVariableName>
	|	<FullEnumVariableName>
<FullBoolVariableName>	::=	<AutomatonName> “.” <BoolVariableName>
<FullIntVariableName>	::=	<AutomatonName> “.” <IntVariableName>
<FullEnumVariableName>	::=	<AutomatonName> “.” <EnumVariableName>
<AutomatonName>	::=	*IDENT*
<BoolVariableName>	::=	*BOOLVAR_IDENT*
<IntVariableName>	::=	*INTVAR_IDENT*
<EnumVariableName>	::=	*ENUMVAR_IDENT*
<VectorName>	::=	*VECTOR_IDENT*

Anhang D

Implementierte Klassen (Ableitungshierarchie)

Programme der MOBY-Reihe benutzen `MObjekt` als Basisklasse aller Objekte, die im Programm enthalten sind. Die Klasse `Class` erlaubt es, einen Hierarchiebaum aller Klassen durch die Methode `printHierarchy` auszugeben, die von der entsprechenden Klasse abgeleitet sind. Demzufolge liefert der Aufruf `MObject::getClass()->printHierarchy(out,0);` einen Hierarchiebaum aller Klassen, die im aktuellen Programm eingebunden sind. Diese Ausgabe wurde (manuell) auf die veränderten und neuen Klassen dieser Diplomarbeit reduziert (ausgenommen sind Basisklassen).

```
MObject
   -> Application
         -> PLCApplication
   -> Collection
         -> SequenceableCollection
               -> OrderedCollection
                     -> PLCSimTrace
                     -> ValueSet
                           -> PLCValueSet
         -> Set
               -> Dictionary
                     -> PLCInputObjects
                     -> PLCTraceOptions
   -> Compiler
         -> PLCTraceCompiler
   -> CompilerError
   -> Controller
         -> GDiagramController
               -> GCurveController
```

```
-> DisplayMedium
      -> Visible
            -> FormWrapper
                  -> TabbedBrowser
            -> View
                  -> CanvasView
                        -> GDiagramView
                              -> GCurveView
                                    -> GGanttView
-> Event
      -> PLCEvent
            -> PLCBreakEvent
            -> PLCInputEvent
                  -> PLCAssignEvent
                  -> PLCDriverEvent
                  -> PLCPollEvent
            -> PLCStateEvent
                  -> PLCCycleEvent
            -> PLCTraceEnd
-> GDiagramParameter
-> GExpressionData
-> GScaledObject
-> Model
      -> Graph
            -> GDiagram
                  -> GCurve
                        -> GGantt
-> PLCBreakpoint
-> PLCEventReporter
-> PLCMsgReceiver
-> PLCSimAutInfo
-> PLCSimOptions
-> PLCSimulator
      -> PLCbgSimulator
-> PLCTExpression
-> PLCTMacro
-> PLCTObjectError
-> PLCTSemanticObj
-> PLCTraceContext
-> PLCTraceEntry
      -> PLCBreakEntry
      -> PLCCheckEntry
      -> PLCCycleEntry
```

```
        -> PLCTimeEntry
        -> PLCVarEntry
-> PLCVarDescription
-> Scheduler
        -> PLCScheduler
                -> PLCbgScheduler
-> TValueObject
        -> TValue
```

Literaturverzeichnis

[AAFT97] Peter Amthor, Nils Arndt, Hans Fleischhack, and Josef Tapken. The Moby Class Library. FB Informatik, Univerität Oldenburg, 1997.

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[Arn97] Nils Arndt. Erweiterung einer SDL-Entwicklungsumgebung um Komponenten zur Prozeßkommunikation und Simulation – Entwurf und Implementierung. Diplomarbeit, FB Informatik, Univerität Oldenburg, 1997.

[Ash90] Peter J. Ashenden. *The VHDL Cookbook*. Dept. Computer Science, University of Adelaide, first edition, July 1990.

[Bar98] Andreas Bartelt. Entwurf und Implementierung einer Java-Basisklassenbibliothek für MOBY. Diplomarbeit, FB Informatik, Univerität Oldenburg, April 1998.

[BFvG+97] Sven Brokmann, Hans Fleischhack, Jens von Garrel, René Hußmann, Andy Muthmann, Michael Möller, Marco Oetken, Andreas Schulze, Josef Tapken, and Christian Wachtendorf. Endbericht der Projektgruppe IEKOΣ. Interner Bericht, FB Informatik, Univerität Oldenburg, 1997.

[BHS91] Ference Belina, Dieter Hogrefe, and Amardeo Sarma. *SDL with Applications from Protocol Specification*. Prentice Hall International, Hertfordshire UK, 1991.

[DFMV98] H. Dierks, A. Fehnker, A. Mader, and F.W. Vaandrager. Operational and Logical Semantics for Polling Real-Time Systems. Technical Report CSI-R9813, Computer Science Institue Nijmegen, Faculty of Mathematics and Informatics, Catholic University of Nijmegen, April 1998.

[Die96] Cheryl Dietz. Graphical Formalisation of Real-Time Requirements. Technical report, Fachbereich Informatik Universität Oldenburg, 1996.

[Die97] H. Dierks. PLC-Automata: A New Class of Implementable Real-Time Automata. In M. Bertran and T. Rus, editors, *ARTS'97*, volume 1231 of *Lecture Notes in Computer Science*, pages 111–125. Springer-Verlag, May 1997.

[DT98] H. Dierks and J. Tapken. Tool-Supported Hierarchical Design of Distributed Real-Time Systems. In *Euromicro Workshop on Real Time Systems*, pages 222–229. IEEE, 1998.

[Gao97] Xia Gao. Entwurf und Implementierung einer MSC-Komponente zur Visualisierung der Simulation von μSDL-Spezifikationen. Diplomarbeit, FB Informatik, Univerität Oldenburg, Juli 1997.

[IEE87] IEEE. *IEEE Standard VHDL Language Reference Manual*, 1987.

[JW97] S. Jähnichen and M. Weber. Integration mathematisch-formaler Techniken in die Entwicklung eingebetteter Systeme. *it+ti*, 3:7–14, June 97.

[KBPO+96] B. Krieg-Brückner, J. Peleska, E.-R. Olderog, D. Balzer, and A. Baer. UniForM - Universal Formal Methods Workbench. In U. Grote and G. Wolf, editors, *Statusseminar des BMBF Softwaretechnologie*, pages 357–378, Berlin, March 1996. BMBF.

[Mau96] S. Mauw. The formalization of Message Sequence Charts. In O. Haugen, editor, *Computer Networks and ISDN Systems - SDL and MSC*, volume 28 (1996), Number 12, June 1996.

[Mit82] I. Mitrani. *Simulation techniques for discrete event systems*. Cambridge University Press, 1982.

[MY96] O. Maler and S. Yovine. Hardware Timing Verification using Kronos. In *Proc. 7th Conf. on Computer-based Systems and Software Engineering*. IEEE Press, 1996.

[Oet98] Marco Oetken. Implementierung von Constraint-Diagrammen. Diplomarbeit, FB Informatik, Univerität Oldenburg, Oktober 1998.

[OY93] Alfredo Olivero and Sergio Yovine. *Kronos: A Tool for Verifying Real–time Systems*. VERIMAG, draft 0.0 edition, 24. August 1993.

[RGG96] Ekkart Rudolph, Peter Graubmann, and Jens Grabowski. Tutorial on Message Sequence Charts. In O. Haugen, editor, *Computer Networks and ISDN Systems - SDL and MSC*, volume 28 (1996), Number 12, June 1996.

[SBS+95] M. Sonnenschein, U. Bartels, St. Schöf, R. Wieting, N. Enderlein, Ch. Gallar, D. Markenke, St. Martin, M. Pensler B. Mielenhausen, H.-P. Pohle, R. Radetke an G. Reents, and W. Tholen. Zwischenbericht der Projektgruppe IDEFIX, „Umgebung zur Simulation mittels THOR-Netzen“. Interner Bericht PSS 95 / 2, FB Informatik, Univerität Oldenburg, Mai 1995.

[SBS+96] M. Sonnenschein, U. Bartels, St. Schöf, R. Wieting, N. Enderlein, Ch. Gallar, D. Markenke, St. Martin, M. Pensler B. Mielenhausen, H.-P. Pohle, R. Radetke an G. Reents, and W. Tholen. Abschlußbericht der Projektgruppe IDEFIX, „Umgebung zur Simulation mittels THOR-Netzen“. Interner Bericht PSS 95 / 2, FB Informatik, Univerität Oldenburg, März 1996.

[Sch97] Johannes Schröder. Entwurf und Implementierung eines kompilativen interaktiven Simulatorsystems für SPS-Automaten. Diplomarbeit, FB Informatik, Univerität Oldenburg, Juli 1997.

[Sei93] Andree Seidel. Entwurf und Implementierung eines Editors für elementare Petriboxen. Diplomarbeit, FB Informatik, Univerität Oldenburg, 1993.

[Tap95] Josef Tapken. Entwurf und Implementierung eines Systems zur Verifikation von SDL-Spezifikationen. Diplomarbeit, FB Informatik, Univerität Oldenburg, 1995.

[Tap97] J. Tapken. Interactive and Compilative Simulation of PLC-Automata. In W. Hahn and A. Lehmann, editors, *Simulation in Industry*, ESS'97, pages 552–556. SCS, 1997.

[Zho93] Zhou Chaochen. Duration calculi: An overview. In D. Bjorner, M. Broy, and I.V. Pottosin, editors, *Formal Methods in Programming and Their Application*, volume 735 of *Lecture Notes in Computer Science*, pages 256–266. Springer-Verlag, 1993.

[ZHR91] Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A Calculus of Durations. *Information Processing Letters*, 40/5:269–276, 1991.

Index

Ich versichere, die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfaßt zu haben.

Oldenburg, den 16. Juli 1999

Michael Möller