Studienarbeit an der

Eberhard-Karls-Universität Tübingen
Wilhelm Schickard-Institut für Informatik

Betreuer: Tommy Kuhn

Simulation digitaler Systeme auf RT-Ebene in Java

Roland Weiss

28. November 1997

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation und Zielsetzung

Die Simulation digitaler Systeme gewinnt aufgrund der extrem hohen Integrationsdichten moderner Chips, und dem damit einhergehenden Komplexitätsanstieg der realisierbaren Schaltungen, immer mehr an Bedeutung. Mittels Simulation versucht man schon vor der Hardwarerealisierung eine möglichst hohe Zuverlässigkeit der VLSI-Systeme sicherzustellen. Die simulierte Entwurfsbeschreibung wird dann von entsprechender Software in eine Schaltungsbeschreibung transformiert, die Grundlage einer Chipfertigung sein kann.

Unter denen auf diesen Aufgabenbereich spezialisierten Sprachen gilt VHDL (siehe [3] und [4]) als Standard. Sie dient als konzeptuelle Vorlage für Teile unserer Arbeit. VHDL ist eine Allround-Sprache, die Schaltungsbeschreibungen auf allen Ebenen zuläßt (siehe Abschnitt 2 für eine Beschreibung der verschiedenen Abstraktionsebenen). Spezielle VHDL-Softwarepakete bestehen in der Regel aus einem Compiler, der VHDL in ein internes Format übersetzt, sowie einem darauf basierenden Simulator.

Die sehr hohe Schaltungskomplexität führt aber zu Problemen bei der Simulation. Die Simulationszeiten steigen derart an, daß das Arbeiten mit herkömmlichen Simulatoren nicht mehr effizient genug erscheint.

Es existieren unterschiedliche Auswege aus diesem Dilemma. Einerseits kann man durch eine gute Parallelisierung die Laufzeiten der Simulationen deutlich verbessern. Das Hauptproblem dieses Ansatzes besteht darin, daß VHDL, wie schon erwähnt, Verhaltensbeschreibungen auf vielen Ebenen erlaubt. Somit müssen die parallel arbeitenden Simulatoren diese auch unterstützen. Hierbei stellt die Komplexität der damit verbundenen parallelen Algorithmen, besonders auf Gatterebene, eine immense Hürde dar. Andere Simulatoren, wie z.B. QuickVHDL von Mentor Graphics, beschreiten eine weiteren Weg. Sie versuchen die Laufzeiten zu optimieren, indem sie nur eine Teilmenge von VHDL implementieren. Damit steigt die Überschaubarkeit der Algorithmen und sie können leichter optimiert werden. Es zeigt sich aber, daß auch dieser Lösungsversuch an seine Grenzen stößt, vor allem in Bezug auf die Laufzeiten.

Entwurfsbeschreibungen erfolgen heutzutage meist auf RT-Ebene, die man sehr gut auch in Programmiersprachen formulieren kann. Die Beschreibungen können direkt interpretiert, oder um die Ausführungszeiten weiter zu steigern, zuerst in Maschinencode übersetzt werden. Man spricht hier von einer Simulation durch Ausführung. Dies stellt den von uns verfolgten Ansatz dar. Es ist aber notwendig, unabhängig von der gewählten Programmiersprache, gewisse Primitive zur Verfügung zu stellen, die auf RT-Ebene vorausgesetzt werden. Sie sind üblicherweise nicht in konventionellen Programmiersprachen enthalten.

Die 90er Jahre gelten als das *objektorientierte* Jahrzehnt, zumindest aus Sicht der Programmierer. Sprachen wie C++ und Smalltalk spielen in der Industrie eine große Rolle. Im Sog des Booms des Internets etablierte sich auch eine weitere objektorientierte Sprache: Java von Sun Microsystems. David Flanagan gibt in [2] einen guten Überblick über die Ideen, die in Java verarbeitet sind. Unsere Wahl fiel ebenfalls auf Java. Die Aspekte, die dabei im Vordergrund standen, werden nun kurz aufgezählt. Java ist eine sehr einfach zu erlernende, objektorientierte Programmiersprache. Sie unterstützt Nebenläufigkeit, da sie über ein sprachimmanentes Thread-Konzept und weitreichende Netzwerkunterstützung verfügt. Immer enger verbundene, heterogene Netzwerke von Computersystemen machen einen weiteren Aspekt von Java sehr attraktiv. Java ist plattformunabhängig, da es portablen Bytecode generiert. Es ist aber trotz seines interpretativen Charakters sehr effizient, was durch die Entwicklung und den Einsatz von Just-In-Time-Compilern noch vorangetrieben wird.

Außerdem bietet unser Simulationsansatz auf der Basis von Programmiersprachen einige zusätzliche Vorteile gegenüber klassischen Simulatoren. So ist der Simulator leicht und von jedermann erweiterbar. Auch kann man direkt auf Daten der Simulation zugreifen, was z.B. in VHDL nicht so ohne weiteres machbar ist. Die Ausnutzung der nebenläufigen Elemente in Java führt zu einem schnellen, parallelen Simulator.

Inwieweit sich die Erwartungen an die Simulation digitaler Systeme auf RT-Ebene in Java erfüllen, wird sich am Ende dieser Arbeit erweisen.

1.2 Aufbau der Arbeit

Die Arbeit beginnt mit einem Grundlagenkapitel, das sich mit der Simulation digitaler Systeme auseinandersetzt. Es werden kurz die verschiedenen Abstraktionsebenen und -sichten vorgestellt, zusammmen mit den typischen Simulationsarten und -verfahren. Dabei gehen wir auch auf die bestehenden Timingmodelle ein. Abschließend zeigen wir die Grenzen dieser Simulation auf. Darauf folgt eine grobe Einführung in Konzepte von VHDL, die für unseren Simulator von Bedeutung sind.

Das nächste Kapitel befaßt sich konkret mit der Verhaltensbeschreibung in Java. Anhand eines einfachen Beispiels wird erläutert, wie man schrittweise eine solche Beschreibung aufbaut. Dann erfolgt eine Einordnung unseres Simulationskonzepts anhand der im Grundlagenteil vorgestellten Konzepte. Den Abschluß bildet eine genaue Beschreibung der dem Simulator zugrundeliegenden Algorithmen.

Danach gehen wir detailiert auf die Implementierung der Algorithmen und RT-Primitive in Java ein. Dazu werden für Signale, Datentypen, Prozesse und den Simulator die wichtigsten Datenstrukturen und Methoden betrachtet. Der Simulator selbst ist entsprechend einem Client-Server-Modell aufgebaut, was ebenfalls erläutert wird.

Abschließend wird anhand eines Beispiels untersucht, ob unser Ansatz zur Simulation digitaler Systeme erfolgversprechend ist. Dabei werden sowohl die prinzipielle Machbarkeit als auch Effizienzaspekte kritisch beleuchtet.

2 Grundlagen

Dieser Abschnitt soll kurz die Grundlagen der Simulation digitaler Systeme zusammenfassen. Dabei stützen wir uns vor allem auf die Arbeiten [5] und [9].

2.1 Begrifflichkeit

Die *Simulation* eines digitialen Schaltnetzes ist die Abarbeitung dessen vorliegender Beschreibung anhand von mathematischen Modellen (nach [9]). In userem Fall liegt eine Verhaltensbeschreibung in Java vor und die zugrundeliegenden Modelle sind in Algorithmen formuliert.

2.2 Abstraktionsebenen und -sichten

Simulationen werden auf verschiedenen Abstraktionsebenen angewandt. Sie unterscheiden sich in ihrem Detailreichtum und ihrer Transparenz für den Entwickler und Designer. Prinzipiell unterteilt man Beschreibungen in drei Kategorien, die die Sichtweise des Systems prägen.

Verhaltensbeschreibungen: geben an, wie Eingabedaten transformiert werden, welche Operationen auf ihnen ausgeführt werden und wie die Ausgabedaten aussehen, ein Algorithmus beschreibt deren funktionale Abhängigkeiten.

Strukturelle Beschreibungen: zeigen die Topologie des Systems, wie einzelne Komponenten zueinander in Relation stehen und wie sie verbunden sind.

Physikalische Beschreibungen: dienen dazu, möglichst exakte Aussagen über Bauteile und ihre Charakteristika machen zu können.

Tavangarian unterscheidet in [9] fünf Abstraktionsebenen mit spezifischen Charakteristika.

System-Ebene: Auf dieser Ebene beschreibt man ein System durch Module wie Prozessoren, Speicher und I/O-Einheiten, sowohl algorithmisch als auch strukturell.

Register-Transfer-Ebene: Die Schaltkreise auf der RT-Ebene sind aus Registern, Schaltern, Bussen und ähnlichen Bauelementen zusammengestellt. Entscheidend ist weiterhin, daß der Simulation ein Taktverhalten zugrundeliegt. Innerhalb der vorgegebenen Takte werden Zustände der einzelnen Bauteile anhand funktionaler Beschreibungen in neue Zustände überführt.

Gatter-Ebene: Die Systeme dieser Ebene basieren auf Gattern und Flipflops, sowie einer Beschreibung deren Verknüpfungen. Das zeitliche Verhalten der einzelnen Komponenten spielt eine wichtige Rolle.

Transistor-Ebene: Die Komponenten sind hier vorwiegend Transistoren und Dioden. Zur Modellierung dieser Schaltungen bedient man sich linearer oder nichtlinearer Differentialgleichungen.

Layout-Ebene: Anhand von speziellen Designvorschriften wird ein Schaltkreis auf verschiedenen Schichten aus geometrischen Symbolen aufgebaut. Die Designvorschriften beschreiben die Zusammenhänge der einzelnen Symbole und sind für eine fehlerfreies Desgin essentiell.

2.3 Simulationsarten

Je nach betrachteter Ebene unterscheidet man unterschiedliche Arten der Simulation. Wir gehen dabei „von unten nach oben“. Damit ist gemeint, daß zuerst die physikalischen Simulationen, und dann die in der Hierarchie höher liegenden, beschrieben werden.

1. Prozeß-Simulation: zur Bestimmung von Dotierungsprofilen, liefert Aussagen über räumliche Verteilung der chemischen Elemente.

2. Bauelement/Device-Simulation: liefert Aussagen über Ströme und Spannungen der Bauelemente (meist Transistoren) und deren Ersatzdarstellungen.

3. Schaltkreis-Simulation: Schaltungen aus mehreren Elmenten werden simuliert und als Ergebnis erhält man Werte über Ströme und Spannungen in Abhängigkeit von der Zeit, komplexere Schaltungen sind aus Laufzeitgründen meist nicht mehr simulierbar.

4. Timing-Simulation: stellt einen Ausweg für die auftretenden Simluationsprobleme der Schaltkreis-Simulation dar, stark vereinfachte Transistormodelle führen zu erheblich schnelleren Simulatoren, dafür sinkt die Genauigkeit der Resultate.

5. Schalter-Simulation: Transistoren werden als Schalter modelliert, dient zur Behandlung von Schaltungen mit bidirektional betriebenen Transistoren.

6. Logik- oder Gatter-Simulation: liefert Aussagen über Zeitabläufe, die Beschreibung erfolgt mittels boolscher Gleichungen.

7. Register Transfer-Simulation (RT): die Simulation besteht aus der Nachbildung von Signalausbreitungen auf den Leitungen, als Daten kommen jetzt Bitvektoren mit mehr als einem Element vor, außerdem kommen auch arithmetische Operationen hinzu, die Operationen sind an Bauteile gebunden.

8. Verhaltens-Simulation: Ziel ist eine Beschreibung des Input/Output-Verhaltens von algorithmischen Beschreibungen auf System-Ebene, oft auch mit Hilfe imperativer Sprachen, die Bindung der Operationen an Hardware-Bausteine ist aufgehoben.

2.4 Timing-Modelle

Im vorhergehenden Abschnitt zeigte sich deutlich, daß die zeitliche Beschreibung eine wichtige Rolle spielt. Auf den verschiedenen Abstraktionsebenen variiert die Bedeutung einer genauen Zeitmessung sowie deren Modellierung. Hier werden nun die gebräuchlichen Modelle vorgestellt.

Zero-Delay: hier steht eine funktionale Sichtweise der Bausteine im Vordergrund, der exakte, zeitliche Signalverlauf interessiert nicht, sondern vielmehr die Werte an Ausgängen von Gatternetzen nach einer hinreichend langen Wartezeit.

Unit-Delay: etwas genauer als das Zero-Delay Verfahren, hier wird davon ausgegangen, daß an allen Einheiten (Units) die Resultate gleichschnell zur Verfügung stehen.

Transportverzögerung: Signale weisen an den Ausgängen von Schaltungen beliebig kurze Signaländerungen auf, als Beispiel nennt Marwedel Koaxialleitungen.

Trägheitsverzögerung: Viele Schaltungen verhalten sich aufgrund physikalischer Gegenbenheiten nicht entsprechend dem Modell der Transportverzögerung, sondern unterdrücken Signalimpulse, die zu kurz sind. Daher bieten Simulatoren oft die Möglichkeit, Signifikanzschwellwerte anzugeben, ab denen eine Signaländerung erst Gültigkeit hat.

Pull-Up/Pull-Down Verzögerung: stellt eine Verfeinerung des letzten Modells dar, wobei noch die unterschiedlichen Verzögerungszeiten von steigenden und fallenden Flanken berücksichtigt werden.

Unsicherheitsintervalle: Die genauen Zeiten für Änderungen variieren teilweise, was durch gestaffelte Wertangaben simuliert werden kann. Dann erhalten Signale für Zeitintervalle undefinierte Werte, und erst ab einer Schranke den endgültig zugewiesenen.

Delta-Delay: dient dazu, Signaländerungen zu simulieren, die unterhalb der Auflösung der Zeiteinheit liegen, d.h. zwischen den Änderungen verstreicht keine Simulationszeit.

In gängigen Simulatoren sind meist mehrere dieser Modelle integriert und der Benutzer kann mit geeigneten Befehlen den jeweils gewüschten Verzögerungsmechanismus wählen.

2.5 Simulationsverfahren

Man unterscheidet zwischen laufzeitorientierten und ereignisorientierten Simulationsverfahren. Abbildung 1 liefert einen Überblick über die gängisten Verfahren.

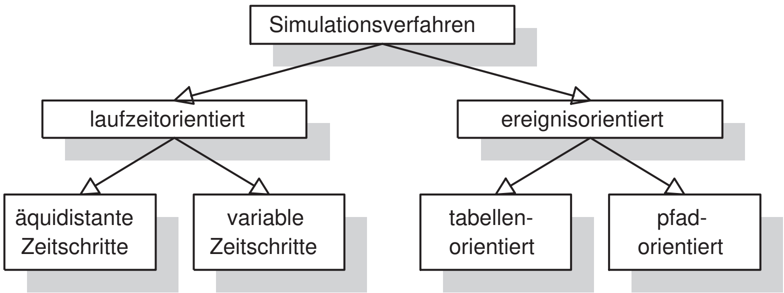

Abbildung 1: Kategorisierung der Simulationsverfahren

Die ereignisorientierten Verfahren unterteilt man noch weiter in synchron und asynchron.

2.5.1 Laufzeitorientierte Simulation

Das Hauptmerkmal ist, daß zu den Zeitpunkten eines definierten Zeitrasters alle Werte der Schaltung berechnet werden. Bei äquidistanten Zeitschritten kann ein solcher Zeitschritt maximal so groß wie die kleinste Gatterverzögerungszeit sein. Der Vorteil dieser Methode ist der einfache, daraus resultierende Algorithmus, der Nachteil die extrem langen Laufzeiten.

Um die Laufzeit zu optimieren, wählt man variable Zeitschritte. Dies geschieht dadurch, daß nach jedem Zeitschritt bestimmt wird, ob sich der Simulator in einem stationären Zustand befindet, d.h. es haben sich keine Wertänderungen ergeben. Dann kann der nächste Berechnungszeitpunkt mit doppeltem Abstand gewählt werden. Erst bei Änderungen an den Signalen muß wieder der Zeitschritt verringert und der exakte Zeitpunkt der Änderungen bestimmt werden. Der Simulationsalgorithmus ist entsprechend komlexer als im äquidistanten Fall.

2.5.2 Ereignisorientierte Simulation

Hierbei protokolliert der Simulator die auftretenden Ereignisse, das sind die Änderungen von Signalzuständen, und verfolgt deren Auswirkungen. Die Simulationszeiten sind deutlich kürzer als bei laufzeitorientierter Simulation, da eine Vielzahl von Berechnungen gespart werden.

Bei tabellenorientierten Simulationen setzt man eine Ereignisliste ein, um die zeitliche Abfolge der Ereignisse zu erfassen. Mit Hilfe einer Tabelle wird bestimmt, welche Elemente sich ändern. Diese Änderungen werden selbst wieder als Ereignisse in die Liste eingetragen. Damit werden dann neue Berechnungen angestellt.

Pfadorientierte Ereignissimulation verfolgt nur Ereignisse entlang der Pfade, also werden nur Gatter untersucht, an deren Eingänge neue Signale anliegen.

Unter synchroner Simulation versteht man Verfahren, bei denen die Fortpflanzung der Ereignisse mit festen Zeitschritten berechnet wird, ausgehend von einer Grundzeiteinheit. Die Ereignisse dürfen auch nur zu Vielfachen dieser Grundzeiteinheit stattfinden. Bei asynchroner Simulation hingegen können die Ereignisse zu beliebigen Zeitpunkten stattfinden. In einer chronologisch sortieren Liste werden die auftretenden Ereignisse eingetragen.

2.6 Grenzen der Simulation

Die Simulation digitaler Systeme liefert meist zufriedenstellende Resultate, die als Basis für Hardwarerealisierungen dienen können. Aber es treten auch Probleme auf:

1. Signale in Verbindung mit Gatterlaufzeiten stellen oft ein Problem dar, daß vom Simulator nicht erkannt wird. Dann liefert er falsche Resultate, ohne Hinweise auf kritische Situationen. Ein beliebter Ausweg ist das Vorgeben von korrekten Komponeten, die experimentell bestimmt wurden.

2. Die Parameter, die den mathematischen Simulationsmodellen zugrundeliegen, sind teilweise nicht genau genug bekannt oder variieren zwischen verschiedenen Produktionsläufen. Dies kann zu Verfälschungen der Simulationsergebnisse führen.

3. Die Simulatoren vernachlässigen die Tatsache, daß die Komponeten mit elektrischer Leistung versorgt werden müssen. Beim gleichzeitigen Schalten vieler Transistoren können Spannungsschwankungen auftreten, die zu fehlerhafter Funktionsweise der Komponenten führen.

2.7 Grundlagen von VHDL

In den folgenden Abschnitten folgt eine kurze Beschreibung der wichtigsten VHDL-Konzepte, die ein Arbeiten auf der von uns anvisierten RT-Ebene ermöglichen. Sie beziehen sich auf [4] und [3].

2.7.1 Designeinheiten

In VHDL beschreibt eine Entity die externe Sichtweise des Entwurfs. Darunter fallen vor allem die Input- und Output-Ports. Eine Entity kann verschiedene interne Realisierungen besitzen, die jeweils in einer Architecture realisiert sind.

2.7.2 Datentypen

VHDL ist eine Sprache, die auf einem strikten Typsystem basiert. Über Packages können die standardmäßig vorhandenen Typen ergänzt werden. Zum Standard-Package gehören Typen wie `Character`, `Bit`, `Integer`, `Real`, `String` und `Bit_Vector`. Es können auch Untertypen gebildet werden, normalerweise durch Einschränkung des Wertebereichs.

2.7.3 Anweisungen

VHDL liefert dem Programmierer viele Befehle, die er von normalen Programmiersprachen gewöhnt ist. Dazu zählen Bedingungen, Schleifen und Variablenzuweisungen.

Hinzu kommen aber noch zwei Kategorien von Anweisungen, die im Zusammenhang mit Konzepten Bedeutung haben, auf die gleich näher eingegangen wird. Dabei handelt es sich um:

1. Anweisungen zur Synchronisation von konkurrenten Beschreibungen
2. Anweisungen zur Manipulation von Signalen

Eine Entity beschreibt ihre externe Sichtweise über sogenannte *Ports*, die genau genommen externe Signale darstellen. Interne Signale werden hauptsächlich dazu verwendet, *Prozesse* zu synchronisieren.

2.7.4 Konkurrentes Verhalten mittels Prozessen

Die bisher genannten VHDL-Konzepte haben gleichwertige Entsprechungen in Programmiersprachen, was sich auch in den vorhandenenen Konstrukten in Java widerspiegelt. Prozesse und die eng damit verwandeten Signale hingegen sind typisch für Simulationen auf RT-Ebene. In den Prozessen erfolgt die eigentliche Verhaltensbeschreibung.

Prozesse werden eingesetzt, um gleichzeitige Aktivitäten von Hardwarekomponenten zu simulieren. Sie werden am Simulationsanfang alle gleichzeitig gestartet, danach arbeiten sie die meiste Zeit autonom. Nur bei Zugriffen auf Signale treten sie in Wechelwirkung miteinander. Prozesse haben vorwiegend lokale Daten. Ihre Kommunikation erfolgt über Signale, die für alle Prozesse zugänglich sind.

Zur Synchronisation von Prozessen dienen `wait`-Statements, die zwei Aufgaben haben. Erstens aktivieren sie zuvor gemachte Signalzuweisungen und zweitens lassen sie eine bestimmte Anzahl von Takteinheiten verstreichen. Es gibt sie in vier Varianten:

1. `wait;`
2. `wait for` *n* `ns;`
3. `wait on` *sensitivity list*`;`
4. `wait until` *condition*`;`

Die erste wird verwendet, um einen Prozeß zu beenden, aber nicht ohne vorher die Signalzuweisungen zu aktivieren. Sie ist nützlich, um Initialisierungsprozesse abzuschließen. Der `wait for`-Befehl erhöht die Simulationszeit des Prozesses um die angegebene Anzahl von Takteinheiten. Der `wait on`-Befehl suspendiert die Ausführung eines Prozesses, bis es zu einer Wertänderung an einem in der *sensitivity list* eingetragenen Signal kommt. Der letzte Befehl schließlich wartet, bis die Bedingung *condition* wahr wird.

2.7.5 Signale

Signale dienen als Verbindungsstücke zwischen einzelnen Elementen struktureller Beschreibungen. Darüberhinaus, und vielleicht noch wichtiger, stellen sie den einzigen

Kommunikationsweg zwischen Prozessen dar. Innerhalb der Simulation wird über Signale eine zeiliche Ordnung geschaffen, denn die Werte von Signalen sind immer mit einem Zeitpunkt assoziiert. Die Ausgabe von Signalen erfolgt meist über ihren *Signalverlauf*, Abbildung 2 zeigt einen typischen Signalverlauf eines Clocksignals.

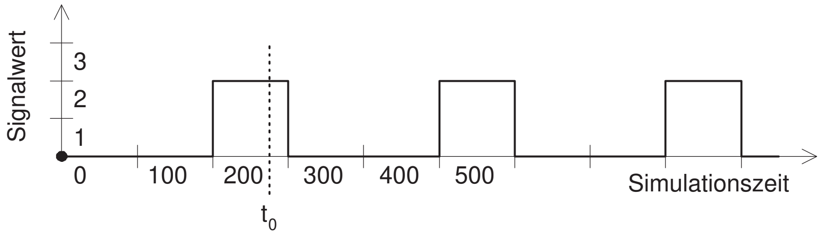

Abbildung 2: Signalverlauf

Um Signalzuweisungen von Variablenzuweisungen unterscheiden zu können, verwendet man in VHDL zwei unterschiedliche Symbole. Signalzuweisungen erfolgen mit `<=`, Variablenzuweisungen mit `:=`.

Signalzuweisungen werden aufgrund ihrer zeitlichen Komponente nicht sofort gültig, sondern erst bei der nächsten `wait`-Anweisung, wie zuvor erwähnt. In VHDL sind zwei unterschiedliche Verzögerungsmechanismen implementiert:

1. inertial Delay (die Standardeinstellung)
2. transport Delay

Mit transport Delays werden Hardwarekomponenten modelliert, die auf Pulsänderungen in unendlich kleinen Zeitschritten reagieren. Inertial Delays hingegen simulieren Schaltelemente, deren Pegelwechsel erst nach einem bestimmten Zeitlimit erfolgt.

3 Verhaltensbeschreibung mit Java

In diesem Abschnitt wird detailiert beschrieben, wie die Verhaltensbeschreibung in Java aussehen soll und welche Primitive und Konzepte dazu benötigt werden. Dabei konzentrieren wir uns auf eine Betrachtung auf RT-Ebene, da ab dieser eine Verhaltensbeschreibung mittels einer imperativen Programmiersprache erst sinnvoll erscheint. Eine genaue Beschreibung der zugrundeliegenden Klassen folgt dann in Abschnitt 4, der sich eingehend mit deren Implementierung befaßt.

3.1 Wahl der Sprache

Zuvor soll aber kurz erläutert werden, weshalb Java überhaupt als Implementierungssprache gewählt wurde. Ausgehend von unserem Ansatz, die Verhaltensbeschreibung eines digitalen Systems in einer Programmiersprache zu formulieren, ist die erste Forderung die, daß es sich um eine objektorientierte Sprache handeln soll. Die Vorteile sind unter anderem ein klares Design, Datenkapselung, Wiederverwertbarkeit und gute Erweiterbarkeit.

Als Standardsprache für objektorientierte Programmierung hat sich in der Industrie C++ etabliert. Sie zeichnet sich durch sehr schnellen Code und einen sehr großen

Sprachumfang aus. Dem gegenüber steht die damit verbundenen Komplexität der Sprache. C++ ist aufgrund seiner endlosen Zahl an syntaktischen und semantischen Sonderfällen gefürchtet. Die Pointerarithmetik in C++ ist der Ursprung vieler Fehler und wird regelmäßig scharf kritisiert.

Unser Simulator soll als Studie dienen, inwiefern Verhaltensbeschreibung in einer imperativen Programmiersprache realisierbar sind. Deswegen haben wir uns aus den soeben genannten Gründen gegen C++ entschieden. Java besticht aufgrund seiner extrem kurzen Einarbeitungszeiten, was auch daran liegt, daß es syntaktisch an C++ angelehnt ist, aber dessen ganze Sonderfälle über Bord wirft und Pointermanipulationen verbietet. Ebenfalls positiv wirkte sich aus, daß Javaprogramme mit Threads leicht parallelisierbar sind und diese zum Sprachumfang gehören. Somit fiel unsere Wahl auf Java als *Referenzsprache* für unsere Studie.

Man sollte aber festhalten, daß die Implementierung prinzipiell auch in einer anderen objektorientierten Sprache erfolgen kann, wie C++, Ada95 oder Smalltalk. Wenn die von uns vorgestellten Algorithmen sich in Java bewährt haben, sollte einer Migration nach einer anderen Sprache nichts im Wege stehen.

3.2 Zielsetzung

Das Designziel ist, einerseits weitestgehend den Richtlinien objektorientierter Programmierung zu folgen und sich bekannter Idiome Javas zu bedienen, andererseits aber auch VHDL-Umsteigern keine allzu feindliche Umgebung zu präsentieren. Die Zugeständnisse an VHDL erscheinen sinnvoll, wenn man dessen immense Bedeutung berücksichtigt. In VHDL spiegelt sich die jahrelange Arbeit im Bereich der Schaltungssimulation wider, und wir sollten versuchen, die für uns wichtigsten Konzepte zu übernehmen. VHDL-Simulatoren dienen uns auch als Vergleichsobjekte in Bezug auf die Korrektheit und die Effizienz unserer Implementierung.

3.3 Migration von VHDL-Konzepten nach Java

Im Grundlagenkapitel wurden die für eine Simulation auf RT-Ebene zentralen Konzepte und Komponenten von VHDL erläutert. Nun wird deren Migration nach Java dargestellt.

3.3.1 Designeinheiten

Bei unserem Vorgehen stellt die Entity eine abstrakte Klasse dar, die, wie in VHDL, die externe Sichtweise des Entwurfs definiert. Die Verwendung einer abstrakten Klasse ist sinnvoll, da auch eine VHDL-Entity kein Verhalten beschreibt und selbst nicht simuliert werden kann. Erst eine konkrete Architecture haucht einer Entity Leben ein. Diese bilden wir auf die Implementierungsklasse ab, in der die ganze Verhaltensbeschreibung des Entwurfs steht.

3.3.2 Datentypen

Da wir in Java arbeiten, können wir natürlich auf die eingebauten Datentypen, wie `int`, `char` etc. zurückgreifen, aber auch alle als Klassen realisierten Datentypen stehen zur Verfügung. Somit steht für den Entwickler sofort ein gewaltiger Fundus an Typen bereit.

VHDL bietet im Gegensatz zu Java aber auch Datentypen an, die im Kontext einer Simulation auf RT-Ebene gebraucht werden. Diese Datentypen sind durch den Wertebereich und die auf ihnen ausführbaren Operationen klassifiziert. Damit wird jedem erfahrenen, objektorientierten Programmierer klar, daß man diese speziellen Datentypen auf eine Klasse abbilden kann. Die Zugriffe auf diese Typen erfolgt über die Mitgliedsfunktionen der Klasse. Die Datenmitglieder der Klassen dienen als Grundlage für den Wertebereich des Typs. Exemplarisch erfolgte eine Implementierung für den Typ `Bitvector`.

Das Klassenkonzept von Java erlaubt es, mit den neuen Typen sehr flexibel umzugehen. So können in Prozeduren zur Deklaration der formalen Parameter die Basisklassen verwendet werden. Bei einem konkreten Aufruf dürfen die aktuellen Parameter dann aus beliebigen, von diesen Basisklassen abgeleiteten Typen bestehen.

3.3.3 Anweisungen

Auch für Anweisungen gilt das zuvor Beschriebene, man kann den kompletten Sprachumfang von Java nutzen. Java bietet eine ausreichende Fülle an Kontrollstrukturen.

3.3.4 Konkurrentes Verhalten mittels Prozessen

Prozesse stellen eine eigenständige Klasse dar. Ihre Implementierung war eine der Hauptaufgaben bei der Bereitstellung von Hilfskonstrukten in Java.

Von den `wait`-Statements wurden drei übernommen, die jetzt in Form von Methoden realisiert sind.

1. `wait()`
2. `waitFor(int)`
3. `waitOn(SignalProg)` bzw. `waitOn(Vector)`

Das Verhalten der migrierten Methoden entspricht dem der VHDL-Statements und wird deshalb nicht noch einmal erklärt. Der `wait until`-Befehl wurde weggelassen, da der gleiche Effekt leicht mit einer bedingten Schleife und der `waitOn()`-Methode erzielt werden kann. Die Funktionsnamen sind aus VHDL übernommen, natürlich bis auf kleine kosmetische Änderungen, um der Syntax von Java gerecht zu werden.

3.3.5 Signale

Manipuliert werden Signale nicht direkt, sondern über bestimmte Methoden. Diese machen auch nur im Kontext eines Prozesses Sinn, da jede Signalzuweisung einen Zeitstempel erhält. Dieser Zeitstempel ist die Simulationszeit des Prozesses mit einer eventuellen Verzögerung. Dies wirft jedoch ein kleines Designproblem auf: sollen die Methoden zur Signalmanipulation in der Prozeß-Klasse oder in der Klasse zur Verwaltung des Signalverlaufs deklariert werden? Die Entscheidung fiel aus verschiedenen Gründen zugunsten der Prozeß-Klasse aus, weswegen leider nicht die übliche, objektorientierte Punktnotation beim Zugriff auf Signale verwendet werden kann.

Die beiden Verzögerungsmechanismen in VHDL haben wir in der Prozeßklasse realisiert, sie sind an dem Suffix der `write`-Methode erkennbar. Es gibt sowohl `writeT`- (Transport) und `writeI`-Methoden (Inertial). Gelesen wird ein Signal mit der `read`-Methode, dabei erhält man immer den Wert aus dem Signalverlauf zur aktuellen Simulationszeit. In Abbildung 2 wäre das zum Zeitpunkt t_0 der Signalwert 2.

3.4 Modellierungsparadigmen

Die zuvor beschriebenen Elemente werden nun in Zusammenhang gesetzt. Es wird beschrieben, wie ein Anwender der Simulator-Hilfsklassen vorzugehen hat, wenn er eine Verhaltensbeschreibung in Java formulieren will. Die angesprochenen Klassen und deren Methoden sind nicht vollständig aufgeführt, da sich Abschnitt 4 explizit damit befaßt.

3.4.1 Definieren der Entity

Die abstrakte Klasse `Entitybase` ist die Basis für die Erstellung einer eigenen Entity, die ebenfalls noch abstrakt bleiben soll. Dies geschieht durch Ableiten von `Entitybase` mit dem Deklarationskopf

```
public abstract class Entity extends Entitybase { }.
```

Der Name `Entity` für diese Klasse ist Konvention und kann notfalls auch anders lauten. Die Klasse `Entitybase` verwaltet die Signale, Inports, Outports sowie Prozesse einer Architecture. Sie stellt zu diesem Zweck Methoden zur Verfügung, die mit `get` beginnen und als Suffix die geforderte Resource haben. Um zwei Outports und einen Inport in der Entity zu definieren ist folgende Befehlsfolge notwendig:

```
Outport out1 = getOutport(new Bitvector());
Outport out2 = getOutport(new Integer());
Inport in1 = getInport(new Bitvector());
```

Der Parameter der Methoden gibt den initialen Wert der Signalverläufe an. Zusätzlich kann man noch die Namen der Resourcen angeben, ansonsten werden für die Namen Defaultwerte eingesetzt. Damit ist die Definition einer Entity im allgemeinen schon abgeschlossen.

3.4.2 Definieren einer Architecture

Die Architecture zur Beschreibung einer Entity ist die Implementierungsklasse derselben. Also wird erneut abgeleitet, aber diesmal nicht mehr abstrakt. Der Name der Architecture ist beliebig und sollte den Zweck deren Verhaltensbeschreibung andeuten.

```
public class Adder extends Entity { }
```

Jetzt sollten die Signale bereitgestellt werden, über die die Prozesse miteinander kommunizieren können. Dies geschieht ganz analog zum Anfordern der Resourcen einer Entity, zur Abwechslung aber ohne Defaultwert für den Namen.

```
Signal hold = getSignal(new Bitvector(4), "hold");
```

Die Basisklasse `Entitybase` deklariert eine abstrakte Funktion `void init()`. Damit muß die erste konkrete, von ihr abgeleitete Klasse diese Methode `void init()` implementieren. In deer Methode werden normalerweise folgende Aufgaben erledigt:

- Setzen der Signalverläufe von Inports
- Integration der verwendeten Prozesse in die Entity
- Tracen von Ports an- und abschalten

Das Setzen von Signalverläufen ist außerhalb von Prozessen nur für Inports erlaubt und erfolgt mit der Methode `setValueAt(int, Object)`, die als Argumente die Simulationszeit und den zugehörigen Signalwert erwartet. Alle Signalverläufe sind mit dem Wert des übergebenen Objekts zum Zeitpunkt 0 voreingestellt. Um also am Inport `in1` einen Peak von $t_0 = 100$ bis $t_1 = 199$ zu erhalten reichen diese zwei Befehle:

```
in1.setValueAt(100, new Bitvector("1"));
in1.setValueAt(200, new Bitvector("0"));
```

Bei Signalen kann mit den beiden Methoden `traceOn()` bzw. `traceOff()` das Tracen an- und abgeschaltet werden. Damit teilt man dem Simulator mit, ob nach Beendigung der Simulation der Signalverlauf ausgegeben werden soll oder nicht. Für unser `hold` Signal reicht dann `hold.traceOn()`.

Essentiell ist, alle verwendeten Prozesse der `Entitybase` mitzuteilen. Dies geschieht ein wenig anders als bei Signalen. Sie werden hier erst dynamisch angelegt und mit der Methode `addProcess(process)` in die Datenstrukturen der `Entitybase` eingebunden. Der einfachste Weg sieht folgendermaßen aus:

```
addProcess(new generateCounter());
```

Hier wird eine Instanz der Klasse `generateCounter` angelegt. Diese muß von der Klasse `process` abgeleitet sein, sonst wirft der Interpreter eine Exception wegen eines illegalen Klassen-Casts.

Die wichtigsten Aufgaben innerhalb der `init()` Methode sind jetzt erläutert, und wir kommen zum wichtigsten Punkt bei der Definition einer Architecture, der eigentlichen Verhaltensbeschreibung in den Prozessen.

Die Prozesse werden in der Architecture selbst also sogenannte *Inner Classes* definiert. Dies ist ein Feature, das Java erst seit dem JDK 1.1 besitzt. Der Nachteil davon ist, daß zur Übersetzung ein aktueller Compiler benötigt wird. Der erzeugte Code läuft aber überall, da keine Änderungen an der Virtual Machine notwendig waren, um Inner Classes zu unterstützen. Der große Vorteil für uns ist, daß wir auf alle Ports und Signale über ihre normalen Identifier zugreifen können.

Hier nochmals schematisch der Aufbau einer Architecture:

```
public class Arch extends Entity {
  // Signale definieren
  // init() Methode implementieren
  public void init() {
    // Inports & Outports festlegen, Signal-Tracing bestimmen ...
  }
  // Prozesse definieren
  class p0 extends process { }
  ...
}
```

3.4.3 Definition von Prozessen

Wie die vorhergehende schematische Übersicht zeigt, werden Prozesse von der abstrakten Klasse `process` abgeleitet. Jetzt kann man alle nötigen Datenstrukturen anlegen, am besten mit einer gleichzeitigen Initialisierung einhergehend.

```
class p1 extends process {
  Bitvector x = new Bitvector("0101");
  int i, j;
  // Rest des Prozesses
}
```

Die Klasse `process` deklariert die abstrakte Methode `void body()`, die also in jedem Prozeß implementiert sein muß. Sie wird solange wiederholt aufgerufen, bis die Simulationszeit des Prozesses über das Simulationsende hinausgeht oder der Prozeß frühzeitig terminiert. In der `body()`-Methode kann beliebiger Javacode stehen. Es ist aber wichtig, einige der von `process` geerbten Methoden einzusetzen. Hinter ihnen verbergen sich die Simulationsprimitive. Würde man beispielsweise eine leere `body()`-Methode implementieren, so geriete der Simulator in eine Endlosschleife, da die Simulationszeit des Prozesses unverändert beim Startwert stehen bliebe. Es werden nun die angesprochenen Primitive genauer erklärt.

3.4.4 Definition des Verhaltens eines Prozesses

Das Verhalten einer VLSI-Schaltung, das wir ja schließlich beschreiben und simulieren wollen, steckt in den `body()`-Methoden der einzelnen Prozesse. Berechnungen, Schleifen und Ähnliches kann der Benutzer wie gewohnt in Java formulieren.

Die Kommunikation zwischen Prozessen erfolgt dann über die schon angesprochenen Signale. Um ein Signal zu lesen dient die Methode `Object read(Port)`. Um z.B. das Signal `hold` einzulesen sind folgende Varianten möglich:

```
x = (Bitvector) read(hold);
x.copy((Bitvector) read(hold));
```

Damit folgen wir den Konventionen zum Gebrauch von Klassen in Java, mehr dazu bei den Implementierungsdetails. Das Schreiben auf Signale präsentiert sich vielgestaltiger, hier gibt es die Ausprägungen `writeT` und `writeI`. `writeT(Outport, Object, int)` ist die ausführliche Version. Dabei ist der erste Parameter der Outport, auf den geschrieben wird und der zweite der Wert, den er annehmen soll. Der dritte Parameter gibt an, um wieviele Zeiteinheiten verzögert (zur aktuellen Simulationszeit) der Wert wirksam werden soll. Gibt man ihn nicht an, so wird ein Defaultwert von 0 verwendet. Der `writeI`-Befehl kann noch einen weiteren Zeitparameter erhalten, das *Rejection Limit*, das hier aber nicht näher erläutert wird. Dazu mehr dann bei der Algorithmenbeschreibung. Damit das Signal `hold` in 12 Zeiteinheiten seinen Wert invertiert ist also diese Codezeile nötig:

```
writeT(hold,((Bitvector) read(hold)).not(), 12);
```

Damit `hold` wirklich einen neuen Wert erhält, benötigt man die `wait`-Methoden. Erst durch sie erfolgt ein Aktivieren der Einträge im Writebuffer, die durch die verschiedenen `write`-Methoden entstanden. Die beiden wichtigsten sind `waitFor(int)` und `waitOn(Vector)`. Im ersten Fall gibt man die Zeiteinheiten an, um die die Clock des Prozesses voranschreiten soll. Die Methode `waitOn` veranlaßt den Prozeß solange zu warten, bis an einem der Signale im übergebenen Vektor eine Signaländerung auftritt. Dann setzt er zu diesem Zeitpunkt die Abarbeitung fort.

3.4.5 Simulator starten

Wenn man ein eigenes Programm geschrieben hat, will man es auch testen. Dazu muß der Simulator gestartet werden. Diese geschieht einfach dadurch, daß in der Architecture die statische Methode `main` überladen wird und eine Instanz der Architecture erzeugt wird. Diese ruft dann noch die von der Basisklasse `Entitybase` geerbte Metheode `simulate` auf. Sie erwartet die Zeit eines Clockzykluses, den Simulationsstartpunkt und das Simulationsende als Argumente.

```
// ----- main program -----
public static void main(String args[]) {
  Adder A = new Adder();
  A.simulate(100, 0,  1000);
}
```

Damit hat der Benutzer der Klassenbibliothek alle Informationen, um eine eigene Verhaltensbeschreibung in Java zu formulieren und zu simulieren, was wir anhand eines kleinen Beispiel demonstrieren.

3.4.6 Ein einfaches Beispiel

Das Beispiel soll das Zusammenwirken der Simulationsprimitive und normalen Javacodes verdeutlichen, es stellt keine sehr sinnvolle Verhaltensbeschreibung dar.

In dem Beispiel wollen wir nur einen Outport, der Bitvektoren als Werte hat, und sonst keine Schnittstelle zur Außenwelt. Damit sieht die Klasse `Entity` so aus:

```
import EDU.UniTuebingen.RT.Entitybase;
import EDU.UniTuebingen.RT.Outport;
import EDU.UniTuebingen.RT.Bitvector;

// Entity has only one Outport
public abstract class Entity extends Entitybase {
  Outport out = getOutport(new Bitvector());
}
```

Unsere Architecture hat zwei interne Signale, das eine ist ein Integerzähler, dessen Startwert auf 1 gesetzt wird. Das andere bekommt nur ein Bit spendiert und dient mehr zu Testzwecken. Dazu kommen dann noch zwei Prozesse, die aber gesondert beschrieben werden. Sowohl die Signale als auch der Outport sollen getraced werden.

```
import java.util.Vector;
import EDU.UniTuebingen.RT.Signal;
import EDU.UniTuebingen.RT.process;
import EDU.UniTuebingen.RT.Bitvector;

public class example extends Entity {
  // define the architecture
  Signal a = getSignal(new Integer(1), "counter");
  Signal b = getSignal(new Bitvector("0110"), "inverter");

  public void init() {
    a.traceOn();
    b.traceOn();
    out.traceOn();
```

```
    // add processes
    addProcess(new p0());
    addProcess(new p1());
  }

  // Prozesse ...

  // ----- main program -----
  public static void main(String args[]) {
    example X = new example();
    X.simulate(100, 0, 1000);
  }
}
```

Die statische `main`-Methode erzeugt eine Instanz des Beispiels und startet es mit einem Clockzyklus von 100 Zeiteinheiten, beendet wird die Simulation zum Zeitpunkt 1000. Jetzt fehlt nur noch der Inhalt der beiden Prozesse `p0()` und `p1()`.

In `p0` geben wir zunächst die Zeit und den aktuellen Wert des Signals `a` aus. Danch wird der interne `int`-Zähler `cnt` mit einer Konstanten multipliziert. Jeden dritten Taktzyklus wird der Zähler dem Signal `a` zugewiesen. Zum Abschluß wird mit `nextCycle()` der Prozeß in den nächsten Taktzyklus geschaltet.

```
// counter process
class p0 extends process {
  int i = 0, cnt = 1;
  public void body() {
    System.out.println("clock " + id + ": " + sim.getClock(id));
    System.out.println("process " + id + " a: " + read(a));
    System.out.println("process " + id + " b: " + read(b));
    cnt *= 121;
    if (i++ == 3) {
      i = 0;
      writeT(a, new Integer(cnt), 57);
    }
    nextCycle(); // or waitFor(100);
  }
}
```

Der zweite Prozeß soll verdeutlichen, wie man mit der Methode `waitOn` umzugehen hat. Als Datenmitglied wird ein Vektor `waitV` angelegt. Um diesen mit Ports zu füllen erstellen wir einen Konstruktor für den Prozeß, in dem dies mit dem Signal `a` geschieht. Im Rumpf des Prozesses wird nur das Signal `b` invertiert und auf den Outport geschrieben. Schließlich wartet man auf Änderungen des Signals `a`, das ja im Vektor `waitV` steckt.

```
// wait on counter process
class p1 extends process {
  Vector waitV = new Vector();
  p1() { waitV.addElement(a); }
  public void body() {
    writeT(b, ((Bitvector) read(b)).not(), 50);
    writeT(out, read(b));
    waitOn(waitV);
  }
}
```

Hier kurz ein Auszug aus der von Prozeß `p0` erzeugten Ausgabe:

```
clock 0: 300.0
process 0 a: 1
process 0 b: 0110
clock 0: 400.0
process 0 a: 214358881
process 0 b: 0110
clock 0: 500.0
process 0 a: 214358881
process 0 b: 1001
```

Der Simulator erzeugt für die angegebenen Signale folgendes Trace-Protokoll:

```
counter:
0.0 - 1
357.0 - 214358881
757.0 - 772479681
inverter:
0.0 - 0110
50.0 - 1001
407.0 - 0110
807.0 - 1001
Outport 1:
0.0 - 0
0.01 - 0110
357.01 - 1001
757.01 - 0110
```

Man sieht zum Beispiel, daß beim Schreiben auf den Outport ein Delta Step erfolgt, da die Verzögerung den Defaultwert 0 erhält.

3.5 Simulationskonzept

In den vorherigen Abschnitten wurde beschrieben, wie eine Verhaltensbeschreibung auszusehen hat und welches die wichtigsten Methoden und Primitive sind. Es war eine informelle Einführung in die Benutzung der Java-Klassen, ohne vorher ein breites Hintergrundwissen über die verwendeten Konzepte und Algorithmen zu liefern. Das wird nun nachgeholt. Zuerst wird das Simulationsverfahren und das Timingmodell beschrieben und anschließend die Algorithmen, die den schon angesprochenen Simulationsprimitiven zugrundeliegen.

3.5.1 Simulationsverfahren

Das Zusammenspiel unserer Simulatorklasse, der Prozesse und Signale läßt sich unter den ereignisorientierten Simulationsverfahren einordnen. Es gibt keine festen Zeitschritte, an denen die Ausgänge der Komponenten berechnet werden. Erst wenn es zu einer Interaktion zwischen Prozessen über Signale kommt ist ein Abgleichen der Zeiten der beteiligten Prozesse notwendig. Die Konsequenz aus diesen Voraussetzungen ist, daß jeder Prozeß eine eigene Clock erhält. Da die Prozesse eine eigene Clock haben, handelt es sich bei unserem Simulator außerdem um eine asynchrone Ereignissimulation. Die Ereignisse, also Signaländerungen an internen und externen Signalen, können zu beliebigen Zeitpunkten auftreten und von unterschiedlichen Gatterverzögerungszeiten abhängig sein. Dafür ist eine eigene Datenstruktur nötig, die die Zeitpunkte der Ereignisse und ihre Wertänderungen speichert. Wir nennen sie *Writebuffer*, da es zu Signaländerungen nur durch Befehle zum Schreiben auf Signale kommen kann. Im Writebuffer steht der Zeitpunkt des Ereignisses, das betroffene Signal und der erwünschte, neue Wert des Signals. Auch der Writebuffer ist ein prozeßeigenes Datenmitglied. Jeder Prozeß schreibt zuerst alle auftretenden Signaländerungen in diesen Zwischenspeicher. Erst die demnächst beschriebenen Algorithmen zur Abarbeitung des Writebuffers entscheiden über die tatsächlichen Signaländerungen mit globalen Auswirkungen.

3.5.2 Timingmodell

Marwedel erläutert in [5] die gebräuchlichen Timingmodelle bei der VLSI Simulation. Wir haben drei davon implementiert, die auch schon kurz angesprochen wurden. Dabei handelt es sich um:

1. Transportverzögerung
2. Trägheit (Intertial Delay)
3. Delta-Delay

Die ersten beiden wurden schon im Zusammenhang mit VHDL-Signalen erklärt. Zu sogenannten Delta-Delays kommt es, wenn eine Signaländerung aktiviert wird, aber keine reale Simulationszeit verstreicht. Ein kurzes Beispiel soll dies verdeutlichen:

```
Signal s = getSignal(new Bitvector(), "test");

process p0 extends process {
  public void body() {
    writeT(s, new Bitvector("1111"));
    waitFor(0);
    writeT(s, new Bitvector("0011"));
    waitFor(0);
    writeT(s, new Bitvector("0101"));
    waitFor(10);
  }
}
```

Die erste Signalzuweisung hat keine Verzögerung und erfolgt also zur aktuellen Simulationszeit. Die beiden anderen ebenso, da nach dem `waitFor(0);` keine Simulationszeit verstrichen ist. Damit erhält das Signal `s` den Wert "1111" mit dem Delta-Delay $\Delta = 1$, den Wert "0011" mit dem Delta-Delay $\Delta = 2$ und den Wert "0101" mit dem Delta-Delay $\Delta = 3$. Ein Auszug aus dem Signaltrace von Signal `s` kann so aussehen:

```
710.01 - 1111
710.02 - 0011
710.03 - 0101
720.01 - 1111
720.02 - 0011
720.03 - 0101
730.01 - 1111
730.02 - 0011
730.03 - 0101
```

Ein Beispiel für die Anwendung von Delta-Delays wäre die Simulation eines Prozessors, dessen Instruktionen in die üblichen Einheiten Fetch, Decode und Memory Access zerlegt werden. Dabei wird die Instruktion während eines Zeitpunktes abgearbeitet, aber die einzelnen Instruktionseinheiten sind mit verschiedenen Delta-Delays versehen.

3.6 Algorithmen

Die Algorithmen lassen sich grob in drei Bereiche unterteilen: Signale lesen, Signale schreiben und Timingverhalten. Die Algorithmen haben Entsprechungen in Java-Methoden, die innerhalb eines Prozesses aufgerufen werden können.

Um die Notation zu vereinfachen, vereinbaren wir folgende Konventionen innerhalb der Algorithmenbeschreibungen:

- i – Nummer des Prozesses, in dem die Methode aufgerufen wurde
- t_j – aktuelle Zeit des Prozesses j
- w_j – Writebuffer des Prozesses j

3.6.1 Signale lesen

Hierbei handelt es sich um den einfachsten Fall und er läßt sich kurz beschreiben. In einem Prozeß tritt ein `read`-Befehl auf und gesucht ist der Wert des Signals zum aktuellen Zeitpunkt des Prozesses. Dazu wird folgender Algorithmus abgearbeitet:

Algorithmus: Signal lesen

Eingabe: Signal `s`
Ausgabe: Wert von `s` zum Zeitpunkt t_i

Schritt (1) - Fehlerfall
falls `s` ein Outport : Fehler; sonst weiter mit (2)

Schritt (2) - Sonderfall Inport
falls `s` ein Inport: ersten Wert aus dem Signalverlauf des Inports bestimmen, dessen Zeitstempel keiner oder gleich t_i ist, fertig; sonst weiter mit (3)

Schritt (3) - Verjüngen
warten, bis es keinen jüngeren Prozeß mehr gibt, der zu einer Signaländerung führen kann, somit scheiden alle beendeten und auf Signaländerungen wartende Prozesse aus:

$$t_i >= t_j \ \wedge \ isAlive(t_j) \wedge !isWaitChange(t_j) \ \forall \ j \neq i$$

weiter mit (4)

Schritt (4) - Wert bestimmen
ersten Wert aus dem Signalverlauf von `s` bestimmen, dessen Zeitstempel kleiner oder gleich t_i ist, fertig.

Schritt (3) ist nur bei internen Signalen nötig, da diese während der Laufzeit der Simulation von Prozessen geändert werden können. Momentan kann zwar nur ein Prozeß ein Signal ändern, um aber spätere Erweiterungen zu vereinfachen ist diese allgemeine Form des Algorithmuses gewählt worden. Die boolschen Funktionen $isAlive$ und $isWaitChange$ testen, ob der Prozeß schon beendet ist, bzw. ob er gerade auf eine Signaländerung wartet.

3.6.2 Signale schreiben

Die `write`-Befehle wirken sich zunächst nur auf den Writebuffer des Prozesses i aus. Erst nach dem Auftreten einer der `wait`-Methoden haben sie globale Auswirkungen auf die Signalverläufe. Hier wird nur beschrieben, welche Änderungen der Writebuffer w_i erfährt, die Wirkungsweise der `waits` folgt danach.

Algorithmus: Signal schreiben

Eingabe: Signal `s`, Wert `val`; optional: Verzögerungszeit t_d, Rejection Limit t_l
Ausgabe: keine

Schritt (1) - Zugriffszeit
Zeitpunkt t_w des Schreibzugriffs bestimmen: $t_w := floor(t_i) + t_d$

Schritt (2) - Einträge invalidieren
alle Einträge im Writebuffer löschen, die einen Zeitstempel tragen, der größer oder gleich t_w ist

Schritt (3) - Transaktion
neue Transaktion in Writebuffer eintragen: `s` mit Zeitstempel t_w und Wert `val`; handelt es sich um Transportverzögerung ($t_l = 0$): fertig, sonst weiter mit (4)

Schritt (4) - Interial Delay
alle Einträge im Writebuffer löschen, die einen Zeitstempel tragen, der im Intervall $[t_w - t_l, t_w[$ liegt und entweder einen Wert ungleich `val` haben oder nicht im Intervall $]t_{max}, t_w[$ liegen, wobei t_{max} der größte Zeitstempel eines Eintrages ist, der einen Wert ungleich `val` hat; fertig.

`write`-Befehle gibt es in zwei Ausprägungen, einerseits für Transportverzögerung, andererseits für Intertial Delays. Alle erwarten den neuen Signalwert `val` und das Signal `s` als Eingabe. Ohne weitere Angaben wird dann die Verzögerungszeit $t_d := 0$ gesetzt, d.h. die Signaländerung soll zum Zeitpunkt t_i erfolgen. Ist dies nicht erwünscht, muß die Verzögerungszeit explizit angegeben werden. `writeT`-Befehle können keine weiteren Eingaben erhalten und setzen $t_l := 0$. Im Trägheitsmodell kann noch ein Rejection Limit angegeben werden. Dieses ist normalerweise gleich der Verzögerungszeit, kann aber auch kleiner gewählt werden.

Der neue Eintrag invalidiert in Schritt (2) nun alle Einträge, die zeitlich hinter ihm liegen, d.h. einen größeren Zeitstempel tragen und wird dann in (3) in w_i eingetragen. Bei Transportverzögerung ist man nun fertig und kann den Algorithmus verlassen.

Bei Intertial Delay kommt jetzt der entscheidende Schritt. Zuerst werden alle Einträge gelöscht, die innerhalb des Ablehnungsintervalls $[t_w - t_l, t_w[$ liegen und einen Wert ungleich `val` haben. Dabei merkt man sich den Zeitpunkt t_{max}, der dem größten Zeitstempel der soeben gelöschten Einträge entspricht. Danach löscht man auch noch die verbliebenen Einträge im Intervall $]t_w - t_l, t_{max}[$. Damit verhindert man, daß Einträge gelöscht werden, die direkt vor t_w liegen und den gleichen Wert haben. Für sie stellt der neue Eintrag keine Signaländerung dar.

Nun kommen wir zur Beschreibung der Algorithmen, die die Einträge im Writebuffer aktivieren.

3.6.3 Timingverhalten

Die letzte Gruppe von Algorithmen befäßt mit den Simulationsprimitiven, die dazu dienen, das Timingverhalten innerhalb eines Prozesses zu beschreiben.

Der einfachste Algorithmus ist mit dem Befehl `waitEnd()` verbunden. Alles, was er macht, ist den Writebuffer an den Simulator zu schicken, damit dieser die Signalverläufe entsprechend ändert. Anschließend wird der Prozeß beendet. Wir nehmen das aber zum Anlaß, den Algorithmus zu erläutern, der zum Schreiben des Writebuffers verwendet wird.

Anmerkung: Die Algorithmen sind entweder in der Klasse `process` oder `Simulator` implementiert. Alle Algorithmen in der letzteren benötigen aus technischen Gründen die Nummer des Prozesses, von dem aus sie aufgerufen wurden. Diese ID wird aber bei der momentanen Algorithmenbeschreibung nicht aufgeführt, da sie per Konvention schon durch die Variable i symbolisiert wird.

Algorithmus: Writebuffer schreiben

Eingabe: Writebuffer `w`
Ausgabe: keine

Schritt (1)
Eintrag e aus den Buffer `w` entnehmen und testen, ob sein Zeitstempel gleich t_i ist, wenn ja: weiter mit (2), ansonsten mit (3); enthält der Writebuffer keine weiteren Einträge weiter mit (4)

Schritt (2) - Transaktion mit Delta Step
die in Eintrag e codierte Transaktion in den Signalverlauf des betreffenden Signals eintragen, aber als Zeitstempel t_s den der Transaktion, um einen Konstante Δ_c, erhöht verwenden, also $t_s := t_i + \Delta_c$; weiter mit (1)

Schritt (3) - Transaktion
die in Eintrag e codierte Transaktion in den Signalverlauf des betreffenden Signals eintragen; weiter mit (1)

Schritt (4)
falls ein Transaktion mit Delta Step erfolgte, Clock des Prozesses um Δ_c erhöhen, also $t_i := t_i + \Delta_c$; allen wartenden Prozesse mitteilen, daß Signaländerungen erfolgten; fertig.

Die Methode `writeWBuffer`, hinter der dieser Algorithmus steckt, wird von allen `wait`-Methoden verwendet. Dabei erfolgt aber meist eine Vorauswahl aus dem prozeßeigenen Writebuffer.

Die `WaitFor(int t)`-Methode wird dazu verwendet, die Prozeßzeit um den Wert `t` zu erhöhen. Dabei müssen ein Teil der Einträge im Writebuffer w_i aktiviert werden. Hier nun der Algorithmus:

Algorithmus: Zeiteinheiten warten

Eingabe: Zeiteinheiten t
Ausgabe: keine

Schritt (1)- Fehlerfall
falls $t < 0$: Fehler; ansonsten weiter mit (2)

Schritt (2) - Transaktionen starten
alle Einträge im Writebuffer w_i aktivieren, d.h. als Eingabe an den Algorithmus *Writebuffer schreiben* weiterreichen, deren Zeitstempel innerhalb des Intervalls $[t_i, floor(t_i + t)]$ liegen; diese Einträge anschließend aus w_i entfernen; weiter mit (3)

Schritt (3)
falls $t > 0$: Clock des Prozesses i erhöhen, also $t_i := floor(t_i + t)$; fertig.

Schritt (2) ist der Kern des Algorithmuses. Schriebe man den ganzen Writebuffer w_i, so würden eventuell Einträge aktiviert, bei denen dies zu fehlerhaften Signalverläufen führen würde. Ein kleines Beispiel soll solch einen Fall aufzeigen.

```
void body() {
  writeT(s, new Bitvector("1"), 2);
  writeT(s, new Bitvector("0"), 15);
  waitFor(10);
}
```

Ginge man naiv vor und schriebe w_i komplett, so würde auch der zweite `writeT`-Befehl in dem Signalverlauf von Signal `s` auftauchen, der Signalverlauf sähe so aus:

Zeitpunkt	2	12	15	22	25	32	...
Wert	1	1	0	1	0	1	...

Dabei wäre nach dem Algorithmus *Signal schreiben* ein kontinuierliches 1-Signal richtig. Der erste `writeT`-Befehl deaktiviert immer den für einen späteren Zeitpunkt eingetragen zweiten.

Um diesen Fehlerfall zu vermeiden, aktiviert man nur die Einträge, die im Intervall $[t_i, floor(t_i+t)]$ liegen und beläßt die übrigen in w_i. Dann wird auch der Eintrag durch den zweiten `writeT`-Aufruf immer korrekt invalidiert.

In Schritt (3) erhöht man die Clock nur für den Fall, daß $t > 0$. Ansonsten würden aufgetretene Delta Steps überschrieben, die schon im Algorithmus *Writebuffer schreiben* die Clock änderten.

Eine Variation des Algorithmus *Zeiteinheiten warten* stellt die Methode `nextCycle` dar. Hier wird nicht eine vom Programmierer festgelegte Anzahl von Zeiteinheiten gewartet. Vielmehr bestimmt der Simulator, wieviele Zeiteinheiten t_{diff} von der aktuellen Prozeßzeit t_i bis zum nächsten Clockzyklus noch fehlen. Mit t_{diff} wird dann einfach `waitFor(`t_{diff}`)` aufgerufen.

Schließlich kommen wir noch zur aufwendigsten Methode dieser Kategorie, der `waitOn` Methode. Sie gibt es in zwei Arten, erstens mit einem Signal und zweitens mit einem Vektor von Signalen als Eingabe. Der erste Fall wird einfach auf den zweiten projiziert, den wir deshalb ausschließlich behandeln.

Was dabei passiert ist schnell gesagt. Wenn man die Methode `waitOn(Vector v)` aufruft, wartet diese solange, bis sich an einem der im Vektor befindlichen Signalen eine Wertänderung ergibt. Dann wird noch die Clock auf den Zeitpunkt der Änderung gesetzt.

Algorithmus: Auf Signaländerung warten

Eingabe: Vektor `v` von Signalen
Ausgabe: keine

Schritt (1) - Fehlerfall
falls eines der Signale in `v` ein Outport ist: Fehler; sonst weiter mit (2)

Schritt (2)
die Werte aller Signale in `v` zum Zeitpunkt t_i bestimmen und in Liste `vals` speichern; weiter mit (3)

Schritt (3) - aktuelle Transaktionen starten
alle Einträge im Writebuffer w_i aktivieren, deren Zeitstempel gleich t_i ist; diese Einträge anschließend aus w_i entfernen; weiter mit (4)

Schritt (4) - obere Schranke bestimmen
alle Signale in `v`, die entweder Inports oder im Writebuffer w_i sind, aus `v` entfernen und dabei die Zeit t_{lim} bestimmen, wobei t_{lim} der früheste Zeitpunkt ist, an dem an einem der soeben entfernten Signale eine Wertänderung erfolgte; falls keine Änderungen auftraten: $t_{lim} := -1$; weiter mit (5)

Schritt (5) - endlos warten
falls `v` jetzt leer ist und $t_{lim} = -1$ (unbestimmt): w_i aktivieren und Prozeß beenden; sonst weiter mit (6)

Schritt (6) - obere Schranke setzen
falls $t_{lim} = -1$: $t_{lim} := sim_{end} + 1$ und weiter mit (7)

Schritt (7) - Signaländerungen testen
alle Signale in `v` untersuchen und falls Änderung zum Zeitpunkt $t_c \in [t_i, t_{lim}[$ auftritt: $t_{lim} := t_c$ sowie Signal aus `v` entfernen; weiter mit (8)

Schritt (8) - Schleife
falls es keinen aktiven Prozeß gibt, der jünger als t_{lim} ist: weiter mit (9), sonst warten bis Signaländerung auftritt und weiter mit (7).

Schritt (9) - Reduktion auf Algorithmus *Zeiteinheiten warten*
falls noch $t_{lim} = sim_{end} + 1$: w_i aktivieren und Prozeß beenden, sonst Zeitdifferenz $\Delta_t = floor(t_{lim}) - floor(t_i)$ berechnen und Algorithmus *Zeiteinheiten warten* mit Δ_t aufrufen; fertig.

Schritt (3) soll verhindern, das Signaländerungen innerhalb eines Delta Steps übersehen werden. Diese müssen auch erkannt werden, sonst droht ein Deadlock.

In (4) untersucht man alle Signale in `v`, die entweder Inports oder im eigenen Writebuffer w_i enthalten sind. Diese Signale können nicht durch andere Prozesse manipuliert werden. Falls in einem dieser Signale eine Wertänderung eingetragen ist, erhält man eine zeitliche obere Schranke für das darauffolgende Warten. Denn alle Prozesse, die den

Zeitpunkt passieren, können keine relevante Signaländerung mehr herbeiführen. Ansonsten muß im schlimmsten Fall die obere Schranke hinter das Ende der Simulation gesetzt werden, was in Schritt (6) geschieht.

In der Schleife, die durch die Schritte (7) und (8) gegeben ist, geschieht dann das tatsächliche Warten auf Signaländerung. Die Abbruchbedingung für die Schleife ist entweder ein leerer Vektor `v`, oder daß alle noch aktiven Prozesse über der derzeitigen oberen Zeitschranke liegen.

Hat man den Zeitpunkt der Änderung nun wirklich gefunden, so läßt sich der Rest der Arbeit mit dem vorherigen Algorithmus *Zeiteinheiten warten* erledigen.

Damit sind die wichtigsten und aufwendigsten Algorithmen erläutert, die hinter den Simulationsprimitiven stecken. Im nächsten Abschnitt beschreiben wir die Realisierung dieser Algorithmen in Java.

4 Implementierung

Die Datenstrukturen, auf denen die Algorithmen basieren, sind teilweise recht komplex. Java selbst bietet mit der Klasse `Vector` im `util` Package ein dynamisches Array, aus dem Objekte beliebig entfernt, eingefügt und manipuliert werden können. Von ihm ihr wird auch rege Gebrauch gemacht, um Daten linear zu speichern. Aber oft ist eine assoziative Datenstruktur hilfreich, z.B. stehen bei Signalen Zeitpunkt und Wert in einem solchen Zusammmenhang. Hier hat sich in den letzten Jahren *generisches Programmieren* als Alternative herausgebildet. Das wohl bekannteste Ergebnis dieses Forschungsbereiches ist die *Standard Template Library* (STL), die in die C++ Standard Bibliothek aufgenommen wurde. Eine Einführung liefert [6]. Object Space hat die STL nach Java portiert und frei zugänglich gemacht. Da sich unsere Implementierung kräftig bei der JGL ([7]) bedient, folgt nun eine kurze Einführung in deren Konzepte.

4.1 Die Java Generic Library - JGL

Die JGL stützt sich ganz grundlegend auf vier Konzepte:

- Container
- Algorithmen
- Iteratoren
- Funktionsobjekte

Container sind die Datenstrukturen, in denen die Daten gespeichert sind. Für Java typisch werden diese als `Object` gehalten. Dies ist die objektorientierte Art Generizität auszudrücken. Der Nachteil dieser Methode ist die nicht vorhandene Typsicherheit zur Compilezeit. Container lassen sich noch weiter unterteilen in: Sequences (Arrays, Listen und Dequeues), Sets, Maps, Queue und Stack.

Um auf den Daten zu operieren stehen eine Vielzahl von Algorithmen zur Verfügung. Mit ihnen kann man sortieren, kopieren, suchen, löschen und ähnliche Operationen auf den Containern ausführen. Dabei handelt es sich meist um Algorithmen, die beim alltäglichen Arbeiten mit Datenstrukturen immer wieder auftauchen. Die JGL ist in dem Sinne generisch, daß sich fast alle Algorithmen auf die meisten Container anwenden lassen.

Damit dies reibungslos geschehen kann braucht man Iteratoren und Funktionsobjekte. Iteratoren sind abstrahierte Zeiger auf Objekte in den Containern. Viele Algorithmen erwarten Iteratoren als Eingabeparameter. Sie arbeiten dann auf dem Bereich, der durch die Iteratoren aufgespannt wird. Das kann natürlich auch der ganze Container sein. So liefert die Methode `end()` einen Iterator auf das Ende eines Containers zurück, analoges gilt für die Methode `start()`. Funktionsobjekte werden gebraucht, um spezialisierte Vergleiche und ähnliches auf den Objekten ausführen zu können. Es soll möglich sein, Zahlen auf- und absteigend oder Zeichenkeiten nach verschiedenen Ordungen zu sortieren. Dann bedient man sich eines Funktionsobjekts, in dem der gewünschte Vergleich implementiert ist.

Ein kleines Beispiel soll die Funktionsweise der JGL demonstrieren. Dazu verwenden wir den Datentyp `OrderedMap`, der ein assoziativer Container ist. Er speichert Daten als Paar von Schlüssel und Datum ab, wie z.B. Telefonnummern und der zugehörige Name.

```
import COM.objectspace.jgl.OrderedMap;
import COM.objectspace.jgl.OrderedMapIterator;

public class test {
  // ----- main program -----
  public static void main(String args[]) {
    OrderedMap o = new OrderedMap();
    o.add(new Integer(915181), new String("Volker"));
    o.add(new Integer(234734), new String("Anette"));
    o.add(new Integer(287471), new String("Markus"));
    o.add(new Integer(360096), new String("Elke"));

    OrderedMapIterator oi = new OrderedMapIterator(o.begin());
    while(!oi.atEnd()) {
      System.out.print(oi.key() + " - ");
      System.out.println(oi.value());
      oi.advance();
    }
  }
}
```

Zuerst wird eine `OrderedMap` angelegt und mit der Standardmethode `add` für Container werden einige Nummern und Namen eingefügt. Dabei werden die Daten automatisch sortiert, wie der Name impliziert. Anschließend holt man sich einen Iterator auf den Anfang von `o` und gibt in einer Schleife die Datenpaare aus.

Die Ausgabe sieht folgendermaßen aus:

```
234734 - Anette
287471 - Markus
360096 - Elke
915181 - Volker
```

4.2 Packagename

Alle Klassen, die zu unserem RT-Simulator gehören, stecken wir in einen eigenen Namensraum, was in Java über das Packageprinzip realisiert ist. Damit vermeiden wir Namenskonflikte und deuten auf den Inhalt der Bibliothek hin. Als Packagenamen verwenden wir:

`EDU.UniTuebingen.RT`

4.3 Implementierung von Signalen

Abbildung 3 zeigt die Klassenhierarchie von Signalen. Wichtig daran ist, daß die abstrakte Klasse `SignalProg` Ausgangspunkt aller verfügbaren Signaltypen ist. In ihr sind alle Datenstrukturen enthalten, sowie die wichtigsten Methoden.

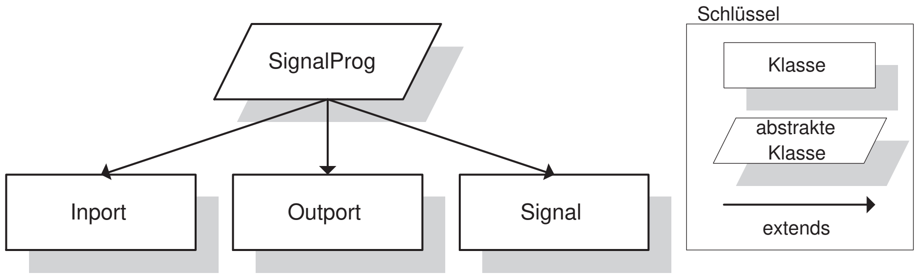

Abbildung 3: Klassenhierarchie von Signalen

4.3.1 Datenstrukturen

Ein Signal ist durch vier Datenstrukturen charakterisiert: seinen Namen, seinen Signalverlauf, ein Flag für eventuelles Tracen und den Prozeß, der darauf zugreifen darf.

```
private String name;
private DList signalProg;
private boolean traced;
private process writePro;
```

Die einzige Struktur, die mehr Intresse verdient, ist der Signalverlauf `signalProg`. Dabei handelt es sich um einen JGL-Container des Typs `DList`, eine doppelt verkettete Liste. Einträge in diese Liste erfolgen über die kleine Hilfsklasse `SPEntry`. Sie enthält Datenmitglieder für den Zeitstempel und den zugehörigen Signalwert, sowie Methoden um diese Werte auszulesen:

```
public class SPEntry {
  private double time;
  private Object value;

  SPEntry(double t, Object val) {
    time = t;
    value = val;
  }
  public double time() { return time; }
  public Object value() { return value; }
}
```

Wir wählen, um den Wert des Signals zu speichern, den allgemeinem Typ `Object`, damit der Typ von Signalen beliebig sein kann. In einem ersten Ansatz hatten wir `SignalProg` von `Bitvector` abgeleitet, was aber die Erweiterbarkeit stark einschränkte.

Auch bekommt man so gleich die ganzen Typen von Java „geschenkt“ und kann sie in Signalen benutzen.

4.3.2 Methoden

Der Konstruktor setzt die obigen Daten und legt eine neue `DList` an. Die beiden entscheidenden Methoden bei Signalverläufen sind:

```
Object getValueAt(double t)
void setValueAt(double at, Object val, process ap)
```

Damit holt bzw. setzt man den Wert eines Signals zum angegebenen Zeitpunkt. Da als Container eine `DList` verwendet wird, muß das sortierte Einfügen von Hand erfolgen. Hierzu ist in der privaten Methode `minPos(t)` eine binäre Suche implementiert, die einen Iterator auf den kleinsten Eintrag im Signalverlauf zurückliefert, dessen Zeitstempel $\leq$ `t` ist. Das Einfügen dann kann mit der Methode `insert(Iterator, Object)` der `DList` erfolgen.

```
signalProg.insert(oi, new SPEntry(at, val));
```

Das Vorgehen zum Auslesen des Wertes zum Zeitpunkt `at` ist analog. Die Klasse `SignalProg` liefert auch noch Methoden, um den Namen zu bestimmen, das Traceflag zu ändern und die nächste Änderung von einem Zeitpunkt aus zu bestimmen. Alle Methoden und Daten sind in der Onlinedokumentation vollständig verzeichnet.

Output- und Input-Ports benötigen eine Sonderbehandlung. Bei Output-Ports wird die Methode `getValueAt` überschrieben, um zu vermeiden, daß während der Simulationszeit von einem Outport gelesen wird. Analog geschieht dies für Input-Ports, aber diesmal für die Methode `setValueAt`.

4.4 Implementierung von Datentypen: Beispiel `Bitvector`

Wie schon erwähnt können in Signalen alle Klassentypen als Werte benutzt werden. Aber bei VLSI-Simulationen benutzt man gerne Typen, die hardwarenahe Komponenten simulieren. Dazu muß ein Entwickler einfach eine eigene Klasse schreiben. Wir wollen am Beispiel der Klasse `Bitvector` zeigen, was dabei beachtet werden muß.

4.4.1 Datenstrukturen

Wir repräsentieren eine Bitvektor durch seine Größe und ein Array von `boolean`s. Natürlich sind die Datenstrukturen stark vom zu implementierenden Typ abhängig.

```
protected boolean [] myValue;
protected int mySize;
```

4.4.2 Methoden

Zwei Methoden sind zum sinnvollen Arbeiten mit einem eigenen Datentyp sehr wichtig. Es handelt sich dabei um Methoden, die auch schon in der Basisklasse `Object` definiert sind. Die erste wird benutzt, um ein Objekt auszugeben, z.B. mit der Methode `System.out.print`. Dazu ist das Überladen von `public String toString()` nötig. In der Klasse `Bitvector` laufen wir dabei über das Array von `boolean`s und ersetzen `true` durch `"1"` sowie `false` durch `"0"`.

Noch wichtiger ist die Methode `public boolean equals(Object o)`. Sie wird von der `Port`-Klasse benutzt, um die Werte von Signalen zu vergleichen. Erst wenn diese entsprechend implementiert ist, kann man ein fehlerfreies Funktionieren der Klasse `Port` garantieren. Ein `Bitvector` liefert `true` zurück, wenn erstens die Länge gleich ist und zweitens alle Werte in `myValue` übereinstimmen:

```
public boolean equals(Object o) {
  if  (mySize == ((Bitvector) o).mySize) {
    for (int i = 0; i < mySize; i++) {
      if (myValue[i] != ((Bitvector) o).myValue[i]) return false;
    }
    return true;
  }
  return false;
}
```

Zu diesen unerläßlichen Methoden sollten dann noch einige kommen, die das Arbeiten mit der neuen Klasse erleichtern. Dazu gehören Konstruktoren und Konversionsroutinen, in unserem Beispiel `valueOf(int)`, `valueOf(String)`, `toInt()`, aber auch eine bessere Vergleichsmethode wie `int compareTo(Bitvector B)`.

4.5 Die `Entiybase`

In Abschnitt 3 wurde beschrieben, daß die `Entiybase` den Ausgangspunkt einer eigenen Architecture bildet. Diese Klasse stellt Methoden zur Verfügung, um die Ressourcen, Signale und Prozesse, anzufordern und den Simulator zu starten.

4.5.1 Datenstrukturen

Die Ressourcen werden alle in Vektoren gespeichert, da diese dynamisch ihre Größe anpassen. Jeder Typ bekommt einen eigenen Vektor spendiert, um ein einfaches Unterscheiden zu ermöglichen.

```
protected Vector Inports = new Vector();
protected Vector Outports = new Vector();
protected Vector Signals = new Vector();
protected Vector Processes = new Vector();
```

4.5.2 Methoden

Die `get`-Methoden wurden prinzipiell auch schon erläutert. Sie erzeugen eine Instanz der angeforderten Resource und geben ihr, falls nicht vorhanden, eine Defaultnamen. Dann wird die Referenz der Ressource in den Vektor eingefügt und der Benutzer kriegt ebenfalls eine Referenz auf die Instanz zurück. Der Code für einen Input-Port ist hier exemplarisch abgedruckt.

```
protected Inport getInport(Object o, String s) {
  Inport p = new Inport(o, s);
  Inports.addElement(p);
  return p;
}
```

4.6 Der Simulator als Client-Server-Modell

Die Aufgabe des Simulators besteht vordergründig darin, die Prozesse einer Architecture zu starten und auf deren Ende zu warten. Damit ist es aber nicht getan. Konzeptuell stellt der Simulator einen Server dar, die Prozesse sind seine Klienten.

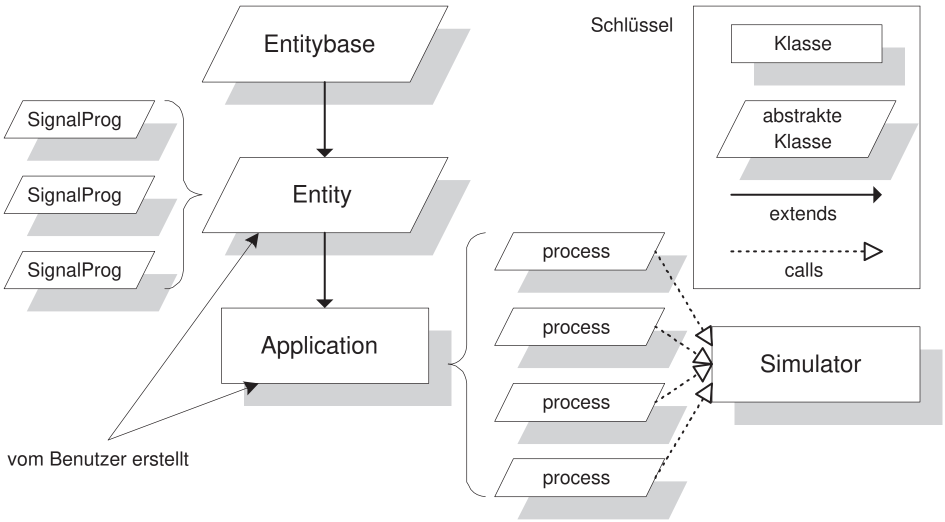

Abbildung 4: Struktur einer Simulation

Abbildung 4 zeigt den Aufbau einer Simulation. Man sieht aber auch deutlich, in welcher Beziehung Prozesse und Simulator stehen. Es gibt eine Vielzahl von Prozessen, aber nur eine Instanz des Simulators. Wenn Prozesse ihren Zustand ändern wollen, wie z.B. die Uhrzeit setzen oder auf ein Signal schreiben, so müsen sie ein Request an den Server, also den Simulator, senden. Auf diese Weise können die Prozesse auch synchronisiert werden. Ruft ein Prozeß eine Methode des Simulators auf, so tritt er in dessen Monitor ein. Hat ein anderer Prozeß diesen schon an sich genommen, muß der Prozeß warten. In Abbildung 5 sieht man, daß es nur einen Weg durch den Simulator gibt. Befindet sich dieser schon in einer als `synchronized` deklarierten Methode, so müssen alle anderen Prozesse warten.

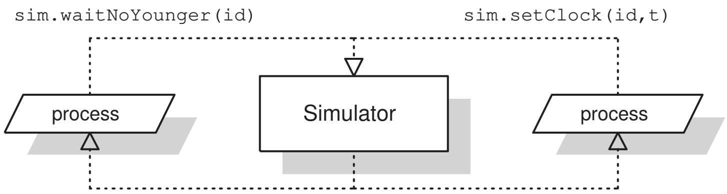

Abbildung 5: Client-Server Verhältnis von Prozessen und Simulator

Andererseits garantiert dieses Client-Server-Modell die Konsiztenz der Daten, da nur der Simulator Daten von Prozessen und Signalen direkt ändern kann. Die beiden Klassen `process` und `Simulator` sind somit auch die aufwendigsten.

4.7 Implementierung von Prozessen

Prozesse dienen dazu, konkurrentes Verhalten zu simulieren. Für jeden erfahrenen Java-Programmierer drängt sich sogleich die Klasse `Thread` auf, die eines der interessantesten Features von Java darstellt. Threads werden benutzt, um Teile einer Applikation unäbhängig zu machen. Damit erreicht man schnellere Response-Zeiten und einen besseren Durchsatz. Letzteres ist für uns von Bedeutung. Die Abbildung von VHDL-Prozessen auf Java-Threads war eine logische Folgerung. Threads und ihre Anwendung sind in [1] detailiert erläutert.

```
public abstract class process extends Thread {}
```

Wir wählten den Namen `process` entgegen der üblichen Konventionen, um Konflikte mit der Klasse `Process` aus dem `java.lang` Package zu vermeiden.

4.7.1 Datenstrukturen

Die Prozesse brauchen eine Zugriffsmöglichkeit auf den Server. In einer objektorientierten Sprache reicht dazu eine Referenz auf dessen Instanz, zumindest wenn der Server ein lokales Objekt ist. Wir haben `sim` als `static` deklariert, um inkonsistente Referenzen der einzelnen Prozesse auf den Simulator unmöglich zu machen.

```
static protected Simulator sim;
protected int id;
private DList writeBuffer = new DList();
```

Das Datum `id` wird am Simulationsanfang vom Simulator gesetzt, damit er die einzelnen Prozesse unterscheiden kann. Und zu guter Letzt hat jeder Prozeß seinen Writebuffer, der schon in dem Abschnitt über Algorithmen Thema war.

Im Writebuffer sind pro Eintrag drei Daten gespeichert: der Zeitpunkt des Schreibzugriffs, das betroffene Signal und der neue Wert des Signals. Als Java-Datenstruktur wählten wir wieder den JGL-Container `DList`. Da verschiedene Signale zum gleichen Zeitpunkt geschrieben werden können, erhält ein Eintrag in den Writebuffer noch das betroffene Signal. Die Klasse `WBEntry` kapselt Einträge in den Writebuffer, sie ist von `SPEntry` abgeleitet. Der Wert ist damit, wie in der `SignalProg`-Klasse, einfach ein `Object`. Das Erstellen eines Eintrags sieht so aus:

```
writeBuffer.pushBack(new WBEntry(newTime, val, p));
```

4.7.2 Methoden

Alle Instanzen eines Prozesses müssen die abstrakte Methode `void body()` implementieren. Sie wird wiederholt aufgerufen, was in der `run()`-Methode geschieht. Sie ist von der Klasse `Thread` geerbt und beschreibt dessen Verhalten. Wie testen darin in einer Schleife, ob der Prozeß noch aktiv ist, sonst verlassen wir `run()` und der Thread terminiert.

```
public void run() {
  while (sim.isAlive(id)) body();
}
```

An dem Code sieht man auch, wie der Prozeß einen Request an den Simulator sendet, hier um seinen Status zu erfragen.

Beim Lesen eines Signals passiert im Prozeß nicht viel aufregendes, es werden die im Algorithmus beschriebenen Fallunterscheidungen gemacht und bei internen Signalen gewartet. Dies geschieht mit einem Aufruf einer Simulatormethode, die den Prozeß warten läßt, wenn es noch jüngere Prozesse gibt.

```
protected Object read(SignalProg p) {
  if (p instanceof Outport)
    throw new IllegalArgumentException("can't use read on Outport");
  if (p instanceof Signal) sim.waitNoYounger(id);
  return p.getValueAt(sim.getClock(id));
}
```

Die Methode ist vollständig widergegeben, um zu zeigen, wie Fehler behandelt werden. Typisch für Java geschieht dies durch das Werfen einer Ausnahme, immer mit einer kleinen Diagnosemeldung versehen.

Alle `write`-Methoden bedienen sich einer privaten Methode:

```
private void write(SignalProg p, Object val, int delay, int reject)
```

Die `writeT` Variationen setzen beim Aufruf von `write` zur Unterscheidung `reject` auf 0, da sie diesen Parameter nicht besitzen.

Um verstreute Einträge aus einer `DList` zu entfernen, eignet sich folgendes Vorgehen. Zuerst trägt man alle zu löschenden Einträge, genauer gesagt deren Iterator, in eine Liste ein. Nach Beendigung der Schleife kann man mit `remove(Iterator)` einfach alle Einträge in einem Aufwasch löschen. Damit umgeht man Probleme, die auftreten, wenn der Eintrag schon in der ersten Schleife entfernt wird, da dann der Iterator nicht mehr richtig weitergeschaltet werden kann.

```
for (int i = 0; i < remove.size(); i++)
  writeBuffer.remove(((DListIterator) remove.elementAt(i)));
```

So werden zuerst die Einträge beseitigt, die älter als die Zugriffszeit sind. Danach wird die neue Transaktion in den Writebuffer geschrieben und zum Abschluß werden noch diejenigen Einträge entfernt, die der Inertial Delay Bedingung widersprechen. Natürlich nur, wenn der Aufruf auch von einer `writeI`-Methode kam.

Die verschiedenen `wait`-Methoden sind aufwendiger. Sie bereiten den Writebuffer auf, bevor er vom Simulator mit `writeWBuffer` ausgewertet wird und die Signalverläufe beeinflußt. Der Algorithmen dazu wurden schon beschrieben und wir gehen hier nur auf Implementierungsdetails ein, die etwas schwieriger sind.

Die Methode `waitFor` schreibt alle Einträge des Writebuffers in einen temporären Buffer `validBuffer`, die nach den Kriterien des Algorithmuses zu aktivieren sind. Anschließend werden diese Einträge aus dem Writebuffer gelöscht (wie schon beschrieben) und der temporäre Buffer an den Simulator übergeben. Es muß noch darauf geachtet werden, daß der Prozeß seine Zeit hochzählt. Im Fall eines Deltadelays, wenn `t == 0`, entfällt dies, da der Delta Step schon in der Methode `writeWBuffer()` aufaddiert wurde.

```
// write entries of current period
sim.writeWBuffer(id, validBuffer);
// increase clock
if (t > 0) sim.setClock(id, floor(sim.getClock(id)) + t);
```

Bei `waitOn` besteht die Hauptaufgabe darin, den Zeitpunkt einer Signaländerung zu ermitteln. Dann kann der Rest mit `waitFor` erledigt werden. Zuerst werden Inports und prozeßeigene Signale untersucht, aber der entscheidende Aufruf geht wieder an den Simulator. Dazu übergibt man `signalChange` die aktuelle Zeit, eine eventuell bestimmte obere Schranke und die noch relevanten Signale.

```
// get time of Signal change
double changeT = -1;
if (testPorts.size() > 0)
  changeT = sim.signalChange(id, currentT, limitT, testPorts);
```

Wenn der Prozeß sich im Simulator befindet, dann kann er mit der Methode `wait()` zum Warten gebracht werden. Es ist unerläßlich, diesen Thread dann wieder mit `notifyAll()` zu reaktivieren.

4.8 Die Klasse Simulator

Der Simulator stellt für die Prozesse eine Art lokalen Server dar. Am Anfang der Simulation initialisiert der Simulator die Prozesse mit Startwerten für Clock, Prozeßzustand etc. Danach rufen die Prozesse Methoden des Simulators auf, um diese Werte zu ändern oder sich mit anderen zu synchronisieren. Abbildung 5 veranschaulicht diesen Vorgang.

4.8.1 Datenstrukturen

```
private Entitybase eb;
private int [] states;
private int proCount;
private int done = 0;
private double [] times;
private int clockStep;
private int startSim;
private int endSim;
```

Wenn ein Simulator erzeugt wird, was normalerweise in der Methode `simulate()` der `Entitybase` passiert, speichert er eine Referenz der Entity, die er simulieren soll, in `eb`. Der Zustand jedes Prozesses befindet sich im Array `states`, dafür sind einige Konstanten definiert worden.

```
private final int proRunning = 0;
private final int proWaitRead = 1;
private final int proWaitWrite = 2;
private final int proWaitChange = 3;
private final int proDone = 99;
```

Die Anzahl der Prozesse der Architecture ist in `proCount`, die der schon terminierten in `done` abgelegt. Mit dieser Variable testet der Simulator, ob er noch laufen soll. Die Clocks sind im Array `times` gespeichert. Dann braucht der Simulator auch noch die Länge eines Clockzykluses und die Start- und Endzeiten. Sie sind in den entsprechenden Datenmitglieder abgelegt und werden im Konstruktor gesetzt.

Die Zeit eines Prozesses ist in einem `double` kodiert. Der ganzzahlige Anteil stellt die reale Simulationszeit dar. Es ist üblich, eine Zeiteinheit so zu normieren, daß sie kleiner oder gleich der minimalen Gatterverzögerung ist. Deshalb sind Start-, Endzeiten und Clockzyklus als `ints` gespeichert. In der Mantisse werden Delta Steps gespeichert. Damit bleibt die zeitliche Ordnung auch mit Delta Delays problemlos erhalten. Die Konstante `deltaDelay` gibt an, wie groß (numerisch) solch ein Delta Step ist. Damit hat man gleichzeitig die Anzahl der maximalen Delta Steps festgelegt: $\Delta_{max} = 1/\Delta_{delay}$. Die Konstante hat den Defaultwert

```
private final double deltaDelay = 0.01.
```

4.8.2 Methoden

Der Konstruktor ruft zuerst die `init()` Methode der Architecture auf, damit alle Vektoren der `Entitybase` richtig initialisiert sind. Dann wird die Anzahl der Prozesse bestimmt und alle Daten auf Defaultwerte gesetzt.

Die Methode `void simulate()` startet alle Prozesse und wartet anschließend, bis sie terminieren. Danach zeigt sie noch eine Liste der Signale, deren Trace-Flag auf `true` gesetzt ist.

In `writeWBuffer()` ist der Algorithmus *Writebuffer schreiben* buchstabengetreu implementiert. Mit dem Befehl `notifyAll()` teilt man am deren Ende allen wartenden Threads mit, daß eine möglicherweise signifikante Signaländerung erfolgte. Diese starten dann ihrerseits Tests und reagieren entsprechend.

Der Simulator hat auch viele Methoden, die der einfachen Manipulation der Datenstrukturen dienen, also um Zeiten zu ändern, den Prozeßstatus zu ändern und ähnliches. Exemplarisch dafür hier die Methode um einen Prozeß zu beenden.

```
synchronized protected void endProcess(int id) {
  setState(id, proDone);
  done++;
  notifyAll();
  ((Thread) eb.Processes.elementAt(id)).stop();
}
```

Zuerst wird der Status auf `proDone` gesetzt und der Zähler der fertigen Prozesse erhöht. Danach ruft man `notifyAll()` auf, um andere Prozesse vom eigenen Ableben zu informieren. Zuletzt wird der Thread, der den Prozess beherbergt, gestoppt.

Von der `read`-Methode benötigt wird `waitNoYounger()`. In ihr wartet der Simulator solange, bis es keinen Prozeß mehr gibt, der an einem Signal Änderungen vornehmen kann, die dessen Wert zum aktuellen Zeitpunkt ungültig machen kann. Die Logik dahinter ist in Schritt (3) des Algorithmuses *Signal lesen* erläutert. Wichtig ist, wie man einen Prozeß, oder genauer gesagt dessen „Wirts-Thread", vorübergehend warten läßt.

```
notifyAll();
try {
  setState(id, proWaitRead);
  wait();
}
catch (InterruptedException e) {}
setState(id, proRunning); // running again
```

Dazu wird der Prozeßstatus auf `proWaitRead` gesetzt und die `wait()` Methode aufgerufen. Damit gibt der Thread den Monitor des Simulators auf und andere, wartende Threads können versuchen weiterzulaufen. Wird der Prozeß wieder aufgeweckt, setzt er den Status zurück und durchläuft die Schleife erneut.

Ganz analog arbeitet auch die Methode `signalChange()`, aber natürlich mit einer anderen Logik, die im Algorithmus *Auf Signaländerungen warten* definiert ist.

5 Beispiele und Ergebnisse

Die vorgestellten Konzepte sollen nun anhand eines Beispieles verdeutlicht werden. Die Resultate und erzielten Laufzeiten werden mit dem Simulator QuickVHDL verglichen.

5.1 Das Beispiel Primzahlgenerator

Als Beispiel dient ein einfacher Primzahlgenerator. Es ist etwas synthetisch, verdeutlicht aber gut die Interaktion mehrerer Prozesse über Signale und `wait`-Methoden.

Die `Entity` ist durch einen einzigen `Outport` beschrieben, an dem die gefundenen Primzahlen anliegen.

```
import EDU.UniTuebingen.RT.Entitybase;
import EDU.UniTuebingen.RT.Outport;

// Entity has only one Outport
public abstract class Entity extends Entitybase {
  Outport out = getOutport(new Integer(0));
}
```

In der `Entity` werden zwei Signale benötigt, `testnr` und `prime`, die auf dem Typ `Integer` basieren. Außerdem kommen noch drei Prozesse hinzu. Der erste, `generator`, erzeugt ungerade Zahlen und legt sie auf das Signal `testnr`. Der Prozeß `prime_tester` wartet, bis an diesem Signal ein neuer Wert anliegt und testet dann, ob es sich um eine Primzahl handelt. In diesem Fall wird der Wert auf das Signal `prime` geschrieben, auf das der Prozeß `collector` wartet. Dieser liest den Signalwert aus und schreibt ihn auf den Outport `out`. Abbildung 6 skizziert dieses Design.

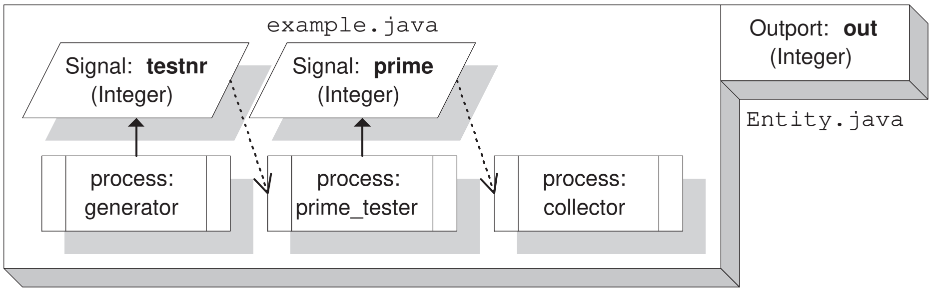

Abbildung 6: Design des Beispiels Primzahlgenerator

In der `main`-Methode wird eine Instanz des Beispiels erzeugt, die mit einem Clockzyklus von 10 Einheiten bis zum Zeitpunkt 2000 läuft.

5.1.1 Source des Beispiels Primzahlgenerator in Java

```
// JDK classes
import java.util.Vector;
// RT classes
import EDU.UniTuebingen.RT.Signal;
import EDU.UniTuebingen.RT.process;
```

```
public class example extends Entity {
  // define the architecture
  Signal testnr = getSignal(new Integer(0));
  Signal prime = getSignal(new Integer(0));
  public void init() {
    out.traceOn();
    // add processes
    addProcess(new generator());
    addProcess(new collector());
    addProcess(new prime_tester());
  }

  // number generator
  class generator extends process {
    int nr = 3;
    public void body() {
      writeT(testnr, new Integer(nr), 2);
      nr += 2;
      nextCycle(); // waitFor(10);
    }
  }
  // worker process: tests if number is prime
  class prime_tester extends process {
    Vector waitV = new Vector();
    prime_tester() { waitV.addElement(testnr); }
    int nr;
    boolean prime_flag;
    public void body() {
      waitOn(waitV);
      nr = ((Integer) read(testnr)).intValue();
      for (int ii = 0; ii < 2000; ii++) {
        prime_flag = true;
        for (int i = 2; i < (nr / 2); i++) {
          if ((nr % i) == 0) {
            prime_flag = false;
            break;
          }
        }
      }
      if (prime_flag) writeI(prime, new Integer(nr), 4);
      waitFor(6);
    }
  }
  // wait on primes
  class collector extends process {
    Vector waitV = new Vector();
    collector() { waitV.addElement(prime); }
    public void body() {
      waitOn(waitV);
      writeT(out, read(prime), 2);
    }
  }

  // ----- main program -----
```

```
  public static void main(String args[]) {
    example X = new example();
    X.simulate(10, 0, 2000);
  }
}
```

5.1.2 Source des Beispiels Primzahlgenerator in VHDL

```
entity prime is
  port(outp : out natural );
end;

architecture prime1 of prime is
  signal testnr : natural := 0;
  signal prime : natural := 0;

begin
  process
  variable nr : natural := 3;
  begin
    testnr <= transport nr after 2 ns;
    nr := nr + 2;
    wait for 10 ns;
  end process;

  process
  variable nr, d : natural;
  variable prime_flag : boolean;
  begin
    wait on testnr;
    nr := testnr;
    for ii in 0 to 2000 loop
      prime_flag := true;
      for i in 2 to (nr / 2) loop
        if ((nr rem i) = 0) then
          prime_flag := false;
          next;
        end if;
      end loop;
    end loop;

    if (prime_flag = true) then prime <= testnr after 4 ns;
    end if;
    wait for 6 ns;
  end process;

  process
  begin
    wait on prime;
    outp <= prime after 2 ns;
  end process;
end;
```

5.1.3 Ausgaben

Die letzten zehn Werte, die am Output-Port anliegen, sollen als Vergleich der Simulationsergebnisse unter Java und QuickVHDL dienen. Sie sind in Tabelle 1 zu finden. Die Java-Werte sind direkt aus dem Trace des Ports übernommen und in der linken Spalte zu sehen. Die Werte des Output-Ports von QuickVHDL sind aus einem Listing der anliegenden Werte nach einem Simulationslauf entnommen. Die Werte stimmen exakt mit denen aus der Java-Spalte überein, was ein Indiz für die Korrektheit unseres Simulators ist. Man sieht auch deutlich, wie die Delta-Steps bei unserem Simulator in der Mantisse der Zeit kodiert sind, wohingegen QuickVHDL sie gesondert behandelt.

Zeit	Wert	Zeit	Delta-Step	Wert
...	...	...	...	...
1738.0	349	1738	+0	349
1758.0	353	1758	+0	353
1788.0	359	1788	+0	359
1828.0	367	1828	+0	367
1858.0	373	1858	+0	373
1888.0	379	1888	+0	379
1908.0	383	1908	+0	383
1938.0	389	1938	+0	389
1978.0	397	1978	+0	397
1998.0	401	1998	+0	401

Tabelle 1: Ausgabe des Output-Ports des Java-Simulators und von QuickVHDL

5.1.4 Laufzeitmessungen

Die Laufzeitmessungen erfolgten auf einer Sparc 10 mit Solaris 2.5.1[1], jeweils mit dem Java-Interpreter von Sun des JDK 1.1.3 und dem Simulator `qvsim v8.4_4.3e` aus QuickVHDL. Dazu kommen Messungen auf einer Windows NT 4.0 Maschine mit einem Dualboard, das mit zwei Pentium 133 bestückt ist. Sie erfolgten mit dem Java-Interpreter von MicroSoft (`jview v4.79`) und dem des JDK 1.1.3. Alle Messungen sind in Tabelle 2 zusammengefaßt.

Simulator	Rechner	t_1	t_2	t_3	Durchschnitt
QuickVHDL	Sparc 10	24 s	24 s	24 s	24 s
JDK 1.1.3	Sparc 10	31.2 s	31.1 s	31.1 s	31.1 s
JDK 1.1.3	NT 4.0	22.7 s	22.6 s	22.6 s	22.6 s
MS Java	NT 4.0	7.3 s	7.0 s	7.0 s	7.1 s

Tabelle 2: Messungen zum Beispiel Primzahlgenerator

5.1.5 Auswertung des Beispiels Primzahlgenerator

Die Korrektheit unseres Simulators und der dafür entworfenen Algorithmen wird durch die Ausgaben des Beispiels unterstrichen.

[1] `ploch.informatik.uni-tuebingen.de`

Die Laufzeit des Beispiels mit unserem System ist mit dem Standardinterpreter des JDK etwas länger als die des Simulators des QuickVHDL-Pakets. Der Einsatz eines JIT-Compilers im Java-Runtimesystem verschiebt die Relationen aber zu unseren Gunsten, was die Messungen mit dem Interpreter von MicroSoft zeigen. Steigt der algorithmische Gehalt in der Verhaltensbeschreibung, so wächst der Vorsprung unseres Systems noch weiter an.

Eine weitere Beobachtung besteht darin, daß die Prozeßsynchronisation einen beträchtlichen Teil der Laufzeit verschlingt. Mit jedem zusätzlichen Prozeß verliert der Simulator hier viel Rechenzeit. Im obigen Beispiel beträgt der Anteil der Prozeßsynchronisation ca. 60% der gesamten Laufzeit[2]. In ernsthaften Anwendungen sind aber die Aufgaben innerhalb der Prozesse viel komplexer, wodurch dieser Overhead relativiert wird.

6 Zusammenfassung und Ausblick

Das vorrangige Ziel dieser Arbeit war es, zu zeigen, daß die Simulation von Verhaltensbeschreibungen digitaler Systeme, in der von uns gewählten Sprache Java, realisierbar ist. Um dies zu erreichen, mußte ein Framework erstellt werden, in dem Beschreibungen erfolgen und anschließend getestet werden können. Dazu wurden die Konzepte und nötigen Algorithmen vorgestellt, gefolgt von der Erläuterung der Referenzimplementierung in Java.

Die angeführten Beispiele und deren Ergebnisse sind ein deutlicher Beleg für das Erreichen unseres primären Ziels. Die Resultate decken sich in allen Fällen mit denen anderer Simulatoren. Es bleibt anzumerken, daß die Korrektheit der Algorithmen noch gesondert bewiesen werden muß.

Die Laufzeitmessungen zeigen aber auch, daß noch Handlungsbedarf im Bereich der Performance des Systems besteht. Hier sollte vor allem im Bereich der Prozeßsynchronisation noch einiges an Leistung herauszuholen sein. Der Einsatz von Just-In-Time-Compilern ist eine weitere Möglichkeit, das Laufzeitverhalten des Java-Simulators stark zu verbessern, was die Messungen mit der Runtime-Umgebung von MicroSoft illustrieren. Diese setzt schon einen Just-In-Time-Compiler ein.

Das vorgestellte System stellt eine gute Basis dar, um ein komplettes und effizientes Paket zur Simulation und Beschreibung digitaler Systeme aufzubauen. Hier kann die Entwicklung auch in Richtung von Verhaltensbeschreibungen oberhalb der RT-Ebene gehen. Sie zeichnen sich dadurch aus, daß das signalgesteuerte Taktverhalten entfällt, sie verwenden ein *kausales* Timing. Diese Beschreibungen sind noch enger verwandt mit imperativen Programmiersprachen.

Unser System bietet momentan noch sehr wenig Komfort, das Setzen von Eingabesignalen geschieht programmtechnisch, die Ausgabe der Signale erfolgt einfach auf die Kommanozeile. Hier setzt eine andere, noch in der Entwicklung befindliche, Arbeit an. Sie visualisiert die Simulationsergebnisse mit Hilfe der reichhaltigen Graphik- und Window-Möglichkeiten von Java.

Die konsequente Parallelisierung von Anfang an ist ein Vorteil unseres Simulators. Er kann über das Thread-Konzept von Java sofort Mutliprozessorarchitekturen ausnutzen. Ebenfalls leicht möglich ist eine Erweiterung für verteilte Systeme mittels der JGL, die es erlaubt, Container auf Remote-Systemen zu erzeugen. Dies wäre ein weiterer interessanter Aspekt, der für zukünftige Forschung im Auge behalten werden sollte.

[2]mittels Profiling-Datei der Laufzeitumgebung bestimmt

Literatur

[1] Daniel J. Berg: *Java Threads - A Whitepaper*, Sun Microsystems Computer Corporation, California, 1996.

[2] David Flanagan: *Java in a Nutshell*, O'Reilly & Associates, Inc., 1996.

[3] *IEEE Standard VHDL Language Reference Manual - Std 1076-1993*, IEEE, New York, 1993.

[4] A. Mäder: *VHDL Kurzbeschreibung*, Universität Hamburg, 1993.

[5] Peter Marwedel: *Synthese und Simulation von VLSI Systemen*, Carl Hauser Verlag, München, 1993.

[6] David R. Musser, Atul Saini: *STL Tutorial and Reference Guide*, Addison-Wesley Publishing Company, 1996.

[7] Object Space Developer Team: *JGL - the Generic Collection Library*, 1996. available at `http://www.objectspace.com/jgl`

[8] Bjarne Stroustrup: *The C++ Programming Language, 2nd Edition*, Addison-Wesley Publishing Company, 1991.

[9] D. Tavangarian: *VHDL-Based Simulation of Electronic Circuits and Systems*, University of Rostock, 1995.

[10] Flávio R. Wagner: *Simulation of Digital Systems*, University of Tübingen, 1993/94.