Technische Universität Kaiserslautern

Eine Oberfläche zur Simulation synchroner Sprachen

Projektarbeit

Technische Universität Kaiserslautern

Arbeitsgruppe Reaktive Systeme

Manuel Gesell, Mike Gemünde

November 2007

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Kaiserslautern, den 15.11.2007

Manuel Gesell

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Kaiserslautern, den 15.11.2007

Mike Gemünde

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation . . . 1
- 1.2 Ziel . . . 1
- 1.3 Vorgehen . . . 2

2 Synchrone Sprachen, Quartz **3**
- 2.1 Zustands- und Ereignisvariablen . . . 4
- 2.2 Anweisungen . . . 4
- 2.3 Module . . . 6
- 2.4 Semantik . . . 7
- 2.5 Logische Korrektheit . . . 11
- 2.6 Zyklische und azyklische Abhängigkeiten . . . 12

3 Kausalität und Simulation **13**
- 3.1 Kausalitätsanalyse . . . 13
- 3.2 Simulator . . . 17
- 3.3 Ausblick . . . 18

4 Value Change Dump **21**

5 Averest **23**
- 5.1 Übersetzung - Der Compiler Ruby . . . 23
- 5.2 Verifikation - Der Modelprüfer Beryl . . . 24
- 5.3 Codeerzeugung - Das Synthese-Werkzeug Topaz . . . 24
- 5.4 Averest Eclipse-Plugin . . . 24

6 Die Simulator-Oberfläche **25**
- 6.1 Benutzerschnittstelle . . . 25
  - 6.1.1 Funktionen . . . 25
  - 6.1.2 Die Simulationsansicht . . . 26
  - 6.1.3 Das Einstellungsfenster . . . 28
- 6.2 Funktionaler Aufbau . . . 29
  - 6.2.1 Kommunikation mit dem Simulator . . . 30
  - 6.2.2 Simulationsablauf . . . 32

6.2.3 Die Werte der Variablen im Speicher 33
6.2.4 Hauptkomponenten 36
6.2.5 Weitere Klassen 40
6.2.6 Verhalten der Oberfläche 40

7 Zusammenfassung 43
7.1 Zusammenfassung 43
7.2 Ausblick 43

Kapitel 1

Einleitung

1.1 Motivation

Eingebette Systemen finden heutzutage Anwendung in vielen Bereichen des Lebens. Sie stecken in einfachen Haushaltsgeräten wie der Mikrowelle oder dem Radio, sie steuern aber auch das ABS unseres Autos, stabilisieren Flugzeuge und unterstützen Ärzte bei Operationen. Wenn das Radio mal einen kleinen Aussetzer hat, ist das meist nicht tragisch. Ein kleiner Aussetzer kann aber für das Automobil oder das Flugzeug schon der eine viel sein. Die Verifikation spielt deshalb im sicherheitskritischen Bereich eine entscheidende Rolle. Man möchte Fehler so früh wie möglich erkennen und diese beseitigen. Hier kommen die synchronen Sprachen auf den Plan, die Eigenschaften besitzen, um diesen Anforderungen gerecht zu werden. Leider ist deswegen aber die Semantik, für den mit der imperativen Programmierung erfahrenen Entwickler, nicht unbedingt intuitiv.

1.2 Ziel

Die Verhaltensweise eines synchronen Programms lässt sich auch mittels der Verifikation herausfinden. Die Verifikation setzt aber vorraus, dass man zumindest eine Vorstellung über das Verhalten hat, oder es genau kennt. Wir möchten hier ein Werkzeug mit an die Hand geben, das es ermöglicht, sich das Verhalten von synchronen Programmen anzusehen und zu studieren. Dabei soll der Benutzer verschiedene Verhaltensweisen in verschiedenen Situationen ausprobieren können und sehen wie das Programm auf die verschiedensten Eingaben reagiert.

Der Simulator kann aber auch zur Fehlersuche eingesetzt werden, indem man das erwartete Verhalten Schritt für Schritt überprüfen kann und die Stelle des Fehlers so besser finden kann.

1.3 Vorgehen

Die ersten Kapitel beschäftigen sich mit den Grundlagen zu synchronen Sprachen, deren Simulation und allem, was noch wichtig für die Oberfläche ist. Im folgenden Kapitel werden wir auf die synchrone Sprache Quartz eingehen und deren Semantik erklären. Das Kapitel 3 beschäftigt sich mit der Kausalität von Quartz und wie man damit Quartz-Programme simulieren kann. Kapitel 4 gibt einen kurzen Einstieg in das VCD-Format, für das wir eine Export-Funktion in den Simulator eingebaut haben. Im Kapitel 5 werden wir Averest und die darin enthaltenen Werkzeuge kurz beschreiben.

Im Kapitel 6 werden wir schließlich auf den Simulator eingehen. Dabei wird zuerst das Verhalten des Simulators aus der Benutzersicht beschrieben und erklärt. Anschließend gehen wir auf den internen Aufbau ein. Dabei beschäftigen wir uns damit, wie die Oberfläche mit dem eigentlichen Simulator kommuniziert und wie das Verhalten der Oberfläche realisiert ist.

Die Einleitung (Kapitel 1), die Grundlagenkapitel (2, 3, 4, 5) und die Zusammenfassung (Kapitel 7) wurden gemeinsam erarbeitet und geschrieben. Das Kapitel 6.2 wurde von Mike Gemünde erstellt. Das Kapitel 6.1 wurde von Manuel Gesell erstellt und er war auch an den Kapiteln 6.2.4 und 6.2.5 beteiligt.

Kapitel 2

Synchrone Sprachen, Quartz

Synchrone Sprachen werden hauptsächlich zur Beschreibung von reaktiven Systemen verwendet. Dabei können synchrone Sprachen eingesetzt werden, um die Hardware als auch die Software zu beschreiben. Erst gegen Ende der Entwicklung ist die Entscheidung zu treffen, welcher Teil zu Hardware und welcher zu Software übersetzt werden soll. Weiterhin sind sie aufgrund ihrer formalen Semantik besonders zur Verifikation geeignet, wodurch sie für sicherheitskritische Anwendungen besonders interessant werden. [1]

Synchrone Sprachen basieren auf dem Paradigma der perfekten Synchronie. Diese lässt sich anhand von Mikro- und Makroschritten wie sie in Quartz oder Esterel vorkommen erklären. Die meisten Anweisungen werden als Mikroschritte ausgeführt und benötigen keine Zeit. Zeitverbrauch wird explizit durch die Anweisung **pause** gekennzeichnet, wodurch Mikroschritte zu Makroschritten zusammengefasst werden. Makroschritte benötigen zur Ausführung genau eine Zeiteinheit (einen Schritt). Dadurch werden gleichzeitig laufende Threads automatisch nach jedem Makroschritt synchronisiert.

Ein Vorteil dieser Semantik ist, dass Programme sehr einfach in Hardware übersetzt werden können. Denn die Nebenläufigkeit, wie sie zwischen den Mikroschritten auftritt, ist in Schaltkreisen allgegenwärtig. Auf der anderen Seite können durch die Makroschritte aus synchronen Multithread-Programmen deterministische Singlethread-Programme gemacht werden, die auf einfachen Mikrokontrollern lauffähig sind und keine komplexen Betriebssysteme benötigen.

Die synchrone Sprache Quartz ist von Esterel abgeleitet. Wir werden uns im Folgenden, da das Averest-Framework mit dieser Sprache arbeitet, auf Quartz beziehen und deren Semantik anhand einiger Befehle und anhand von Übergangsrelationen erklären. Die nachfolgenden Unterkapitel basieren auf [6], [7] und [2]. In letzterem ist auch eine ausführliche Einführung zur Programmierung mit synchronen Sprachen zu finden.

2.1 Zustands- und Ereignisvariablen

In der Sprache Quartz wird prinzipiell zwischen Zustands- und Ereignisvariablen unterschieden. Zustandsvariablen haben hier das selbe Verhalten wie Variablen aus anderen Programmiersprachen, z.B. C. Wenn wir einer Zustandsvariablen einen Wert zuweisen, so behält sie diesen bis zur nächsten Zuweisung. Ereignisse dagegen sind immer nur in dem Schritt gültig, in dem sie *emittiert* wurden, bzw. müssen für jeden Schritt, in dem sie gültig sein sollen, explizit gesetzt werden, ansonsten haben sie im entsprechenden Schritt ihren Standardwert. Zustandsvariablen und Ereignisse werden anhand ihrer Deklaration unterschieden. Deklarationen und Wertzuweisungen können z.B. folgendermaßen aussehen:

Zustandsvariablen:	Ereignisvariablen:
int x;	**event int** x;
bool $a,\ b$;	**event bool** $a,\ b$;
$x\ =\ 7$;	$x\ =\ 7$;
next$(a)\ =$ **true**;	**next**$(a)\ =$ **true**;
	emit $x\ =\ 7$;

Dabei bewirkt das Schlüsselwort **next**, dass die Wertzuweisung erst für den nächsten Makroschritt gilt. Da Zuweisungen Mikroschritte sind, können mehrere Zuweisungen gleichzeitig ausgeführt werden. Es können Schreibkonflikte entstehen, wenn versucht wird einer Variablen in einem Makroschritt verschiedene Werte zuzuweisen. Im Folgenden wird auf das Schlüsselwort **emit** verzichtet, da es aus dem Kontext ersichtlich ist, ob Ereignis- oder Zustandsvariablen gemeint sind.

2.2 Anweisungen

Wir haben vorher erwähnt, dass die Anweisung **pause** einen Makroschritt kennzeichnet. In Quartz gibt es allerdings noch andere Anweisungen, die Zeit verbrauchen. Im Folgenden werden wir vor allem die Basisanweisungen beschreiben, aus denen sich noch andere konstruieren lassen.

- **pause** ;

  Der Kontrollfluss bleibt hier bis zum nächsten Schritt stehen. **pause** ist eine der Anweisungen, mit der Makroschritte bestimmt werden.

- **await** b ;

  Diese Anweisung verbraucht mindestens einen Schritt und definiert somit wie **pause** auch das Ende eines Makroschritts. Nach dem ersten Schritt wird die Bedingung b geprüft. Wenn sie gilt, wird der Kontrollfluss fortgesetzt, ansonsten bleibt der Kontrollfluss bis zum nächsten Schritt hier stehen und b wird erneut geprüft. Eine Abwandlung dieses Befehls ist **await immediate** b, bei dem die Gültigkeit von b schon im ersten Schritt überprüft wird.

- S1 ; S2

  Benötigt S1 zur Ausführung keine Zeit, so wird die Ausführung von S2 im selben Schritt gestartet. Ansonsten wird S2 erst in dem Schritt gestartet, in dem S1 terminiert. Dieses Konstrukt macht es zum Beispiel möglich durch x = 5; y = 8 den beiden Variablen x und y im selben Schritt einen Wert zuzuweisen.

- S1 || S2

  Die beiden Anweisungen S1 und S2 werden gleichzeitig gestartet und nach jedem Makroschritt synchronisiert. Sollten beide Anweisungen direkt terminieren, so terminiert die gesamte Anweisung direkt. Andernfalls wird Zeit verbraucht. Terminiert eine der beiden Anweisungen in einem Schritt, so bleibt der Kontrollfluss so lange stehen, bis auch die andere Anweisung terminiert. Die Ausführung nach S1 || S2 wird also erst fortgesetzt, wenn S1 und S2 beendet sind.

- **if** (b) {S1} **else** {S2}

  Die **if**-Anweisung entscheidet anhand der Bedingung b, welche der beiden Anweisungen S1 oder S2 ausgeführt wird.

- **loop** { S };

  loop definiert eine Schleife, die S immer wieder ausführt. Die Anweisung S muss immer Zeit verbrauchen. Wenn S keine Zeit verbrauchen würde, würden unendlich viele Mikroschritte (die Schleife immer wieder) gleichzeitig ausgeführt werden.

- **abort** S **when** (b);

  S wird direkt, ohne b zu prüfen, ausgeführt. Falls S direkt terminiert, terminiert die gesamte Anweisung. Andernfalls wird ab dem zweiten Schritt (und in jedem evtl. folgenden) die Bedingung b geprüft. Wenn b gilt, wird der Ausdruck sofort abgebrochen. Es gibt zwei Erweiterungen zu dieser Anweisung. Die eine ist **weak abort**. Hier wird nach Auftreten von b der aktuelle Schritt von S noch ausgeführt. Die andere ist **when immediate**, bei der die Bedingung b schon im ersten Schritt geprüft wird. Beide Varianten können auch gleichzeitig auftreten.

2.3 Module

Ein Quartz-Programm besteht aus einer Liste von Modulen. Module haben Namen und deklarieren Ein- und Ausgaben, wobei Ausgaben an einem &-Zeichen zwischen dem Typ und dem Variablennamen zu erkennen sind. Der Rumpf eines Moduls enthält Anweisungen und Aufrufe anderer Module.

Beispiel

Wir wollen nun folgendes Beispiel näher betrachten:

```
module ABRO(event bool a, event bool b, event bool r,
         event bool &o)
{
   loop
      abort {
         await(a); || await(b);

         emit o;
         await(r);
      } when(r);
}
```

Das Modul trägt den Namen ABRO. Die Ereignisvariablen a, b und r vom Typ **bool** sind Eingaben. Die Ereignisvariable o vom Typ **bool** ist eine Ausgabe. Das Programm wartet bis die beiden Ereignisse a und b eintreten. Hierbei ist es egal, welches zuerst eintritt oder ob eines erst später eintritt als das andere. Erst wenn beide (mindestens) einmal gültig waren wird der Programmfluss nach **await**(a); || **await**(b); fortgesetzt. Der Befehl **emit** o; gibt die Ausgabe im gleichen Schritt aus, indem das zweite Ereignis (a oder b) eingetreten ist. Mit der Ereignisvariablen r wird die **abort** Anweisung beendet und durch die umliegende Schleife wird das Programm neu gestartet. Die Anweisung **await**(r); ist nötig, um den Programmfluss anzuhalten, da ansonsten, ohne Auftreten von

r, **abort** beendet werden würde. Es ist noch zu beachten, dass die beiden Ereignisse a und b erst nach dem ersten Schritt geprüft werden.

2.4 Semantik

Bisher haben wir die Sprache Quartz mit einigen Anweisungen informell kennengelernt. Wir möchten nun die Semantik auf eine formalere Grundlage stellen, und diese anhand von Transitionsregeln erklären. Dazu benötigen wir zusätzlich den Begriff der *Umgebung*.

Definition: Umgebung

Eine Umgebung $\mathcal{E}$ ist eine Funktion, die Variablen auf Werte abbildet. Die Umgebung ist für einen Makroschritt konstant, denn alle Variablen können in einem Schritt nur einen Wert haben. Die Funktion $\mathcal{E}$ wird ebenfalls dazu benutzt um Ausdrücke auszuwerten. Zum Beispiel ist $\mathcal{E}(x+y) = 13$ wenn $\mathcal{E}(x) = 6$ und $\mathcal{E}(y) = 7$ gilt.

Definition: Transitionsregeln

Die Semantik kann durch Transitionsregeln dargestellt werden. Wir schreiben Transitionsregeln in der Form:

$$S \xrightarrow[\mathcal{E}]{\mathcal{D},b} S'$$

mit den folgenden Bestandteilen:

- $\mathcal{E}$ ist die aktuelle Umgebung
- $\mathcal{D}$ die Menge von Aktionen, die von S ausgeführt werden
- $b \in \{\mathbf{true}, \mathbf{false}\}$ markiert eine zeitlose Ausführung
  ($b = \mathbf{true} \Leftrightarrow$ Ausführung von S ist zeitlos)

Die Transitionsregel bedeutet, dass wenn die Anweisung S in der Umgebung $\mathcal{E}$ gestartet wird, werden die Aktionen $\mathcal{D}$ direkt ausgeführt. b gibt an, ob S' im nächsten Makrostep weiter ausgeführt werden muss. Ansonsten gilt $b = \mathbf{false}$ und $S' = \mathbf{nothing}$.

Folgende Transitionsregeln, die auch SOS-Regeln genannt werden, gelten für die oben aufgeführten Befehle:

$$\textbf{nothing} \xrightarrow[\mathcal{E}]{\{\},\textbf{true}} \textbf{nothing}$$
$$\textbf{pause} \xrightarrow[\mathcal{E}]{\{\},\textbf{false}} \textbf{nothing}$$
$$x := \tau \xrightarrow[\mathcal{E}]{\{x:=\tau\},\textbf{true}} \textbf{nothing}$$
$$\textbf{next}(x) := \tau \xrightarrow[\mathcal{E}]{\{next(x):=\tau\},\textbf{true}} \textbf{nothing}$$

Die Transitionsregeln sind sehr leicht zu verstehen. Die **pause** Anweisung ist die einzige, die Zeit verbraucht, also auch die einzige, für die $b = \textbf{true}$ gilt. Die folgenden Anweisungen setzen sich aus anderen Anweisungen zusammen und hängen somit von dem Verhalten der einzelnen Teile ab. Zum Beispiel hängt die Terminierung von S1; S2 im aktuellen Schritt von der Terminierung von S1 *und* der Terminierung von S2 ab.

- S1 ; S2

$$\frac{S1 \xrightarrow[\mathcal{E}]{\mathcal{D}_1,\textbf{true}} S1' \qquad S2 \xrightarrow[\mathcal{E}]{\mathcal{D}_2,b_2} S2'}{S1;S2 \xrightarrow[\mathcal{E}]{\mathcal{D}_1\cup\mathcal{D}_2,b_2} S2'}$$

$$\frac{S1 \xrightarrow[\mathcal{E}]{\mathcal{D}_1,\textbf{false}} S1' \qquad S2 \xrightarrow[\mathcal{E}]{\mathcal{D}_2,b_2} S2'}{S1;S2 \xrightarrow[\mathcal{E}]{\mathcal{D}_1,\textbf{false}} S1';S2}$$

- **if** (b) {S1} **else** {S2}

$$\frac{S1 \xrightarrow[\mathcal{E}]{\mathcal{D}_1,b_2} S1' \qquad \mathcal{E}(\sigma) = \textbf{true}}{\textbf{if}\ (\sigma)\ S1\ \textbf{else}\ S2 \xrightarrow[\mathcal{E}]{\mathcal{D}_1,b_1} S1'}$$

$$\frac{S2 \xrightarrow[\mathcal{E}]{\mathcal{D}_2,b_2} S2' \qquad \mathcal{E}(\sigma) = \textbf{false}}{\textbf{if}\ (\sigma)\ S1\ \textbf{else}\ S2 \xrightarrow[\mathcal{E}]{\mathcal{D}_2,b_2} S2'}$$

- **abort** S **when** (b) ;

$$\frac{S \xrightarrow[\mathcal{E}]{\mathcal{D},\mathbf{true}} S'}{\mathbf{abort}\ \ S\ \ \mathbf{when}\ (\sigma)\ \xrightarrow[\mathcal{E}]{\mathcal{D},true}\ \mathbf{nothing}}$$

$$\frac{S \xrightarrow[\mathcal{E}]{\mathcal{D},\mathbf{false}} S'}{\mathbf{abort}\ \ S\ \ \mathbf{when}\ (\sigma)\ \xrightarrow[\mathcal{E}]{\mathcal{D},true}\ \mathbf{abort}\ \ S'\ \ \mathbf{when\ immediate}\ (\sigma)}$$

- **abort** S **when immediate** (b) ;

$$\frac{\mathcal{E}(\sigma)\ =\ \mathbf{true}}{\mathbf{abort}\ \ S\ \ \mathbf{when\ immediate}\ (\sigma)\ \xrightarrow[\mathcal{E}]{\{\},true}\ \mathbf{nothing}}$$

$$\frac{S \xrightarrow[\mathcal{E}]{\mathcal{D},\mathbf{true}} S', \qquad \mathcal{E}(\sigma)\ =\ \mathbf{false}}{\mathbf{abort}\ \ S\ \ \mathbf{when\ immediate}\ (\sigma)\ \xrightarrow[\mathcal{E}]{\mathcal{D},true}\ \mathbf{nothing}}$$

$$\frac{S \xrightarrow[\mathcal{E}]{\mathcal{D},\mathbf{false}} S', \qquad \mathcal{E}(\sigma)\ =\ \mathbf{false}}{\mathbf{abort}\ \ S\ \ \mathbf{when\ immediate}\ (\sigma)\ \xrightarrow[\mathcal{E}]{\mathcal{D},false}\ \mathbf{abort}\ \ S'\ \ \mathbf{when\ immediate}\ (\sigma)}$$

- S1 || S2

$$\frac{S1 \xrightarrow[\mathcal{E}]{\mathcal{D}_1,\mathbf{true}} S1' \qquad\qquad S2 \xrightarrow[\mathcal{E}]{\mathcal{D}_2,\mathbf{true}} S2'}{S1\ ||\ S2\ \xrightarrow[\mathcal{E}]{\mathcal{D}_1\cup\mathcal{D}_2,\mathbf{true}}\ \mathbf{nothing}}$$

$$\frac{S1 \xrightarrow[\mathcal{E}]{\mathcal{D}_1,\mathbf{false}} S1' \qquad\qquad S2 \xrightarrow[\mathcal{E}]{\mathcal{D}_2,\mathbf{true}} S2'}{S1\ ||\ S2\ \xrightarrow[\mathcal{E}]{\mathcal{D}_1\cup\mathcal{D}_2,\mathbf{false}}\ S1'}$$

$$\frac{S1 \xrightarrow[\mathcal{E}]{\mathcal{D}_1,\mathbf{true}} S1' \qquad S2 \xrightarrow[\mathcal{E}]{\mathcal{D}_2,\mathbf{false}} S2'}{S1 \mid\mid S2 \xrightarrow[\mathcal{E}]{\mathcal{D}_1 \cup \mathcal{D}_2,\mathbf{false}} S2'}$$

$$\frac{S1 \xrightarrow[\mathcal{E}]{\mathcal{D}_1,\mathbf{false}} S1' \qquad S2 \xrightarrow[\mathcal{E}]{\mathcal{D}_2,\mathbf{false}} S2'}{S1 \mid\mid S2 \xrightarrow[\mathcal{E}]{\mathcal{D}_1 \cup \mathcal{D}_2,\mathbf{false}} S1' \mid\mid S2'}$$

- **await** (b);

$$\mathbf{await}\ (\sigma) \xrightarrow[\mathcal{E}]{\{\},\mathbf{false}} \mathbf{await\ immediate}\ (\sigma)$$

- **await immediate** (b);

$$\frac{\mathcal{E}(\sigma) = \mathbf{true}}{\mathbf{await\ immediate}\ (\sigma) \xrightarrow[\mathcal{E}]{\mathcal{D},true} \mathbf{nothing}}$$

$$\frac{\mathcal{E}(\sigma) = \mathbf{false}}{\mathbf{await\ immediate}\ (\sigma) \xrightarrow[\mathcal{E}]{\{\},false} \mathbf{await\ immediate}\ (\sigma)}$$

Die Transitionsregeln geben uns also an, wie welche Anweisungen auszuführen sind. Die Anwendung dieser Regeln setzt allerdings eine Umgebung $\mathcal{E}$ voraus. Wir müssen nun noch die *Konsitenz* zwischen den ausgeführten Aktionen und unserer Umgebung prüfen, denn nicht jede Ausführung passt zu jeder Umgebung. Wenn $\mathcal{E}(x) = 3$ gilt, aber $\mathbf{x} = \mathbf{5} \in \mathcal{D}$ eine ausgeführte Aktion ist, ist recht einfach zu erkennen, dass eine solche Ausführung keinen Sinn ergibt. Für die Überprüfung benötigen wir zusätzlich noch die Menge der Aktionen $\mathcal{D}_{old}$ aus dem letzten Schritt, da dort eventuell Zuweisungen durch **next** getätigt wurden, die im aktuellen Schritt gültig sein sollen. Um den Wert der Variablen, die keine Zuweisung erfahren haben aus dem letzten Schritt zu übernehmen brauchen wir auch die Umgebung $\mathcal{E}_{old}$ von diesem Schritt. Eine Zustandsvariable ist in einem Schritt genau dann konsitent, wenn alle Wertzuweisungen an die Variable den gleichen Wert haben oder, falls keine Zuweisung an die Variable in $\mathcal{D}$ und $\mathcal{D}_{old}$ existiert, den Wert aus dem letzten Schritt hat. Für eine Ereignisvariable

gilt, das gleiche, außer, dass sie im Fall keiner Zuweisung ihren Standardwert annehmen muss.

Formal definieren wir für eine Variable x

$$\mathcal{D}_x := \{\mathcal{E}_{old}(\tau) \mid \mathbf{next}(x) := \tau \ \in \mathcal{D}_{old}\} \ \cup \ \{\mathcal{E}(\tau) \mid x := \tau \ \in \mathcal{D}\}$$

die Menge der Zuweisungen, die im aktuellen Schritt Gültigkeit haben. Für eine Zustandsvariable muss dann gelten:

$$\mathcal{D}_x \neq \emptyset \Rightarrow \forall \pi \in \mathcal{D}_x[\mathcal{E}(x) = \pi] \ \ \wedge \ \ \mathcal{D}_x = \emptyset \Rightarrow \mathcal{E}(x) = \mathcal{E}_{old}(x)$$

und für eine Ereignisvariable:

$$\mathcal{D}_x \neq \emptyset \Rightarrow \forall \pi \in \mathcal{D}_x[\mathcal{E}(x) = \pi] \ \ \wedge \ \ \mathcal{D}_x = \emptyset \Rightarrow \mathcal{E}(x) = \text{defaultValue}(x)$$

wobei wir für den ersten Schritt $\mathcal{E}_{old}(x) := \text{defaultValue}(x)$ und $\mathcal{D}_{old} := \{\}$ definieren.

Zu einer korrekten Programmausführung gelangen wir mit jeder Umgebung $\mathcal{E}$, die die Ausführung der Transitionsregeln ermöglicht und zu einem Konsistenten Zustand führt.

2.5 Logische Korrektheit

Nach obiger Überlegung können aber auch keine sowie mehrere Umgebungen existieren, die eine korrekte und konsistente Ausführung des Programms möglich machen. Da ein Programm, das für gleiche Eingaben verschiedene Ausgaben produzieren kann, keinen Sinn macht, wollen wir uns auf *logisch korrekte* Programme beschränken, für die es immer genau eine erfüllende Umgebung gibt. *Deterministische* Programme produzieren für jede Folge von Eingaben maximal eine Folge von Ausgaben. Für *reaktive* Programme gilt, dass sie für eine Folge von Eingaben mindestens eine Folge von Ausgaben produzieren. Die *logische Korrektheit* ist die Kombination von beiden Eigenschaften. Ein nicht-*deterministisches* Programm könnte zum Beispiel folgendermaßen aussehen:

```
module M1(event bool &o) {
    if(o) o = true;
}
```

Es gibt zwei Belegungen, die das Programm *erfüllen*. Das Ereignis o kann **true**, oder **false** sein. Ein nicht-*reaktives* Programm wäre beispielsweise das folgende:

```
module M2(event bool &o) {
    if(o) nothing; else o = true;
}
```

In diesem Fall gibt es keine Belegung, die das Programm *erfüllen* kann, die beiden Möglichkeiten (o = **true** oder o = **false**) führen jeweils zu einem Widerspruch.

2.6 Zyklische und azyklische Abhängigkeiten

Bei synchronen Sprachen kann man nicht immer auf den ersten Blick entscheiden, welche Mikroschritte zusammen ausgeführt werden sollen, da das Ergebnis eines Befehls die Ausführung eines anderen beeinflussen kann. So kann zum Beispiel x = **true**; ein Ereignis setzen, das im selben Schritt Grundlage einer **if**-Abfrage ist und somit deren Verhalten steuert. Die Information kann also rückwärts fließen.

An den beiden obigen Beispielen ist zu erkennen, dass die Auswertung der **if**-Abfrage von deren Verhalten (welcher Zweig ausgeführt wird) abhängt. Hier handelt es sich um zyklische Programme. Allerdings müssen zyklische Programme nicht zwingenderweise logisch inkorrekt sein. Als Beispiel sei hier folgendes Programm aufgeführt:

```
module M3(event bool &o) {
    if(o) o = true;
    else o = true;
}
```

Bei diesem Programm gibt es nur eine Belegung, die das Programm erfüllt (o = **true**). Durch die **if**-Abfrage wird auf jeden Fall die Zuweisung an o ausgeführt.

Zyklische Programme werden in *selbsterfüllende* und *konstruktive* Programme eingeteilt, wobei die *konstruktiven* durch die Kausalitätsanalyse definiert werden.

Kapitel 3

Kausalität und Simulation

Bis jetzt haben wir die synchrone Sprache Quartz mit ihrer Semantik kennengelernt. Wir haben aber noch keine Möglichkeit gefunden, die Ausgaben von unserem Programm zu berechnen. Dieses Defizit wollen wir in diesem Kapitel teilweise nachholen. Wir werden hier auf die Kausalitätsanalyse eingehen und zeigen, wie wir diese für die Simulation von synchronen Programmen verwenden können. Wir können mit der hier vorgestellten Methode allerdings nicht alle *logisch korrekten* Programme simulieren sondern nur einen Teil, den wir als *kausale* Programme definieren. Es sei aber angemerkt, dass es auch Methoden gibt alle *logisch korrekten* Programme zu simulieren, allerdings versucht man mit entprechend eingeschränkten Definitionen der Kausalitätsanalyse langen Laufzeiten aus dem Weg zu gehen. Weitere Literatur zu diesem Thema ist unter [2], [7] und [9] zu finden.

3.1 Kausalitätsanalyse

Wir werden zuerst einige Begriffe einführen und erklären, die wir für die Kausalitätsanalyse brauchen. Darauf aufbauend werden wir einen Algorithmus angeben, der versucht die Ausgaben eines Programms für einen Makroschritt zu bestimmen. Wie schon erwähnt, können wir mit unserer verwendeten Analyse nicht alle Programme betrachten, unser Algorithmus kann also nicht notwendigerweise zu jedem logisch korrekten Programm alle Ausgaben bestimmen. Ein zweiter Algorithmus stellt uns die Vollständigkeit und Widerspruchsfreiheit der berechneten Ausgaben sicher.

Dreiwertige Logik

Für unsere Analyse betrachten wir eine dreiwertige Logik über der Menge $\{1, 0, \perp\}$, wobei wir die Operationen $\wedge, \vee$ und $\neg$ wie folgt definieren:

$\wedge$	$\bot$	0	1
$\bot$	$\bot$	0	$\bot$
0	0	0	0
1	$\bot$	0	1

$\vee$	$\bot$	0	1
$\bot$	$\bot$	$\bot$	1
0	$\bot$	0	1
1	1	1	1

x	$\neg x$
$\bot$	$\bot$
0	1
1	0

Mit Hilfe dieser dreiwertigen Logik werden wir im Folgenden versuchen abzuleiten, welche Aktionen im Makroschritt definitiv ausgeführt werden, bzw. welche Aktionen auf keinen Fall ausgeführt werden. Dazu nehmen wir für alle Variablen, die keine Eingansvariablen sind, also von denen wir den Wert noch nicht kennen, $\mathcal{E}(x) = \bot$ an und versuchen die Umgebung iterativ herzuleiten. Da aber in der Sprache Quartz noch andere Datentypen als **bool** existieren werden wir für diese auch das Symbol $\bot$ benutzen, um darzustellen, dass wir den Wert einer Variablen noch nicht kennen. Das Ergebnis arithmetischer Oprationen auf dem Wert $\bot$ definieren wir als $\bot$. Für die Bestimmung der auszuführenden und nicht auszuführenden Aktionen reicht der Datentyp **bool**, da alle Steueranweisungen über diesem definiert sind.

Monotonie

Eine partielle Ordnung für alle Umgebungen sei folgendermaßen definiert:

$$\mathcal{E}_1 \preceq \mathcal{E}_2 :\Leftrightarrow \forall x[\mathcal{E}_1(x) = \bot \ \vee \mathcal{E}_1(x) = \mathcal{E}_2(x)]$$

MustAct, CanAct

Wir definieren die Mengen MustAct$(\mathcal{E}, S)$ und CanAct$(\mathcal{E}, S)$ rekursiv über eine gegebenen Anweisung S und Umgebung $\mathcal{E}$. Die Menge MustAct beeinhaltet alle Anweisungen, von denen wir anhand der Umgebung $\mathcal{E}$ ableiten können, dass sie im aktuellen Makroschritt auf jeden Fall ausgeführt werden. In der Menge CanAct sind alle Anweisungen, für die es nicht ausgeschlossen ist, dass sie in Bezug auf $\mathcal{E}$ noch ausgeführt werden.

Für $\text{Act} \in \{\text{MustAct}, \text{CanAct}\}$ *gilt* :

$$\begin{array}{lcl}
\text{Act}(\mathcal{E}, \textbf{nothing}) & = & \{\} \\
\text{Act}(\mathcal{E}, l : \textbf{ pause}) & = & \{\} \\
\text{Act}(\mathcal{E}, \alpha) & = & \{\alpha\}, \\
 & & \alpha \in \{x = \tau, \textbf{next}(x) = \tau\} \\
\text{Act}(\mathcal{E}, S1 \mid\mid S2) & = & \text{Act}(\mathcal{E}, S1) \cup \text{Act}(\mathcal{E}, S2) \\
\text{Act}(\mathcal{E}, \textbf{do } S \textbf{ while } \sigma) & = & \text{Act}(\mathcal{E}, S) \\
\text{Act}(\mathcal{E}, [\textbf{weak}] \textbf{ abort } S \textbf{ when } \sigma) & = & \text{Act}(\mathcal{E}, S) \\
\text{Act}(\mathcal{E}, \textbf{weak abort } S \textbf{ when immediate } \sigma) & = & \text{Act}(\mathcal{E}, S)
\end{array}$$

$$\text{CanAct}(\mathcal{E}, S1;\ S2) = \begin{cases} \text{CanAct}(\mathcal{E}, S1) & \mathcal{E}(Inst(S1)) = false \\ \text{CanAct}(\mathcal{E}, S1) \cup \text{CanAct}(\mathcal{E}, S2) & sonst \end{cases}$$

$$\text{CanAct}(\mathcal{E}, \textbf{if}(\sigma)\ S1\ \textbf{else}\ S2) = \begin{cases} \text{CanAct}(\mathcal{E}, S1) & \mathcal{E}(\sigma) = true \\ \text{CanAct}(\mathcal{E}, S2) & \mathcal{E}(\sigma) = false \\ \text{CanAct}(\mathcal{E}, S1) \cup \text{CanAct}(\mathcal{E}, S2) & sonst \end{cases}$$

$$\text{CanAct}(\mathcal{E}, \textbf{while}(\sigma)\ \textbf{do}\ S\ \textbf{end}) = \begin{cases} \{\} & \mathcal{E}(\sigma) = false \\ \text{CanAct}(\mathcal{E}, S) & sonst \end{cases}$$

$$\text{CanAct}(\mathcal{E}, \textbf{abort}\ S\ \textbf{when immediate}\ \sigma) = \begin{cases} \{\} & \mathcal{E}(\sigma) = true \\ \text{CanAct}(\mathcal{E}, S) & sonst \end{cases}$$

$$\text{MustAct}(\mathcal{E}, S1;\ S2) = \begin{cases} \text{MustAct}(\mathcal{E}, S1) \cup \text{MustAct}(\mathcal{E}, S2) & \mathcal{E}(Inst(S1)) = true \\ \text{MustAct}(\mathcal{E}, S1) & sonst \end{cases}$$

$$\text{MustAct}(\mathcal{E}, \textbf{if}(\sigma)\ S1\ \textbf{else}\ S2) = \begin{cases} \text{MustAct}(\mathcal{E}, S1) & \mathcal{E}(\sigma) = true \\ \text{MustAct}(\mathcal{E}, S2) & \mathcal{E}(\sigma) = false \\ \{\} & sonst \end{cases}$$

$$\text{MustAct}(\mathcal{E}, \textbf{while}(\sigma)\ \textbf{do}\ S\ \textbf{end}) = \begin{cases} \text{MustAct}(\mathcal{E}, S) & \mathcal{E}(\sigma) = true \\ \{\} & sonst \end{cases}$$

$$\text{MustAct}(\mathcal{E}, \textbf{abort}\ S\ \textbf{when immediate}\ \sigma) = \begin{cases} \text{MustAct}(\mathcal{E}, S) & \mathcal{E}(\sigma) = false \\ \{\} & sonst \end{cases}$$

Trivialerweise gilt, dass $\mathrm{MustAct}(\mathcal{E}, S) \subseteq \mathrm{CanAct}(\mathcal{E}, S)$. Beide Mengen sind endlich, da sie durch die Anzahl der Anweisungen in S beschränkt sind.

Fixpunktiteration

Bei der Fixpunktiteration werden wir versuchen, die Belegungen für alle Variablen zu bestimmen. Grundlagen zur Fixpunktiteration werden in [10] behandelt. Dabei benötigen wir die Umgebung $\mathcal{E}_{old}$ des letzten Schritts, um auf Werte von Zustandsvariablen zurückzugreifen. Desweitern wird die Menge $\mathcal{D}_{old}$ benötigt um **next**-Zuweisungen des letzten Schritts zu beachten. Die Umgebung $\mathcal{E}$ sollte den Eingangsvariablen definierte Werte und allen anderen Variablen $\perp$ zuweisen. Der Algorithmus sieht folgendermaßen aus:

function computeOutputs($\mathcal{E},\ \mathcal{E}_{old},\ \mathcal{D}_{old},\ S,\ \mathcal{V}_{out}$)
  do
    $\mathcal{E}' := \mathcal{E}$
    $D_{can} := \mathrm{CanAct}(\mathcal{E},\ S)$
    $D_{must} := \mathrm{MustAct}(\mathcal{E},\ S)$
    for all $x \in \mathcal{V}_{out} \wedge \mathcal{E}(x) = \perp$ **do**
      if next$(x) = \tau \in D_{old}$ **then set** $\mathcal{E}(x) := \tau$
      elsif $x = \tau \in D_{must}$ **then set** $\mathcal{E}(x) := \tau$
      elsif $\nexists \tau [x = \tau \in D_{can}]$ **then**
        if isEvent(x) **then set** $\mathcal{E}(x) := \mathrm{defaultValue}(x)$
        else set $\mathcal{E}(x) := \mathcal{E}_{old}(x)$
        end
    end
  while $\mathcal{E}' \neq \mathcal{E}$
  return $(D_{can},\ \mathcal{E})$
end

Dabei ist leicht zu erkennen, dass in jedem Interationsschritt der Schleife stehts $\mathcal{E}' \preceq \mathcal{E}$ gilt. Denn es bekommen höchstens Variablen, die den Wert $\perp$ haben einen definierten Wert zugewiesen. Dies beinhaltet aber folgenden Sachverhalt:

- $\mathrm{CanAct}(\mathcal{E}', S) \supseteq \mathrm{CanAct}(\mathcal{E}, S)$
- $\mathrm{MustAct}(\mathcal{E}', S) \subseteq \mathrm{MustAct}(\mathcal{E}, S)$

Die Menge $\mathrm{MustAct}(\mathcal{E}, S)$ ist nach oben durch die Menge an Anweisungen in S beschränkt. Die Menge $\mathrm{CanAct}(\mathcal{E}, S)$ ist trivialerweise nach unten beschränkt

und kann initial nur endlich viele Anweisungen enthalten. Dadurch ist sichergestellt, dass irgendwann der Fixpunkt erreicht wird, also $\mathcal{E}' = \mathcal{E}$ gilt und der Algorithmus terminiert.

Wie einleitend erwähnt, sind nicht alle logisch korrekten Programme kausal. Das bedeutet, das trotz Terminierung des Algorithmus die Berechnung nicht erfolgreich sein muss. Dies ist genau dann der Fall, wenn Variablen existieren, die immer noch den Wert $\perp$ haben oder widersprüchliche Zuweisungen an Variablen vorgenommen werden. Diese Konsitenz überprüfen wir mit folgendem Algorithmus:

function isConsistent($\mathcal{D}_{old},\ \mathcal{E}_{old},\ \mathcal{D},\ \mathcal{E},\ x$)
  $\mathcal{D}_{asign}\ :=\ \{\mathcal{E}_{old}(\tau) \mid \mathbf{next}(x) := \tau\ \in \mathcal{D}_{old}\}\ \cup\ \{\mathcal{E}(\tau) \mid x := \tau\ \in \mathcal{D}\}$
  if $\mathcal{D}_{asign} = \{\pi_1, \pi_2, \ldots \pi_n\}\ \neq\ \emptyset$ **then**
    return $(\pi_1 = \pi_2 = \ldots = \pi_n)$
  elsif isEvent(x) **then**
    return $(\mathcal{E}(x) = \text{defaultValue}(x))$
  else
    return $(\mathcal{E}(x) = \mathcal{E}_{old}(x))$
  end
end

Wir benötigen die Mengen $\mathcal{D}_{old}, \mathcal{E}_{old}$, um die Konsistenz zu dem vorherigen Makroschritt sicherzustellen, weil auf alte Werte von Variablen zurückgegriffen werden muss und Anweisungen des letzten Schritts (z.B. $\mathbf{next}(x) := \tau;$) den aktuellen Wert beeinflussen können.

Die Konsistenz eines Makroschritts folgt aus der Konsitenz jeder Variablen der Umgebung. Wir bestimmen also zuerst alle Zuweisungen, die für die Variable x im aktuellen Schritt gültig sind. Wenn Zuweisungen exisitieren, müssen diese alle in dem gleichen Wert für die Variable resultieren. Falls keine Zuweisungen an die Variable gemacht werden, ist zu unterscheiden, ob x eine Zustands- oder Ereignisvariable ist. In erstem Fall muss der aktuelle Wert $\mathcal{E}(x)$ dem des letzten Schritts entsprechen. Ansonsten muss $\mathcal{E}(x)$ den Standardwert für die Ereignisvariable annehmen.

3.2 Simulator

Der vorgestellte Algorithmus computeOutputs($\mathcal{E}_{in},\ \mathcal{E}_{old},\ \mathcal{D}_{old},\ S,\ \mathcal{V}_{out}$) dient unserem Simulator als Grundlage um in jedem Schritt, ausgehend von den Eingangsvariablen, die Ausgaben zu berechnen.

function *simulate*$(S,\ \mathcal{V}_{out})$
  $\forall x\ [\ \mathcal{E}_{old}(x)\ :=$ defaultValue$(x)]$
  $\mathcal{D}_{old}\ := \{\}$
  do
    $\mathcal{E}_{in}\ :=\ ReadInputs\ ()$
    $(D',\ \mathcal{E})\ :=$ computeOutputs $(\mathcal{E}_{in},\ \mathcal{E}_{old},\ \mathcal{D}_{old},\ S,\ \mathcal{V}_{out})$
    $(D,\ b,\ S')\ :=\ ComputeTR\ (S,\ \mathcal{E})$
    if $\exists x \in \mathcal{V}_{out}$ [$\neg$isConsistent $(\mathcal{D}_{old},\ \mathcal{E}_{old},\ \mathcal{D},\ \mathcal{E},\ x)$] **then fail end**
    $\mathcal{E}_{old}\ :=\ \mathcal{E}$
    $\mathcal{D}_{old}\ :=\ \mathcal{D}$
    $\mathcal{S}\ :=\ \mathcal{S}'$
  while $\neg b$
end

Um den ersten Schritt nicht separat betrachten zu müssen definieren wir am Anfang $\mathcal{E}_{old}$ und $\mathcal{D}_{old}$ entsprechend. Die Funktion *ReadInputs* () liefert uns die Belegung der Eingabevariablen und setzt alle anderen Variablen auf den Wert $\perp$. Auf dieser Grundlage versucht computeOutputs$(\mathcal{E}_{in},\ \mathcal{E}_{old},\ \mathcal{D}_{old},\ S,\ \mathcal{V}_{out})$ die restlichen Variablen zu bestimmen. Nun haben wir eine Umgebung, die für die Ausführung des Schritts in Frage kommt. Mithilfe dieser Umgebung kann $ComputeTR\ (S,\ \mathcal{E})$ über die Transitionsrelationen bestimmen, welche Aktionen $\mathcal{D}$ im aktuellen Schritt ausgeführt werden und welche Anweisungen S' im nächsten Schritt zu betrachten sind. Anschließend ist die Konsistenz zwischen den ausgeführten Aktionen und der Umgebung zu überprüfen. Dazu wird isConsistent $(\mathcal{D}_{old},\ \mathcal{E}_{old},\ \mathcal{D},\ \mathcal{E},\ x)$ für jede Variable x ausgeführt. Wenn kein Fehler auftritt haben wir die Ausgaben für den aktuellen Schritt gefunden und wir fahren mit dem nächsten Schritt fort. Hierbei ist an dem Wert b zu erkennen, ob es einen nächsten Schritt gibt, oder das gesamte Programm terminiert.

Falls wir nicht alle Variablen belegen konnten ist das Programm, da die Kausalitätsanalyse fehlgeschlagen ist, *nicht konstruktiv*. Das bedeutet aber noch nicht, dass es nicht *logisch korrekt* ist.

3.3 Ausblick

Wir haben versucht eine Kausalitätsanalyse zu definieren, mit deren Hilfe wir synchrone Programme simulieren können. Allerdings haben wir auch festgestellt, dass nicht alle Programme sich durch unseren Simulator berechnen lassen. Hierzu sei nochmals das folgende Beispiel aufgeführt:

```
module M3(event bool &o) {
    if(o) o = true;
    else o = true;
}
```

Wie oben schon erwähnt, wäre $o = \mathbf{true}$ die einzige erfüllende Belegung für das Programm, dass nur einen Schritt benötigt. Bei unserer Analyse wird aber zuerst $\mathcal{E}(o) =\perp$ gesetzt, da o eine Ausgabevariable ist. die entsprechenden Mengen CanAct und MustAct sehen folgendermaßen aus:

- $\text{CanAct}(\mathcal{E}, \alpha) = \{o = \mathbf{true}\}$
- $\text{MustAct}(\mathcal{E}, \alpha) = \{\}$

In unserem Algorithmus können wir anhand unserer Tests das Ausführen von $o = \mathbf{true}$; also nicht wider- und auch nicht belegen. Das bedeutet, dass der Algorithmus mit einem undefinierten Wert für o terminiert.

Dieses Beispiel könnte man z.B. mit einer anderen Definition von CanAct und MustAct bearbeiten, indem man für die **if**-Abfrage auch Gemeinsamkeiten der beiden Zweige erfasst. Eine Definition dafür könnte beispielsweise folgendermaßen aussehen:

$$\text{MustAct}(\mathcal{E}, \mathbf{if}(\sigma)\ S1\ \mathbf{else}\ S2)$$
$$= \begin{cases} \text{MustAct}(\mathcal{E}, S1) & \mathcal{E}(\sigma) = true \\ \text{MustAct}(\mathcal{E}, S2) & \mathcal{E}(\sigma) = false \\ \text{MustAct}(\mathcal{E}, S1) \cap \text{MustAct}(\mathcal{E}, S2) & sonst \end{cases}$$

Kapitel 4

Value Change Dump

Das “Value Change Dump”-Format ist im Verilog-Standard [4] spezifiziert. Es bietet die Möglichkeit die Belegegungen für Variablen in jedem Schritt zu erfassen. Dabei reicht es auch, den Wert einer Variablen nur dann anzugeben, wenn sich dieser verändert hat. Ein Beispiel dieses Format, dass die Bestandteile zeigt, die wir zum Abspeichern unserer Simulationsergebnisse benötigen ist in Abb. 4.1 zu sehen.

Im Kopf der Datei werden die Version des Simulators und die Skalierung der Zeit angegeben. Da wir bei Quartz von *logischer* und nicht von physikalischer Zeit sprechen, ist diese Angabe nicht wirklich relevant. Sie wird aber in Programmen zur Anzeige von VCD-Dateien zur Skalierung verwendet. Als nächstes werden Variablen deklariert. Dabei werden ihnen Typen und Bezeichner zugewiesen. Die Bezeichner dienen dazu, die Variable später in der Auflistung der Werte zu referenzieren. In unserer Datei wird für eine Variable mit dem Namem “*name*” der Bezeichner “(*name*” verwendet. Die Variablen werden in Sichtbarkeitsbereiche unterteilt. Wir verwenden hier die Bereiche “input” “output” und “local” um die entsprechenden Variablen zu unterscheiden.

Die Definition eines Schrittes beginnt mit der Nummer i des Schrittes in der Form “$\#i$”. Danach werden alle Variablen mit Werten aufgelistet, die sich in diesem Schritt verändern. Der Wert beginnt mit einem b um die binäre Form zu verdeutlichen und wird dementsprechend als Folge aus 0 und 1 angegeben. Dabei können führende Nullen vernachlässigt werden. Zwischen dem b und dem Wert darf kein Leerzeichen stehen. Der Variablenbezeichner folgt schließlich durch eine Trennung von mindestens einem Leerzeichen. Die Schritte und Variablen müssen nicht vollständig angegeben sein. Fehlt eine Variable, so wird angenommen, dass sie sich in diesem Schritt nicht verändert hat. Fehlt der komplette Schritt, so hat sich keine der Variablen verändert.

Im Verilog-Standard sind noch mehr Variablentypen definiert, als die, die wir verwenden. Allerdings spielen diese für die Sprache Quartz keine Rolle und werden hier nicht erwähnt. Wir benutzen das VCD-Format um eine Export-Funktion bereitzustellen, die es ermöglicht die Simulationsergenisse auch mit anderen Programmen anzuzeigen. Zum Anzeigen von Dateien im VCD-Format

können die Programme GtkWave[1] und Dinotrace [2] verwendet werden.

```
$version
    VERILOG-XL 2.5
$end

$timescale
    1s
$end

$scope module input $end
$var integer 8 (a a $end
$upscope $end

$scope module output $end
$var reg 1 (o o $end
$upscope $end

$scope module local $end
$var reg     5 (b b $end
$var integer 8 (count count $end
$upscope $end

$enddefinitions $end

#0
b10100110 (a
b1 (o
b11110 (b
b00000001 (count

#1
b10111010 (a
b10100 (b
b00000010 (count

#2
b11101000 (a
b0 (o
b10101 (b
b00000011 (count
```

Abbildung 4.1: Beispiel einer VCD-Datei

[1] http://home.nc.rr.com/gtkwave/
[2] http://www.veripool.com/dinotrace/

Kapitel 5

Averest

Averest wurde von der Arbeitsgruppe “Reaktive Systeme” an der Technischen Universität Kaiserslautern entwickelt. Die Homepage mit Dokumentation und Downloadbereich ist unter [1] erreichbar.

Averest baut auf der oben beschriebenen Sprache Quartz auf. Es umfasst verschiedene, weitgehend eigenständige Tools, die fast die komplette Entwicklung reaktiver Systeme abdecken. Enthalten sind Compiler, Modellprüfer und ein Werkzeug für die Hardware-/Softwaresynthese. Darüber hinaus existiert ein Plugin zur Integration in die Entwicklungsumgebung Eclipse [3]. Die einzelnen Bestandteile werden im folgenden kurz und in [8] genauer beschrieben.

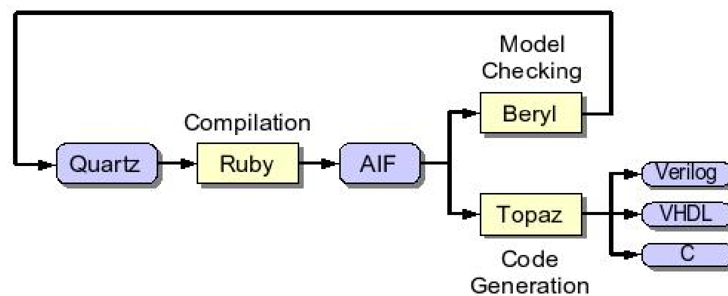

Abbildung 5.1: Aufbau [1]

5.1 Übersetzung - Der Compiler Ruby

Quartz-Dateien werden mit Hilfe des Compiler Ruby in das Averest Interchange Format (AIF) [5] übersetzt. Dieses bildet die Grundlage sowohl zur Verifikation als auch zur Codeerzeugung. Die Korrektheit der Übersetzung ist weitgehend vom interaktiven Theorembeweiser HOL sichergestellt. Der Compiler verfügt darüber hinaus über die Möglichkeit, ein in Quartz geschriebenes Programm zu

simulieren und die Werte der Variablen und Events in jedem Schritt auszugeben. Dieser Simulator ist die Grundlage für unsere Oberfläche.

5.2 Verifikation - Der Modelprüfer Beryl

Beryl ist ein symbolischer Modelprüfer, der auf dem Averest Interchange Format aufbaut. Er prüft, ob ein Programm eine gegebene Spezifikation erfüllt, beides wird im AIF übergeben. Er kann dabei lokale, globale und beschränkte Modellprüfungsagorithmen verwenden. Er arbeitet dabei auf endlichen und unendlichen Zustandssystemen.

5.3 Codeerzeugung - Das Synthese-Werkzeug Topaz

Topaz dient zur Codeerzeugung aus dem AIF. Dabei können als Ausgabeformate die Hardwarebeschreibungssprachen VHDL und Verilog sowie Quellcode der Programmiersprache C verwendet werden. Des weiteren ist es möglich C-Code zu erzeugen, der auf BrickOS aufbaut. Aus VHDL und Verilog können dann mittels weiterer Programme Hardware erzeugt werden.

5.4 Averest Eclipse-Plugin

Das Averest Eclipse Plugin bietet eine komfortable grafische Oberfläche, um Quartz-Dateien und -Projekte zu erstellen, editieren und mittels der Averest-Tools weiterzuverarbeiten. Dabei kann auf die Projektverwaltung von Eclipse zurückgegriffen werden. Mittels des Plugins können die Parameter von Compiler, Verifikations- und Synthese-Tool variiert und eingestellt werden und die Tools direkt aus der Oberfläche heraus aufgerufen werden. Dabei können die Averest-Tools auf dem lokalen Rechner installiert sein, oder es kann über eine Netzwerkverbindung der Averest-Server benutzt werden. Zweites macht es z.B. möglich, Averest ohne großen Aufwand unter Windows zu benutzen.

Die Averest-Werkzeuge können über das folgendes Menü aufgerufen werden:

Mit dem linken Knopf können die Einstellungen der einzelnen Aktionen geändert werden und alle Aktionen hintereinander ausgeführt werden. Daneben hat man für die Aktionen Übersetzung, Verifikation und Codeerzeugung jeweils einen Knopf. Mit dem rechten Knopf startet man ein abschließendes Skript oder ein externes Programm. Zum Beispiel könnte man mit dem abschließenden Skript die direkte Übertragung auf einen Mikrokonroller bewerkstelligen.

Kapitel 6

Die Simulator-Oberfläche

Wir haben das Averest-Eclipse-Plugin um die Simulations-Oberfläche erweitert, auf deren Benutzung wir als erstes eingehen werden. Anschließend werden wir die internen Zusammenhänge zwischen Simulator und Oberfläche, sowie den Hauptteil der Implementierung erklären. Die Bezeichnung "Simulator" wird hier für den im Übersezter Ruby implementierten Simulator verwendet. Als "Simulationsoberfläche" oder auch "Oberfläche" bezeichnen wir den in das Eclipse-Plugin integrierten Teil zur Simulation.

6.1 Benutzerschnittstelle

6.1.1 Funktionen

Mithilfe der Simulatoroberfläche ist es möglich, bequem über das Averest-Eclipse-Plugin Programme der synchronen Sprache Quartz zu simulieren. In den Kapiteln über die Sprache Quartz haben wir gesehen, was ein Schritt in einer synchronen Sprache bedeutet. Ein Schritt hat hier eine genau festgelegte Definition und kann als Übergang von einem Zustand des Systems in einen anderen aufgefasst werden. Das heißt aber auch, dass in einem Schritt alle Variablen gleichzeitig verändert werden können. Deshalb bietet sich die schrittweise Darstellung der Werte aller Variablen in einer Tabelle an.

Des weiteren ist es möglich die Eingabevariablen eines Moduls für jeden Schritt zu definieren und das Verhalten des Systems für diese Eingabefolge zu beobachten. Dabei wird das Programm immer bis zu einer Obergrenze an Schritten simuliert. Natürlich ist es möglich, auf der bereits simulierten Folge aufzubauen oder Eingabewerte zwischendrin zu ändern und an dieser Stelle neu aufzusetzen.

Das Plugin kann mehrere Instanzen des Simulatorfensters verwalten. Somit können mehrere Programme gleichzeitig simuliert und analysiert werden. Nach Beenden und Neustart der Eclipse-Umgebung wird versucht, den alten Zustand wiederherzustellen. Das setzt natürlich vorraus, dass die Dateien, in denen die

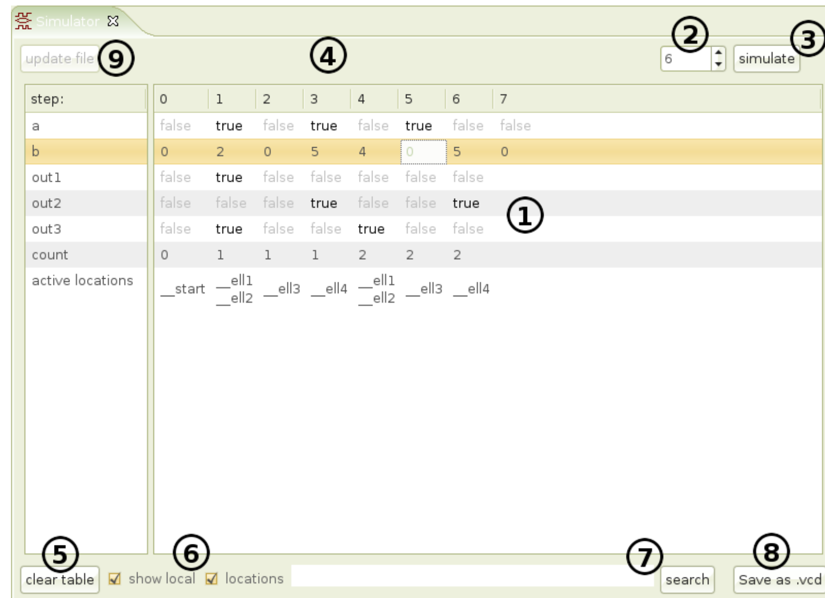

Abbildung 6.1: Bildschirmfoto der Simulationsoberfläche

Werte abgespeichert werden und die auch zur Kommunikation zwischen Oberfläche und Simulator dienen, noch alle vorhanden sind.

6.1.2 Die Simulationsansicht

Das Averest-Plugin wurde um eine Aktion zum Simulieren einer Quartz-Datei erweitert.

Um die Simulationsansicht für eine Quartz-Datei zu öffnen, muss diese Datei im Editor angezeigt und der Simulationsknopf in der Aktionsleiste des Averest-Plugins betätigt werden. Daraufhin wird der initiale Simulationsprozess zum Bestimmen der Variablenbezeichnungen und Typen gestartet. Für jede zu simulierende Quartz-Datei entsteht ein neues Ansichtsfenster (Abb. 6.1), welches sich aus folgenden Bestandteilen zusammensetzt:

1) Tabelle

Die Tabelle stellt die Simulationsdaten grafisch dar. Sie zeigt zeilenweise die Variablen und spaltenweise ihre Belegung für den jeweiligen Schritt, der im Spaltenkopf definiert ist, an. In der Tabelle kann man mithilfe der Pfeiltasten oder der Maus navigieren. Die Werte der Eingabevariablen können verändert werden. Durch das Verändern werden alle folgenden und bereits simulierten Werte ungültig. Wir färben diese lila ein, um dies zu verdeutlichen. Zusätzlich wird bei einer Änderung überprüft, ob der eingegebene Wert zu dem Datentyp passt. Falls nicht wird versucht, den Wert entsprechend anzupassen, ansonsten wird der Standartwert des Datentyps benutzt. Fehler während der Simulation werden rot hervorgehoben. Es besteht die Möglichkeit, bei Booleschen Variablen die Zeichenketten für "true" und "false" und deren Farbe zu ändern. Dies wird bei den Einstellungen genauer beschrieben.

2) Letzter zu simulierende Schritt

Hier wird angegeben, wie viele Schritte simuliert werden sollen. Mit Erhöhen des Wertes wird auch die Tabelle entsprechend erweitert, wobei Variablen mit dem letzten vorhandenen Wert fortgesetzt und Ereignisse auf ihren Standardwert gesetzt werden. Die Werte der lokalen Variablen und Ausgabevariablen werden nicht angezeigt, da diese Werte noch nicht durch eine Simulation ermittelt wurden und somit nicht das Verhalten des Programms widerspiegeln würden. Beim Verringern der Schrittzahl zeigt die Tabelle weiterhin alle Werte der Eingabevariablen an, damit diese erhalten bleiben. Allerdings wird bei einem Simulationsvorgang nur bis zu dem angegebenen Schritt simuliert und alle nichtsimulierten Werte gelöscht.

3) Simulations-Knopf

Dieser Knopf startet die Simulation beginnend mit dem kleinsten Schritt, der geändert wurde bis zu dem Schritt der mit 2) angegeben ist. Falls diese Schritte aber schon simuliert wurden und keine Änderung vorgenommen wurden, wird nachgefragt, ob die Simulation dennoch gestartet werden soll. Außerdem wird darauf hingewiesen, falls die zu simulierende Datei nicht gespeichert wurde.

4) Systemmeldungen

Hier werden Systemmeldungen angezeigt.

5) Löschen-Knopf

Nach Bestätigung des Löschvorgangs, werden alle Werte der Tabelle gelöscht und diese neu initialisiert.

6) Lokale Variablen und "Locations" anzeigen

Da wir eine Unterscheidung zwischen Ausgabevariablen und lokalen Variablen haben, ist es möglich, die lokalen Variablen auszublenden, um die Tablelle übersichtlicher zu gestalten. Das gleiche gilt für die in jedem Schritt aktiven Kontrollflusspunkte.

7) Suchfeld

Man kann in der Tabelle zur einfachen Navigation nach Schritten, Variablenbezeichnungen oder beidem suchen. Bei der Suche nach einer Variablen wird die gesuchte Variable im aktuellen Schritt markiert. Die Suche nach einem Schritt funktioniert entsprechend. Die Angaben zur Suche nach Variable und Schritt werden durch Komma oder Leerzeichen getrennt.

Bei der Variablensuche können auch Teilwörter verwendet werden, wobei immer ausgehend von der aktuellen Position gesucht wird. Dies macht es zum Beispiel möglich, alle Variablen mit einem gemeinsamen Teilwort zu durchlaufen.

8) Als VCD-Datei exportieren

Hiermit exportiert man die Daten der Tabelle in das VCD-Format um die Daten mit anderen Programmen visualisieren zu können.

9) Aktualisieren Knopf

Sobald die Simulationsansicht geöffnet ist, registriert diese Änderungen an der Quartz-Datei und daraus referenzierten Dateien. Nach Änderungen in diesen Dateien, wird das Ansichtsfenster gesperrt, da es zu Inkonsistenzen zwischen den angezeigten und den simulierten Daten kommen kann. Die alten Werte werden beibehalten, bis das Ansichtsfenster vom Benutzer aktualisiert wird.

6.1.3 Das Einstellungsfenster

Das Einstellungsfenster ermöglicht es, die Tabelle übersichtlicher zu gestalten, indem die Werte "true" und "false" und deren Farbe verändert werden kann. Die Änderungen werden dauerhaft gespeichert und in alle offenen Simulationsfenster übernommen.

1) Anzeigen von Wahr und Falsch

Man hat drei Einsellungsmöglichkeiten zur Anzeige der Booleschen Werte:

- Für Wahr "true" und für False "false" anzeigen
- Für Wahr "1" und für False "0" anzeigen

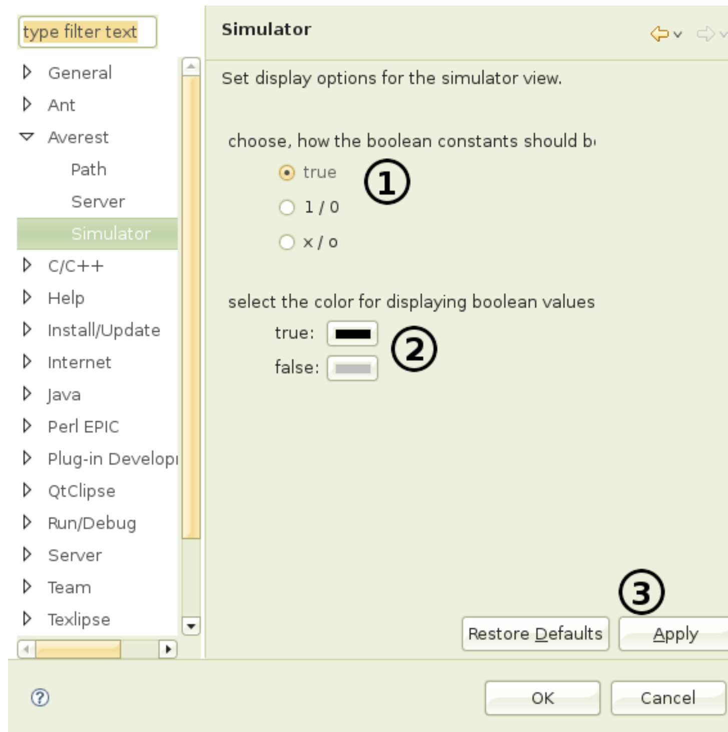

Abbildung 6.2: Bildschirmfoto des Einstellungsfenster

- Für Wahr "x" und für False "o" anzeigen

2) Farbe für die Werte Wahr und Falsch

Zusätzlich kann man festlegen, in welcher Farbe die Werte dargestellt werden, um die Tabelle übersichtlicher zu gestalten.

3) Einstellungen übernehmen-Knopf

Alle Änderungen werden sofort in die Tabelle übernommen und angezeigt.

6.2 Funktionaler Aufbau

Die Simulationsoberfläche muss eine große Menge an Werten für die Variablen verwalten, mit dem Simulator kommunizieren, eine Benutzerschnittstelle bieten und sich in die Eclipse-Umgebung einfügen. Im Folgenden werden diese Zusammenhänge beschrieben. Dieser Abschnitt dient als Gesamtüberblick und als erste Dokumentation. Für weitere Details ist der Quellcode heranzuziehen.

6.2.1 Kommunikation mit dem Simulator

Einer der wesentlichen Bestandteile bildet die Kommunikation mit dem Simulator. Diese wird durch eine XML-Datei geregelt, die die relevanten Informationen erfasst. Die XML-Datei hat den gleichen Namen (Der Simulator als Kommandozeilenwerkzeug akzeptiert auch andere Dateien), wie die zu simulierende Datei, allerdings bekommt sie die Dateierweiterung ".sim" Das Beispiel (Abb. 6.3) zeigt die wesentlichen Bestandteile des Formats.

Die zur Simulation erforderliche Quartz-Datei, sowie die Simulationsdatei selbst werden in dem "file"-Bereich angegeben.

Der Simulator sowie auch die Oberfläche lesen die Informationen aus der Datei aus, verändern diese und schreiben sie zurück in die Datei. Im Prinzip hat der Simulator und auch die Oberfläche seine eigenen Bereiche in der Datei, die sie verändern. Die Oberfläche ermöglicht, stark vereinfacht dargestellt, das Editieren dieser Datei aus Benutzersicht. Sie liest aus dem Quartz-Programm keine Informationen aus. Die Oberfläche schreibt lediglich die Eingabewerte, die der Benutzer setzt, in die Datei. Der Simulator muss auf der anderen Seite das Quartz-Programm simulieren. Er kennt also die Informationen aus dem Quartz-Programm und schreibt auch Diese, soweit für die Oberfläche zur Anzeige wichtig, in die Datei. Desweitern simuliert er das Programm und bestimmt somit Ausgabewerte und lokale Variablen. Diese schreibt er ebenfalls in die Datei.

Im Bereich "declarations" werden die im Programm deklarierten Variablen aufgelistet. Anhand dieses Bereichs kann also der komplette Typ einer Variablen abgeleitet werden. Dieser Teil wird, wie schon erwähnt, vom Simulator erzeugt, da er die Variablendeklarationen kennt.

Im "past"-Bereich werden alle bereits simulierten Schritte abgespeichert. Dazu gehören die Belegungen aller Variablen, sowie die aktiven "locations" und getätigte `next`-Zuweisungen. Im "future"-Bereich dagegen werden die noch nicht simulierten Schritte abgelegt. Hier werden allerdings nur die Eingabevariablen gespeichert. Der Simulator liest die Datei aus, erstellt sich anhand der Vergangenheit einen Zustand, an dem die Simulation fortgesetzt werden soll und simuliert danach die unter "future" angegebenen Schritte. Das Simulationsergebnis, wird danach unter "past" eingetragen. Der Simulator verschiebt also Schritte von "future" nach "past". Die Oberfläche geht den umgekehrten Weg. Der Benutzer kann beliebige Eingabewerte ändern, dadurch werden entweder unter "future" neue Schritte angelegt, oder diese verändert (oder die Änderungen bezieht sich auf einen Schritt im "past"-Bereich). Dann wird der Schritt (und alle Nachfolgenden) in den "future"-Bereich verschoben, da nun das Simulationsergebnis nicht mehr stimmen muss. Bei dem Verschieben werden die angegebenen Werte von lokalen Variablen und von Ausgabevariablen, sowie `next`-Zuweisungen und "locations" gelöscht.

```
<?xml version="1.1" ?>
<sim>
  <files>
    <module>quartz_program.qrz</module>
    <trace>quartz_program.sim</trace>
  </files>
  <declarations>
    <decl name="a" flow="input" storage="memorized">bool</decl>
    <decl name="count" flow="local" storage="memorized">int</decl>
    <decl name="out1" flow="output" storage="event">bool</decl>
    <decl name="out2" flow="output" storage="event">bool</decl>
  </declarations>
  <past>
    <step count="0">
      <locations>__start</locations>
      <env name="a">false</env>
      <env name="out1">false</env>
      <env name="out2">false</env>
      <env name="count">0</env>
      <next name="count">1</next>
    </step>
    <step count="1">
      <locations>__ell1 __ell2</locations>
      <env name="a">true</env>
      <env name="out1">true</env>
      <env name="out2">false</env>
      <env name="count">1</env>
    </step>
  </past>
  <future>
    <step count="2">
      <env name="a">false</env>
    </step>
    <step count="3">
      <env name="a">true</env>
    </step>
    <step count="4">
      <env name="a">false</env>
    </step>
    <step count="5">
      <env name="a">true</env>
    </step>
  </future>
</sim>
```

Abbildung 6.3: Beispiel der Simulationsdatei

Die in der Datei angegebenen Schritte müssen immer vollständig sein. Das heißt für einen Schritt unter "past" alle Werte für alle Variablen sowie die `next`-Zuweisungen und aktiven "locations". Im "future"-Bereich dürfen nur die Eingabevariablen auftauchen. Für den Simulator müssen Sie nicht alle angegeben sein. Fehlt hier ein Wert, nimmt der Simulator einen Zufallswert an. Die Oberfläche setzt aber immer alle Werte.

6.2.2 Simulationsablauf

Der Ablauf eines Simulationsvorgangs wurde im vorhergehenden Abschnitt schon kurz angedeutet, er wird hier aber noch mal detaillierter beschrieben. Das Austauschen der Simulationsdatei erfolgt über die Mechanismen des Averest-Plugins. Die Datei kann auf dem lokalen Rechner simuliert werden, sie kann aber auch an den Averest-Server geschickt werden. Diese Möglichkeiten wollen wir hier nicht beachten, sondern nur davon ausgehen, dass dem Simulator beziehungsweise der Oberfläche die Simulationsdatei irgendwie verfügbar gemacht wird. Im Diagramm (Abb. 6.4) ist der Ablauf der Simulation schematisch dargestellt, wobei hier horizontale Pfeile das Austauschen der Simulationsdatei darstellen.

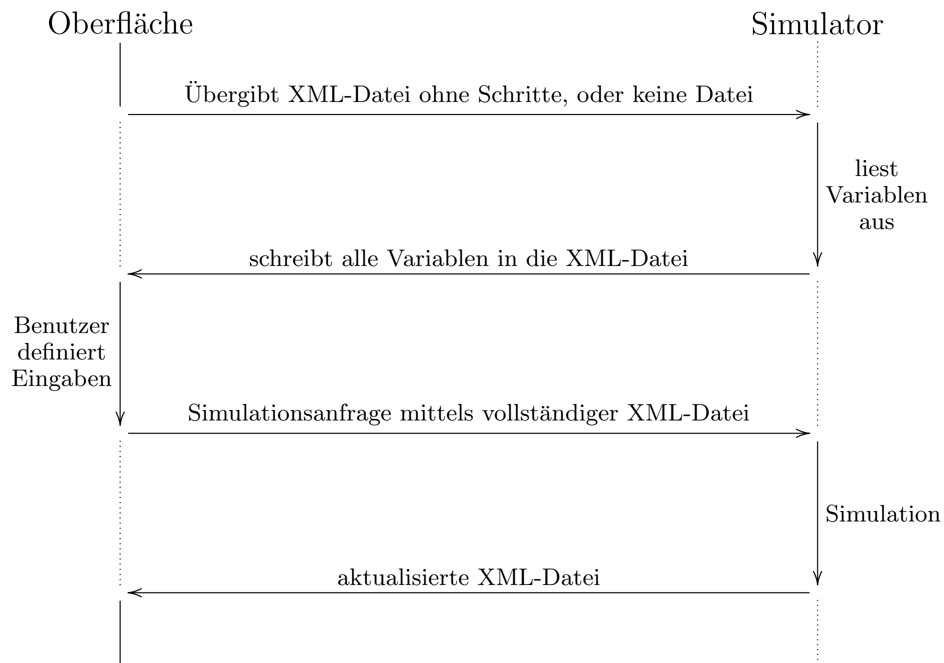

Abbildung 6.4: Simulationsablauf

Wie oben schon erwähnt, liest die Oberfläche die Variablen nicht selbst aus dem Quartz-Programm aus. Deshalb übergibt sie dem Simulator eine Datei, die keine Schritte enthält (oder gar keine Datei). Der Simulator führt dann keine Simulation durch, sondern aktualisiert nur die Variablendeklarationen (oder erstellt eine neue Datei). Jetzt hat die Oberfläche die Informationen über die Variablen und der Benutzer kann Werte für die Eingabevariablen in beliebigen Schritten vorgeben. Wenn der Benutzer alle Werte eingegeben hat, kann er

die Simulation starten. Die Werte werden in die Simulationsdatei geschrieben und dem Simulator übergeben. Dieser simuliert das Programm und aktualisert die Datei mit den Simulationsergebnissen. Diese können dann wieder von der Oberfläche angezeigt werden und der Benutzer kann wieder Änderungen an den Eingabevariabblen durchführen. Bei einer darauffolgenden Simulation werden nur noch die Schritte simuliert, die verändert wurden.

6.2.3 Die Werte der Variablen im Speicher

Die Variablenwerte werden durch die oben beschriebene XML-Datei zwischen Simulator und Oberfläche ausgetauscht. Die Werte müssen also von der Oberfläche aus der Datei gelesen und im Fenster angezeigt werden. In der Oberfläche soll der Nutzer zusätzlich die Möglichkeit haben, Werte von Variablen (zumindest von den Eingabevariablen) zu verändern. Also müssen die Werte im Speicher gehalten werden, um Änderungen möglich zu machen. Vor dem nächsten Simulationsvorgang müssen die Werte allerdings wieder zurück in die Datei geschrieben werden, damit der Simulator darauf zugreifen kann.

Unser erster Ansatz einer Datenstruktur, um die Werte in jedem Schritt zu repräsentieren, war eine Klasse, die für jede Variable eine Werteliste enthielt. Diese Werteliste war gekapselt, um für jeden Variablentyp eine Funktion bereitzustellen, die die Eingaben des Benutzers verifiziert. Dem Benutzer ist es prinzipiell möglich beliebige Zeichenketten als Werte in die Tabelle einzugeben, deshalb muss geprüft werden, ob der eingegebene Wert zu dem Typ der Variablen passt. Soweit war das ganze kein Problem. Zum erstellen der XML-Datei ist aber noch mehr notwendig, als die reinen Werte der Variablen. Um an einer beliebigen Stelle mit der Simulation aufsetzen zu können, werden die in jedem Schritt aktiven "locations" sowie die ausgeführten `next`-Zuweisungen benötigt. Und um die Datei wieder vollständig anzugeben auch die Variablendeklarationen.

Das Ganze lief darauf hinaus, doch die ganze XML-Datei (nur in einer anderen Form) im Speicher zu behalten. Warum also nicht auf bestehende Datenstrukturen zurückgreifen. Die bestehende Variablenstruktur und die bereits implementierten Parser wurden entfernt und durch ein Document Object Model (DOM) ersetzt. Dieses DOM ist die direkte Repräsentation der XML-Struktur im Speicher, und es stehen bereits Parser bereit, diese auszulesen. Um den Zugriff auf die Werte und die Struktur dann doch noch etwas zu vereinfachen ist das DOM in die Klasse `VariablesStructure` gepackt, welche Methoden bereitstellt um das DOM nach unseren Vorstellungen und Ansprüchen zu bearbeiten.

Also kommen wir nun zur Funktionsweise. Um bestimmte Aufgaben für Werte und Variablen zu erledigen, brauchen wir die Deklaration der Variablen. Um diese nicht jedesmal aus der XML-Struktur auslesen zu müssen, machen wir dies nur einmal und speichern uns für jede Variable die Informationen in der Klasse `Variable`. Diese bringt die folgenden Attribute mit:

- `static enum TYPE`
  Dieser Aufzählunstyp unterscheidet die Variablen in ihrem Typ.

- `static enum FLOW`
  Dieser Aufzählunstyp unterscheidet die Variblen nach ihrer Sichtbarkeit / Flussrichtung.

- `Variable (name, flow, type, storage, size)`
  Der Konstruktor extrahiert die Deklaration aus den Zeichenketten und erzeugt ein neues Objekt.

- `convertValue (value)`
  Diese Methode überprüft, ob der gegebene Wert zu der aktuellen Variablen passt. Die Methode versucht einen nicht passenden Wert so zu erweitern, dass er trotzdem zur Variablen passt. Dadurch wird dem Benutzer die Eingabe von Werten erleichtert.

- `getStandardValue ()`
  Gibt den Standardwert zu der aktuellen Variablen zurück.

Die Klasse `VariablesStructure` lässt sich am einfachsten über die wichtigsten Methoden beschreiben. Allerdings hat sie noch ein paar kleine Eigenheiten. Der Wert `lastPastStep` gibt den letzten Schritt an, der in den "past"-Bereich gehört. Nach dem Erstellen, ist das der größte Wert eines Schritts unter dem "past"-Knoten. Wenn allerdings Werte an Variablen zugewiesen werden, wird die Struktur nicht direkt aktualisiert, sondern nur `lastPastStep` entsprechend gesetzt. Die Struktur wird erst dann aktualisiert, wenn dies nötig ist, also bevor die Struktur in eine Datei geschrieben wird. `lastStep` merkt sich einfach den größten existierenden Schritt. Die Methoden der Klasse sind die Folgenden:

- `VariablesStructure (file)`
  Der Konstruktor liest aus der angegebenen Datei die XML-Struktur aus und überprüft direkt, ob die Struktur mit der von uns erwarteten übereinstimmt: Das heißt

  - alle Elemente existieren, die benötigt werden
  - in jedem Schritt alle Variablenwerte genau einmal angegeben sind
  - alle Variablen auch deklariert sind
  - alle Schritte in der richtigen Reihenfolge und lückenlos angegeben sind

  Zusätzlich bestimmt die Methode noch die Werte `lastStep` und `lastPastStep`. Der erste gibt an, bis zu welchem Schritt überhaupt Werte angegeben sind und der zweite gibt den letzten Schritt an, der schon simuliert ist.

- `appendSteps (lastStepToAppend)`
  Der Benutzer soll natürlich die Möglichkeit haben so viele Schritte zu simulieren, wie er möchte. Dazu muss es möglich sein, Schritte anzuhängen. Durch diese Methode wird die Struktur bis zum angegebenen Schritt erweitert. Das bedeutet, dass Zustandsvariablen mit ihrem letzten Wert und

Ereignisvariablen mit ihrem Standardwert fortgesetzt werden. Diese Methode hängt nur Knoten im “future”-Bereich der XML-Struktur an, da die Schritte ja noch nicht simuliert sind.

- `getStepElement (step)`
  Um auf einen Knoten eines Schrittes zuzugreifen, wird intern diese Methode verwendet. Der Vorteil ist, dass die Schritte in der korrekten Reihenfolge vorliegen. Das bedeutet, dass wir beim Suchen bei dem Index `step` in der Knotenliste beginnen können und damit eigentlich immer richtig liegen sollten.

- `getValue (String VariableName, int step)`
  Über diese Methode kann auf Werte einer Variablen zugegriffen werden. Diese Methode erweitert die XML-Struktur mittels `appendSteps ()` auf den entsprechenden Schritt, falls dieser noch nicht existiert.

- `setValue (variableName, step, value)`
  Um einen Wert zu setzen, wird diese Methode aufgerufen. Sie setzt dann den Wert für die angegebene Eingabevariable im angegebenen Schritt. Dabei greift sie auf die Funktion `convertValue ()` der Klasse `Variable` zurück, um den Wert zu überprüfen und anzupassen.

- `updateXMLStructure ()`
  Entsprechend der oben erwähnten Variablen `lastPastStep` wird durch diese Methode die XML-Struktur aktualisert. Dazu werden alle Knoten, die einen Schritt darstellen, der größer als `lastPastStep`, vom “past”-Knoten in den “future”-Bereich verschoben und alle Werte, bis auf die Werte der Eingabevariablen, entfernt.

- `writeSubtreeBack (file, lastWriteStep)`
  Diese Methode schreibt die XML-Struktur in die angegebene Datei. Dabei werden nur die Schritte bis zu `lastWriteStep` berücksichtigt.

- `importMissingFutureValues (variablesStructure)`
  Nach einer Simulation wird das Ergebnis der Simulation aus der neu erstellten Datei ausgelesen. Es kann allerdings sein, dass nur ein Teil der in der Oberfläche angezeigten Schritte simuliert wurde. Also stehen auch nur die simulierten Schritte in der Datei. Um nun die anderen Werte, die der Benutzer schon eingegeben hat, nicht zu verwerfen, wird mit dieser Methode aus der alten Struktur die mehr vorhandenen Werte von Eingabevariablen ausgelesen und hinzugefügt.

- `clearVariables ()`
  Um dem Nutzer dann doch noch die Möglichkeit zu geben, seine Werte zu löschen und eine Simulation neu anzufangen, können mit dieser Methode alle vorhandenen Werte gelöscht werden.

Um die Werte aller Variablen anzuzeigen, ist prinzipiell genausoviel Aufwand nötig wie bei dem ersten Ansatz. Allerdings ist der Quellcode durch diese Variante übersichtlicher und einfacher zu verstehen geworden.

6.2.4 Hauptkomponenten

Wir werden nun den Aufbau der Klassen beschreiben, die den Hauptteil der Simulationsoberfläche ausmachen.

Die beiden Klassen `SimulatorViewPart` und `TableManager` bilden den für den Benutzer sichtbaren Teil. Erstere ist von `ViewPart` abgeleitet, um sich in die Obferfläche der Eclipse Umgebung einzupassen. Desweiteren implementiert sie die Schnittstelle `Runnable` und damit eine Methode `run ()`, die in einem anderen Thread ausgeführt werden kann. Das ist nötig, da in der Eclipse-Umgebung Oberflächenelemente nur aus einem bestimmten Thread (SWT) heraus verändert werden können und die Oberfläche über das Ende eines Simulationsvorganges aus einem anderen Thread benachrichtigt wird. Die Oberfläche könnte sonst nicht die Simulationsergebnisse anzeigen. Die weiteren Methoden werden im Folgenden beschrieben:

- `saveState (memento)`
  Diese Methode wird aufgerufen, wenn die Eclipse-Umgebung beendet wird und die Oberfläche noch geöffnet ist. Innerhalb dieser Methode werden die wichtigsten Daten, welche zur Wiederherstellung der Oberfläche benötigt werden, in dem `IMemento`-Objekt gespeichert. Für unsere Oberfläche ist das der Name der Datei und der Name des zugehörigen Projektes.

- `init (site, memento)`
  Nach dem Neustart der Eclipse-Umgebung wird diese Methode benutzt, um den Zustand beim Schließen wiederherzustellen. Das übergebene `IMemento`-Objekt enthält die gleichen Daten, die beim Beenden (siehe `saveState()`) gespeichert wurden.

- `createPartControl (parent)`
  Die Methode veranlasst das Objekt, seine Oberfläche zu erstellen. Dazu wird ein `Composite`-Objekt übergeben, in das die weiteren Teile der Oberfläche eingebettet werden. `Composite` fungiert bei der Eclipse-Umgebung als eine Art Container, die weitere Elemente einer grafischen Oberfläche aufnehmen kann. Die Komponenten der Oberfläche werden hier erstellt. Um die Darstellung der Tabellen kümmert sich der `TableManager`. Beim Erstellen ist zu unterscheiden, ob es sich um einen Neustart der Eclipse-Umgebung mit Widerherstellung der Simulationsoberfläche, oder um einen Start aus der laufenden Umgebung heraus handelt. Denn beim Neustart muss noch ein Abgleich mit dem Quartz-Programm geschehen, da sich die Dateien mittlerweile geändert haben könnten und nicht mehr zueinander passen müssen. Der Abgleich kann jedoch nicht automatisch passieren, da die Elemente der Eclipse-Umgebung, die dazu nötig sind, zu diesem Zeitpunkt noch nicht existieren müssen.

- `feedback (response)` und `run ()`
  Diese beiden Methoden sind zusammen zu nennen. Wie schon erwähnt können Änderungen nicht aus jedem Thread heraus vorgenommen werden.

Über die Methode `feedback()` benachrichtigt der `Simulator` (hier ist ein Objekt der Klasse `Simulator` gemeint, die weiter unten beschrieben wird) die Oberfläche über Veränderungen. Dies kann zum Beispiel das Ende eines Simulationsvorganges sein oder auch eine Veränderung an dem Quartz-Programm. Nun wird `response` zwischengespeichert und `run ()` in dem entsprechenden Tread (JWT) aufgerufen. `run ()` kann nun auf die übergebene Nachricht zurückgreifen und entsprechend die Oberfläche aktualisieren.

- changeState (state)
  Nach dem Aufruf von `feedback (response)` und `run ()` wird in der Regel der Zustand der Oberfläche verändert. Dabei werden dem Benutzer andere Bedienelemente verfügbar gemacht. Auf die Zustände wird weiter unten noch genauer eingegangen.

Die Klasse `TableManager` verwaltet zwei Tabellen um die Variablen und die Werte für jeden Schritt darzustellen. Dabei besitzt die eine Tabelle nur eine einzige Spalte, in der die Namen der Variablen stehen. In der zweiten Tabelle werden nur die Werte für jeden Schritt angezeigt, wobei die Schritte als Spalten dargestellt sind. Diese Trennung ist notwendig, um das Scrollen durch alle Schritte zu ermöglichen und dabei immer die Spalte mit den Variablennamen zur besseren Zuordnung sichtbar zu haben. Weitere Methoden von `TableManager` sind:

- `setContent (variablesStructure)`
  Diese Methode zeigt mit Hilfe der übergebenen `variablesStructure` die Variablen und deren Werte in der Tablelle an, ohne diese neu zu erstellen. Zusätzlich werden diverse Beobachter erstellt.

- `updateInput (variablesStructure)`
  Wird nach der Simulation aufgerufen und aktualisiert in der Tabelle die lokalen Variablen und Ausgabevariablen.

- `createColumnsAndItems ()`
  Erstellt die benötigten Spalten und Zeilen in der Tabelle und löscht nicht Benötigte.

- `fillTableNames ()`
  Hiermit werden die Variablennamen in die dafür vorgesehene Tabelle eingetragen. Wenn die lokalen Variablen ausgeblendet sind, werden diese nicht in die Tabelle eingetragen beziehungsweise aus der Tabelle gelöscht.

- `fillTableInput ()`
  Diese Methode trägt die Werte aller Eingabevariablen in die dafür vorgesehenen Felder ein, wobei die Farben und Zeichenketten für Boolesche Werte berücksichtigt werden.

- `fillTableOutput (steps)`
  Diese Methode trägt die Werte aller lokalen Variablen und Ausgabevariablen in die dafür vorgesehenen Felder ein, wobei die Farben und Zeichenketten für Boolesche Werte berücksichtigt werden. `steps` gibt dabei an, wie

weit die Variablen definert sind. Zum Anzeigen der lokalen Variablen wird die Funktion `fillTableLocal()` benutzt.

- `fillTableLocal ()`
  Hiermit werden die Werte der lokalen Variablen mit Berücksichtigung der Einstellungen angezeigt oder aus der Tabelle entfernt.

- `update (inputStartStep, outputStartStep)`
  Sobald Änderungen an der Tabelle vorgenommen werden müssen, wird diese Methode aufgerufen. Sie füllt die Tabellen mit Werten und entfernt die lila und roten Markierungen. Wobei für Eingabevariablen vom Schritt `inputStartStep` und bei den restlichen Variablen vom Schritt `outputStartStep` gestartet wird. Diese Startmarkierungen haben wir eingeführt, um die Laufzeit zu verkürzen, da viele Teile der Tabelle nicht oder nur selten geändert werden müssen. Zum Beispiel ändern sich die Eingabevariablen während der Simulation nicht. Somit müssen nur die restlichen Variablen aktualisiert werden.

- `handleChange (column)`
  Diese Methode wird aufgerufen, wenn der Benutzer Änderungen an den Eingabevariablen. Sie färbt ungültig gewordene Schritte lila ein und fügt, falls der letzte angezeigte Schritt geändert wurde, neue Schritte hinzu.

- `addListeners ()`
  Fügt zu der Tabelle eine Ereignisbehandlung hinzu, um auf Tastatur- und Mauseingaben zu reagieren. Damit wird es ermöglicht, mit Maus und Tastatur in der Tabelle zu navigieren und auf einfache Weise Werte zu ändern.

- `handleSelection (index)`
  Hiermit wird in beiden Tabellen das Element mit dem Index `index` ausgewählt.

- `addListenerForModification (editor, text)` und `acceptUserInput (text)`
  Die erste Methode wird aufgerufen, wenn ein Variablenwert geändert werden soll. Sie fügt dem übergebenen Textfeld `Listener`-Objekte hinzu, die bestimmte Benutzereingaben registrieren. Anhand Dieser wird unterschieden, ob die Eingaben verworfen oder durch die Methode `acceptUserInput ()` der entsprechenden Variablen zugewiesen werden.

- `select (searchPattern, searchOffset)` und `searchItem (startRow, searchString)`
  Die Methode versucht aus dem übergebenen `searchPattern` eine Suchbedingung abzuleiten. Diese ist durch ein Teilwort eines Variablennamens und/oder einer Zahl, die einen Schritt angibt, gegeben. Dann wird versucht, den entsprechenden Schritt und die Variable zu markieren. Die Suche beginnt an der aktuellen Stelle, die um `searchOffset` veschoben wird. Das Suchen der Variablen übernimmt die Methode `searchItem ()`.

- `switchShowLocalVariables (show)`
  Blendet die lokalen Variablen aus beziehungsweise ein.

- `loadPreferences ()`
  Diese Methode läd die gespeicherten Einstellung nach dem Starten der Oberfläche.

Wir haben nun die Klassen kennengelernt, die sich um die Darstellung der grafischen Elemente kümmern. Es gibt aber noch einiges an Verwaltungsarbeit zu erledigen, die wir in die Klasse `Simulator` ausgelagert haben. Hier wird auch die Kommunikation mit dem Simulator über das Dateisystem oder über den Averest-Server geregelt. Die Klasse kümmert sich also darum, einen Simulationsvorgang auszuführen und danach die Variablen bereitzustellen. Treten Fehler auf, werden diese hier abgefangen. Die Klasse besitzt folgende Methoden:

- `Simulator (simulatorViewPart)`
  und `Simulator (simulatorViewPart, filename, projectname)`
  Die beiden Konstruktoren erstellen jeweils ein neues `Simulator`-Objekt. Sie finden allerdings in verschiedenen Situationen Anwendung. Der Erste wird aus dem laufenden Betrieb von Eclipse aufgerufen, wenn der Benutzer ein Simulationsfenster erstellt. Der Simulator nimmt die Datei, die aktuell im Editor editiert wird als Grundlage zur Simulation. Der Zweite wird aufgerufen, wenn bei einem Neustart von Eclipse direkt ein Simulationsfenster wiederhergestellt wird, dass beim Beenden noch geöffnet war. In diesem Fall bekommt der Simulator den Dateinamen und den Namen des zugehörigen Projekts übergeben.

- `getFileName ()` und `getFileProjekt ()`
  Die beiden Methoden werden von `SimulatorViewPart` verwendet um den Zustand beim Beenden von Eclipse zu speichern.

- `getVariablesStructure ()`
  Liefert die Struktur der aktuellen Variablen zurück. Diese verändern sich nach einem Simulationsdurchlauf und müssen somit von `SimulatorViewPart` neu angefordert werden.

- `notifyFileChanged ()` und `notifyFileRemoved ()`
  Beim Start wird ein `ChangeTracker`-Objekt bei Eclipse registriert, um über Änderungen benachrichtigt zu werden. Von diesem werden die beiden Methoden aufgerufen, wenn sich die zu simulierende Datei oder eine ihrer `include`-Dateien verändert hat oder gelöscht wurde. Daraufhin wird die Oberfläche aktualisiert, da die Simulationsergebnisse nun nicht mehr korrekt sein müssen beziehungsweise keine Simulation mehr stattfinden kann.

- `simulateToStep (lastSimulationStep)`
  Durch diese Methode wird die Simulation gestartet. Dabei wird die XML-Struktur bis zum Schritt `lastSimulationStep` in die Simulationsdatei geschrieben. Dann wird entschieden, ob der Averest-Server benutzt werden,

oder die Simulation lokal ausgeführt werden soll. Anschließend wird ein neues `JobSequence`-Objekt erstellt und die Simulation entsprechend gestartet. Zusätzlich wird dem `JobSequence`-Objekt noch ein `SimulationCallback` übergeben, der über das Ende der Simulation benachrichtet.

- `updateVariables()`
  Diese Methode erstellt ein neues Objekt vom Typ `VariablesStructure` aus der aktuellen Simulationsdatei. Danach wird für dieses Objekt die Methode `importMissingFutureValues ()` mit der alten Variablenstruktur aufgerufen, um die vorhandenen Eingabewerte zu übernehmen.

6.2.5 Weitere Klassen

Es gibt noch weitere Klassen, die zwar keine notwendige Funktionalität implementieren, die aber das Leben einfacher machen und die Oberfläche besser in das Eclipse-Plugin integrieren.

- `ChangeTracker`
  Diese Klasse implementiert das Interface `IResourceChangeListener` und Objekte der Klasse werden beim Eclipse-Workspace registriert, damit sie über Änderungen der Quartz-Dateien informieren. Das Objekt bekommt ein `IncludeFile`-Objekt übergeben, um zu überprüfen, ob die Datei selbst, oder eine der Dateien, die in ihr eingebunden werden, von der Änderung betroffen sind.

- `TableManagerPreferenceChangeListener`
  Diese Klasse registriert sich bei den Averest-Einstellungen und überwacht, ob Änderungen an den Einstellungen vorgenommen werden. Ist dies der Fall wird überprüft, ob der `TableManager` von diesen Einstellungen betroffen ist. In dem Fall wird der `TableManager` dazu veranlasst, die Tabellen, entsprechend den neuen Einstellungen zu veränden.

- `VCDExport`
  Die Klasse `VCDExport` bietet die Funktionalität um aus den Variablen in der XML-Struktur eine VCD-Datei zu erstellen. Dabei wird zuerst ein neues Objekt erstellt, dass die Variablenstruktur übergeben bekommt. Anschließend kann dieses Objekt die VCD-Datei bis zu dem letzten vollständig angegebenen Schritt (siehe `lastPastStep` in `VariablesStructure`) erstellt werden.

6.2.6 Verhalten der Oberfläche

Wir haben nun die Klassen und deren Methoden der Simulationsoberfläche weitgehend kennengelernt und wir haben gesehen, wie der Simulator mit der Oberfläche kommuniziert. Während der Benutzung der Oberfläche können aber andere, zum Teil unvorhergesehene Dinge passieren, die das Verhalten der Oberfläche beeinflussen. Deshalb möchten wir in diesem Abschnitt die Zustände der

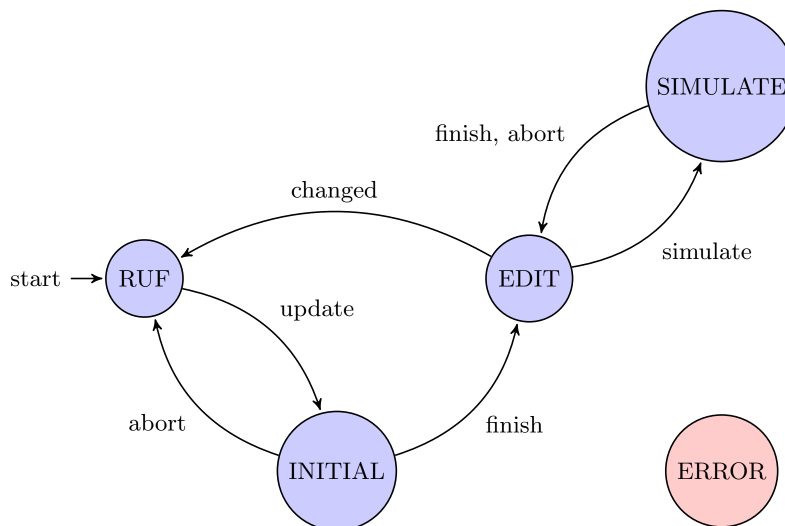

Abbildung 6.5: Zustandsübergangsdiagramm der Oberfläche

Oberfläche beschreiben und zeigen, wann welcher Zustand angenommen wird. Dieser Teil soll auch zum besseren Verständnis der inneren Abläufe beitragen.

Die Oberfläche stellt dem Benutzer nicht immer alle Kontrollelemente zur Verfügung, sondern immer nur diese, die er im Moment auch benutzen darf. So kann der Benutzer beispielsweise, wenn eine Simulation läuft, keine Änderungen an den Werten vornehmen. Welche Bedienelemente dem Benutzer verfügbar gemacht werden, lässt sich aus Zuständen ableiten, die wir in Abb. 6.5 grafisch dargestellt haben. Das Diagramm zeigt auch die Verbindung der einzelnen Zustände.

Die Zustände der Oberfläche sind durch die Knoten dargestellt. Nachrichten, die `Simulator` an `SimulatorViewPart` schickt, sind die Beschriftungen der Kanten. Wir beginnen im Zustand "REQUIRE UPDATE FILE" (RUF), indem wir noch keine Variablennamen kennen, da die Oberfläche gerade gestartet wird. Normalerweise warten wir in diesem Zustand darauf, dass der Benutzer den Knopf zum aktualisieren der Datei drückt. Beim Start der Oberfläche wird allerdings direkt ein `Simulator`-Objekt instanziiert, das dann automatisch die initiale Simulation startet und die Oberfläche wechselt in den Zustand "INITIAL". Dort wird gewartet, bis die Antwort vom Simulator eintrifft und die Variablen ausgelesen wurden. Dann wird das Ende der Simulation bekannt gegeben und in den Zustand "EDIT" gewechselt um den Benutzer Werte ändern zu lassen. Tritt bei der Simulation oder beim Auslesen der Variablen ein Fehler auf, wird zurück in den Zustand "REQUIRE UPDATE FILE" gewechselt.

Im "EDIT"-Zustand nimmt der Benutzer Änderungen an der Variablenbelegung vor, bis er auf den Knopf zur Simulation drückt, oder sich die Quartz-Datei, die simuliert wird, ändert. Wenn die Datei sich ändert springt die Oberfläche

wieder in “REQUIRE UPDATE FILE”. Wir eine Simulation gestartet geht Sie in den Zustand “SIMULATE”, bis die Simulation beendet ist oder abgebrochen wird.

Der Zustand “REQUIRE UPDATE FILE” stellt also sicher, dass nach einer Änderung an der Quartz-Datei keine Simulation stattfinden kann, bis die Oberfläche wieder neu die Variablennamen eingelesen hat. Das Aktualisieren wird dem Benutzer überlassen, da er auf diese Weise selbst entscheiden kann, wann er aktualisieren will. Desweitern würden die vorhandenen Simulationsergebnisse überschrieben werden.

Der “ERROR” Zustand ist angedeutet und kann eignetlich aus jedem anderen Zusatnd aus erreicht werden, wenn es ein schwerwiegendes Problem gibt. Das ist der Fall, wenn die Quartz-Datei gelöscht wird oder wenn nach dem Neustart von Eclipse das Projekt nicht mehr existiert. Also Fälle, in denen auf keinen Fall eine weiter Simulation möglich ist. Dieser Zustand wird dann auch nicht mehr verlassen und die Oberfläche muss geschlossen werden. Die Kanten in den “ERROR”-Zustand sind wegen der Übersichtlichkeit weggelassen worden.

Kapitel 7

Zusammenfassung

7.1 Zusammenfassung

Ziel der Projektarbeit war es, das Verhalten von Programmen der synchronen Sprache Quartz im Averest-Eclipse-Plugin darzustellen. Dazu wurde der Simulator im Werkzeug Ruby benutzt und dessen Ausgaben tabellarisch aufbereitet. Zusätzlich wurde es ermöglicht kompfortabel Änderungen vorzunehmen und Simulationen zu starten. Während der Programmierung und Nutzung unserer Simulationsoberfläche, sind uns noch hilfreiche Erweiterungen eingefallen. Wir haben zum Beispiel die Möglichkeit der farbigen Hervorhebung Boolescher Werte hinzugefügt, um die angezeigten Daten übersichtlicher zu gestalten.

Wir haben großen Wert darauf gelegt, die Simulationsoberfläche nicht vom AIF-Format abhängig zu machen. Zum einen da dieses zur Zeit der Arbeit Änderungen unterlegen ist, zum anderen ist die Oberfläche jetzt unabhängig von der Spache Quartz. Man kann die Oberfläche somit einfach erweitern um auch andere Daten, die in diesem Format vorliegen, anzuzeigen.

Mit unserer Oberfläche kann der Benutzer die Verhaltensweisen in unterschiedlichen Situationen ausprobieren und sehen wie das Programm auf die verschiedensten Eingaben reagiert.

7.2 Ausblick

Ein Nachteil des Simulators ist es, dass er noch keine Arrays unterstützt. Diese sind aber in Quartz vorhanden. Diese Arbeit hat sich deshalb nicht mit der Darstellung der Arrays in der Oberfläche beschäftigt. Allerdings wäre es Sinnvoll diese Erweiterung einzubauen, wenn der Simulator sie unterstützt und es wäre dann zu klären, wie Arrays in der Tabelle dargestellt werden könnten.

In dem Abschnitt über die Kausalitätsanalyse haben wir gesehen, wie synchrone Programme simuliert werden können und wie mittels der Fixpunktiteration die Werte für jeden Schritt bestimmt werden. Es wäre denkbar den Simulator so zu erweitern, dass er seine Zwischenergebisse, also die einzelnen

Durchläufe der Iteration, auch in die Simulationsdatei schreibt. In der Oberfläche könnte man dann das Anzeigen der Zwischenschritte erlauben und man könnte in jedem Schritt nachfolziehen, wie genau ein Wert Zustande kommt.

Bei der Verifikation ist es bei manchen Programmen möglich, sich ein Gegenbeispiel ausgeben zu lassen, wenn die Spezifikation nicht erfüllt wird. Es wäre gut, dieses Gegenbeispiel in den Simulator laden zu können, um es genauer zu studieren und auch ausprobieren zu können, wie sich Programmänderungen darauf auswirken.

Literaturverzeichnis

[1] Averest homepage. http://www.Averest.org/.

[2] G. Berry. The Esterel v5 language primer, July 2000.

[3] Eclipse homepage. http://www.Eclipse.org/.

[4] IEEE Computer Society. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*. New York, USA, 2001. IEEE Std. 1394-2001.

[5] K. Schneider. Averest interchange format. http://www.averest.org/documentation/aif.pdf, 2006.

[6] K. Schneider. The synchronous programming language Quartz, 2007.

[7] K. Schneider. Vorlesung: System description languages. http://rsg.informatik.uni-kl.de/teaching/syslang/, WS06/07.

[8] K. Schneider and T. Schuele. Averest: Specification, verification, and implementation of reactive systems, 2005.

[9] K. Schneider and M. Wenz. A new method for compiling schizophrenic synchronous programs. In *Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 49–58, Atlanta, Georgia, USA, 2001. ACM.

[10] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.