Universität Leipzig
Fakultät für Mathematik und Informatik
Mathematisches Institut

Entwurf und Implementierung eines parallelen Logiksimulators auf Basis von TEXSIM

Diplomarbeit

vorgelegt von
DENIS DÖHLER

Leipzig, Dezember 1996

Inhaltsverzeichnis

Abbildungsverzeichnis iii

Tabellenverzeichnis iv

1. Einleitung 1
   - 1.1. Allgemeines . . . 1
   - 1.2. Danksagung . . . 2
   - 1.3. Inhalt . . . 2
   - 1.4. Verschiedenes . . . 3

2. Simulation 4
   - 2.1. Logiksimulation . . . 4
   - 2.2. Die Modellzeit . . . 7
   - 2.3. Parallele zeitgesteuerte Logiksimulation . . . 10

3. Grundlagen 12
   - 3.1. Der sequentielle Simulator TEXSIM . . . 12
   - 3.2. Logiksimulation mit TEXSIM . . . 14
   - 3.3. Der TEXSIM-Client Xmon . . . 20
   - 3.4. Voraussetzungen . . . 23

4. Der Logiksimulator *parallel*TEXSIM 24
   - 4.1. Aufbau . . . 24
   - 4.2. Das parallele Simulationsmodell . . . 26
   - 4.3. Das Shellskript pTEXSIM . . . 31
   - 4.4. Die Slaves sTEXSIM . . . 32
     - 4.4.1. Die Programm-Struktur . . . 32
     - 4.4.2. Der Nachrichtenaustausch . . . 35

4.4.3. Der parallele Clock-Cycle-Algorithmus 37
4.5. Der Master mTEXSIM 39
4.5.1. Aufbau 39
4.5.2. Die Schnittstellen 43
4.6. Praktische Handhabung 45
4.6.1. Der Programmstart 45
4.6.2. Die Programmkonfiguration 46

5. Der Clock-Cycle-Algorithmus 50

5.1. Das strukturelle Hardware-Modell 50
5.2. Das Box-Levelizing 51
5.3. Das funktionelle Hardware-Modell 55
5.3.1. Definition des funktionellen Hardware-Modells 55
5.3.2. Zur Interpretation des funktionellen Hardware-Modells 56
5.3.3. Hardware-Zustände und das funktionelle Hardware-Modell 57
5.4. Der sequentielle Clock-Cycle-Algorithmus 58
5.5. Die Cone-Darstellung 61
5.6. Cone-basierte Modellpartitionierung 63
5.7. Der parallele Clock-Cycle-Algorithmus 67

6. Zusammenfassung 68

6.1. Ergebnisse 68
6.2. Schlußfolgerungen 69
6.3. Ausblick 69

Literatur 72

Anlagenverzeichnis 74

Abbildungsverzeichnis

1. Abstraktionstufen im Logikdesign 5
2. Verifikationsprozeß in der Mikroprozessorentwicklung 6
3. Hierarchie einer Schaltung 7
4. Synchrone Schaltungsentwürfe 9
5. Cone in einer synchronen Schaltung 11
6. DA_DB und TEXSIM 13
7. Aufbau einer TEXSIM-Facility 15
8. Das ALTER-CLOCK-RETRIEVE Schema 16
9. Das TEXSIM Client-Server-Prinzip 18
10. Grundablauf eines TEXSIM-User-Programs 19
11. C-Struktur Facility-Referenz 19
12. Der Programmablauf von TEXSIM 21
13. Einordnung des Xmon API 22
14. Simulation mit *parallel*TEXSIM 24
15. Namenskonvention für das TEXSIM-Modell 26
16. Namenskonvention für die Modellblöcke 26
17. Facility-Hierarchie 27
18. Der Aufbau von sTEXSIM 32
19. Aufbau der Communication Facilities 33
20. Zugriff auf ein geschnittenes Netz 34
21. Verweis auf eine Message-Handler-Funktion 35
22. Struktur der Master-Slave-Messages 36
23. Implementierung des parallelen Clock Cycle 37
24. Datentransfer beim parallelen Clock Cycle 38
25. Aufbau eines Communication Vector 39
26. Der Aufbau von mTEXSIM 40

27. Struktur der Facilities 41
28. Zugriff auf eine aufgeteilte Facility 44
29. ptexsim.cfg-Beispiel für Workstation-Cluster 48
30. ptexsim.cfg-Beispiel für Xmon und RS/6000 SP 49
31. Der sequentielle Clock-Cycle-Algorithmus 59
32. Der parallele Clock-Cycle-Algorithmus 67

Tabellenverzeichnis

1. Typographische Konventionen 3
2. ALTER-CLOCK-RETRIEVE 16
3. Teilprogramme von *parallel*TEXSIM 25
4. Format der pmod-Files für die Slaves 28
5. Format der pmod-Files für den Master 29

1. Einleitung

1.1. Allgemeines

Die Simulation von Modellen natürlicher und künstlicher Vorgänge auf Computern ist in Wissenschaft und Technik unverzichtbares Instrument zu deren Analyse und Prognose und damit zum Phänomen gesellschaftlichen Ausmaßes geworden. Im Alltag wird dies beispielsweise durch die laufenden Wettervorhersagen stets aufs neue dokumentiert. Mit Simulationen zur Entstehung und Entwicklung des Universums, der Erde und des Lebens oder von gesellschaftlichen Prozessen und deren Einfluß auf das Weltklima sind durch den Rechnereinsatz Probleme mit philosophischer oder politischer Relevanz überhaupt erst umfassend behandelbar geworden. Weiterhin führt die unaufhörlich fortschreitende Vervollkommnung der Technik zu immer komplexeren Geräten, deren Entwurf durch effiziente Maßnahmen für deren Funktionsüberprüfung zu begleiten ist. Auch hierbei hat sich die Verwendung von Computern als der einzig gangbare Weg erwiesen.

Da die entsprechenden Simulationsprogramme naturgemäß zumeist enorme Anforderungen an Rechenleistung und Speicherbedarf stellen, sind geeignete Mittel und Methoden zu deren Bewältigung gefragt. Durch die Erhöhung des Rechen- und Speicherpotentials werden zudem komplexere und damit umfassendere Simulationen möglich, die wiederum zu besseren Analysen und Prognosen führen. In den letzten Jahren hat es sich immer mehr herausgestellt, daß der Einsatz von parallelen und verteilten Rechensystemen und der damit einhergehenden Parallelisierung der Software der vielversprechendste Weg zur Leistungssteigerung ist.

Die wachsende Anzahl leistungsfähiger Parallelrechner steht jedoch im Widerspruch zum gegenwärtig geringen Umfang an Software, die solche Architekturen effizient nutzt. Dies behindert die weite Verbreitung paralleler Methoden, und in der Literatur wird daher von der „Parallel Software Crisis“ gesprochen. In dieser Arbeit wird als ein Beitrag zur Überwindung dieser Krise ein konkretes Parallelisierungsprojekt aus dem Bereich der Ingenieurwissenschaften detailliert vorgestellt.

Bei der seit vielen Jahren konstant wachsenden Zahl von integrierten logischen Elementen ist die funktionelle Simulation der Logik von Mikroprozessoren mit dem Problem extrem wachsender Laufzeiten und Speicheranforderungen entsprechender Simulatoren verbunden. Innerhalb der Firmen IBM und Motorola wird zur Logiksimulation im Prozessorentwurf sehr oft das Programm TEXSIM (TEXas SIMulator) eingesetzt. Trotz der im Vergleich zu ähnlichen Programmen sehr hohen Arbeitsgeschwindigkeit von TEXSIM kann gegenwärtig das Verhältnis der simulierten zur tatsächlichen CPU-Zeit Werte zwischen 10^5 und 10^7 erreichen. Durch die in dieser Diplomarbeit durchgeführte Parallelisierung von TEXSIM soll diesem wirtschaftlich nicht nur für IBM und Motorola sehr bedeutsamen Problem beizukommen versucht werden.

In verschiedenen Projekten wurden schon mehrere, hauptsächlich experimentelle Logiksimulatoren parallelisiert bzw. neu entwickelt [1]. Die Besonderheit der vorliegenden Arbeit liegt

darin, daß es sich bei TEXSIM um ein praxisrelevantes Programm handelt, welches auf Basis des *Clock-Cycle*-Algorithmus zeitgesteuert arbeitet. Dem Verfasser sind keine Simulatoren bekannt, bei denen ein paralleler Clock-Cycle-Algorithmus auf lose-gekoppelten Architekturen implementiert ist. (Auf SMP-Workstations von Sun kann SpeedSim/3 bis zu 8 Prozessoren über Multi-Threading nutzen.)

TEXSIM ist in ein komplexes Umfeld verschiedenster, aufeinander abgestimmter Design-Tools eingebettet. Aufgrund einer Kooperation mit IBM Böblingen konnte die Funktionsfähigkeit der parallelisierten Version *parallel*TEXSIM mit Tests an realen Prozessormodellen (CMOS S/390 wie PICASSO, MONET/J, MONET[1]) mit Hilfe der speziellen TEXSIM-Erweiterung Xmon ausgiebig überprüft werden. IBM stellte außerdem Tests für den PowerPC 604 aus der Kooperation mit Apple und Motorola bereit, so daß die Kompatibilität auch für andere Design-Methoden überprüfbar war.

1.2. Danksagung

An erster Stelle möchte ich Prof. SPRUTH für die Vergabe des interessanten Themas dieser Diplomarbeit und seinem Assistenten Dr. HERING für die Betreuung der Arbeit und die anregenden Diskussionen danken. Auch alle anderen am Forschungsprojekt Beteiligten, insbesondere Herr REILEIN aufgrund seiner stets hervorragenden Zusammenarbeit, seien darin mit eingeschlossen.

Durch die IBM Entwicklungslabors Böblingen und Austin wurde ich ideell und materiell (besonders in Form mehrerer Studienaufenthalte) unterstützt. Hier seien vor allem die Herren ANDERSON, GERST, ROESNER und ZIKE genannt, die mir mit ihrem fachlichen Rat zur Seite standen.

1.3. Inhalt

In Abschnitt 2 dieser Dokumentation werden Simulationsparadigmen unter dem Aspekt der parallelen zeitgesteuerten Logiksimulation vorgestellt. Danach werden in Abschnitt 3 einige Grundlagen zu TEXSIM erläutert. Abschnitt 4 ist dann vollkommen Entwurf, Implementierung und Handhabung von *parallel*TEXSIM gewidmet.

Im Abschnitt 5 wird ein funktionelles Hardware-Modell eingeführt, auf dessen Grundlage der Clock-Cycle-Algorithmus mathematisch formuliert wird. Daran schließt sich die formale Beschreibung des parallelen Clock-Cycle-Algorithmus an.

Der letzte Abschnitt 6 faßt die Ergebnisse dieser Diplomarbeit zusammen. Anhang A listet die Funktionen des TEXSIM-API und ihre Unterstützung durch *parallel*TEXSIM auf. In Anhang B

[1]hierbei handelt es sich um die während der Entwicklung verwendeten Codenamen

werden noch einmal die Formate für Cross-Referenz- und Signalschnitt-Listen aus [2] angegeben. Die Befehle des TEXSIM-Subcommand-Interfaces listet Anhang C auf. Erste Ergebnisse aus Performance-Messungen von *parallel*TEXSIM im Vergleich zu TEXSIM finden sich in Anhang D.

1.4. Verschiedenes

Zur visuellen Unterstützung des schnellen Auffindens von gesuchten Informationen sowie aus Gründen der besseren Lesbarkeit werden die durch Tabelle 1 gesetzten typographischen Standards im gesamten Dokument (ausgenommen das Literaturverzeichnis) verwendet.

Schriftstil	Verwendung
`Type Writer`	Programm- und Dateinamen
	Quell-Codes
	Shell-Eingaben
emphasize	besondere Hervorhebung
Sans Serif	Geschützte Bezeichnungen
SMALL CAPS	Personennamen

Tabelle 1: Typographische Konventionen

Im Text werden Soft- und Hardwarebezeichnungen erwähnt. In vielen Fällen sind diese auch eingetragene Warenzeichen oder geschützte Namen und unterliegen als solche den gesetzlichen Bestimmungen.

*parallel*TEXSIM baut auf dem Quellcode von TEXSIM auf. Dieser ist Eigentum von IBM Austin und ebenso wie TEXSIM als „IBM internal use only“ deklariert. In Ergänzung zum IBM Copyright auf *parallel*TEXSIM wurde vom Autor ebenfalls eines auf das Programm erhoben, mit denen die von diesem vorgenommenen Erweiterungen geschützt werden sollen. Zudem ergab sich dafür die Notwendigkeit aufgrund eines vom Autor mit IBM Böblingen geschlossenen Vertrages. Es handelt sich hierbei daher weder um Public Domain, Freeware oder ähnliche Formen von freier Software.

Die über SpeedSim/3 im Text gemachten Aussagen beruhen auf einer eMail, in der DOUG DAY, Vice President of Sales der Hersteller-Firma SpeedSim, auf Fragen des Autors geantwortet hat.

2. Simulation

Nach [3] ist Simulation (lat. *simulare*: nachbilden, vortäuschen) ein Hilfsmittel, um das Verhalten komplexer Systeme zu analysieren. Beispiele dafür sind Raumflüge, Elementarteilchen, rotierende Sterne, Organismen oder Mikroprozessoren. Durch die Vereinfachung des Sytems in Form eines Modells kann dieses in einer Simulation nachgebildet werden. Die Vereinfachung geschieht dabei so, daß sich das Modell in Bezug auf relevante Aspekte wie das reale System verhält. Dadurch wird in einem gewissen Rahmen eine Analyse und Verhaltensprognose des Systems ermöglicht.

Für die rechnergestützte Systemuntersuchung ergeben sich aus verschiedenen Gründen Notwendigkeiten. Beispielsweise sind sehr oft Bedingungen zu betrachten, die in der Realität entweder zu aufwendig (z.B. Katastrophenfälle), zu gefährlich oder schädlich (dazu zählen leider auch Nuklearwaffentests[2]) oder schlicht unmöglich sind. In seiner Diplomarbeit [4] beschreibt STEPHAN WINTERSTEIN-THEOBALD weitere Gründe:

> „Außerdem können in der Realität gar nicht existierende Systeme untersucht werden (z.B. neu entwickelte Mikrochips); Vorgänge, die sehr langsam oder sehr schnell ablaufen (z.B. die Entstehung des Universums), können in einem faßbaren Zeitrahmen abgebildet werden. Neben Analyse, Prognose und Validierung wird Simulation zunehmend auch dazu benutzt, die Realität (*eine* Realität) so exakt wie möglich nachzubilden und für die menschlichen Sinne erfahrbar zu machen, was u.a. der Ausbildung und Unterhaltung dienen kann; Flugsimulatoren und *Virtual Reality*-Systeme sind Beispiele dafür."

Daß die sich immer weiter vervollkommnende Computersimulation auch ernsthafte Gefahren insbesondere in Form der *Virtual Reality* mit sich bringt, ist seit Mitte der 60er Jahre durch die fundamentalen Werke[3] von STANISŁAW LEM bekannt, aber erst mit der allgemeinen Verfügbarkeit entsprechender Verfahren in das öffentliche Bewußtsein gedrungen.

2.1. Logiksimulation

Ein wichtiger Bestandteil der Entwicklung neuer Computersysteme ist der VLSI-Schaltkreisentwurf, der in den letzten Jahren auf Grund der stetig wachsenden Zahl von funktionellen Elementen je Chip zunehmend automatisiert wurde. Mit den größten Mikroprozessoren bewegt man sich heutzutage in Größenordnungen von mehreren zehn Millionen integrierten Transistoren, und in den nächsten Jahren wird sich dieser Wert um eine Größenordnung steigern. In

[2] M. B. KALINOWSKI: *Atomtests im Rechner: Ausweg oder Gefahr*. c't 2/96, S. 70–73.

[3] vor allem *Summa Technologiæ* (1964): „Phantomologie" und „Phantomatik"

der Prozessorentwicklung spielt daher die Simulation als Entwurfsverifikationsmethode eine entscheidende Rolle zur Fehlererkennung und -beseitigung.

VLSI-Entwürfe lassen sich auf verschiedenen Abstraktionsebenen betrachten [5]. Das breite Spektrum reicht dabei von der niedrigsten Stufe, den Diffussionsprozessen in kristallinen Strukturen (Prozeßebene), bis hin zur Betrachtung von auf verschiedenen Prozessoren laufenden kooperierenden Prozessen (Systemebene) als höchster Stufe. Mit der Erhöhung der Abstraktionsstufe sinkt die Komplexität und damit der Simulationsaufwand. Leider sind in der Literatur die Definitionen der Abstraktionsebenen sehr unscharf gefaßt und werden mitunter widersprüchlich verwendet.

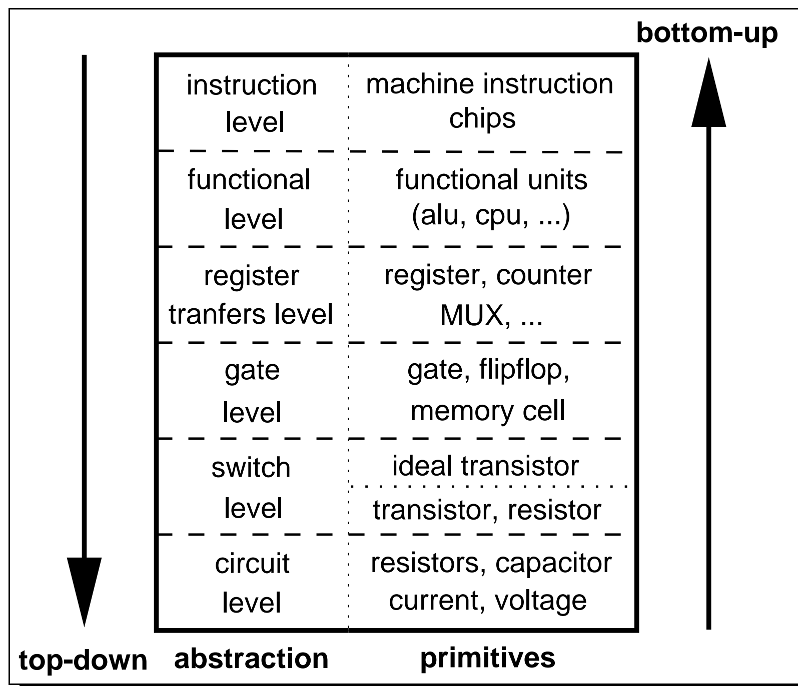


Abbildung 1: Abstraktionstufen im Logikdesign

Nach [1] werden die in Abbildung 1 dargestellten Abstraktionsstufen im Logikdesign behandelt und Simulationen auf diesen Ebenen als *Logiksimulation* bezeichnet. Im Designprozeß eines Mikroprozessors spielt seit der Entstehung leistungsfähiger Hardwarebeschreibungssprachen und Systemen zur Logiksynthese die Gate- und Register-Transfer-Ebene eine herausragende Rolle [6]. Die in Abbildung 2 gegebene schematische Darstellung[4] der dementsprechenden Entwicklungsmethode stammt aus [7]. Jedoch ist der Aufwand von Verifikationsmethoden auf diesen Ebenen trotz ihrer relativ hohen Abstraktion bei modernen Prozessoren sehr hoch: Die simulierten Zeiten differieren gegenüber den realen CPU-Zeiten um Faktoren, die in der Endphase der Entwicklung um bis zu *sieben* Größenordnungen darüber liegen können [8].

[4] Die meisten der darin auftretenden Begriffe werden im weiteren Verlauf noch erläutert.

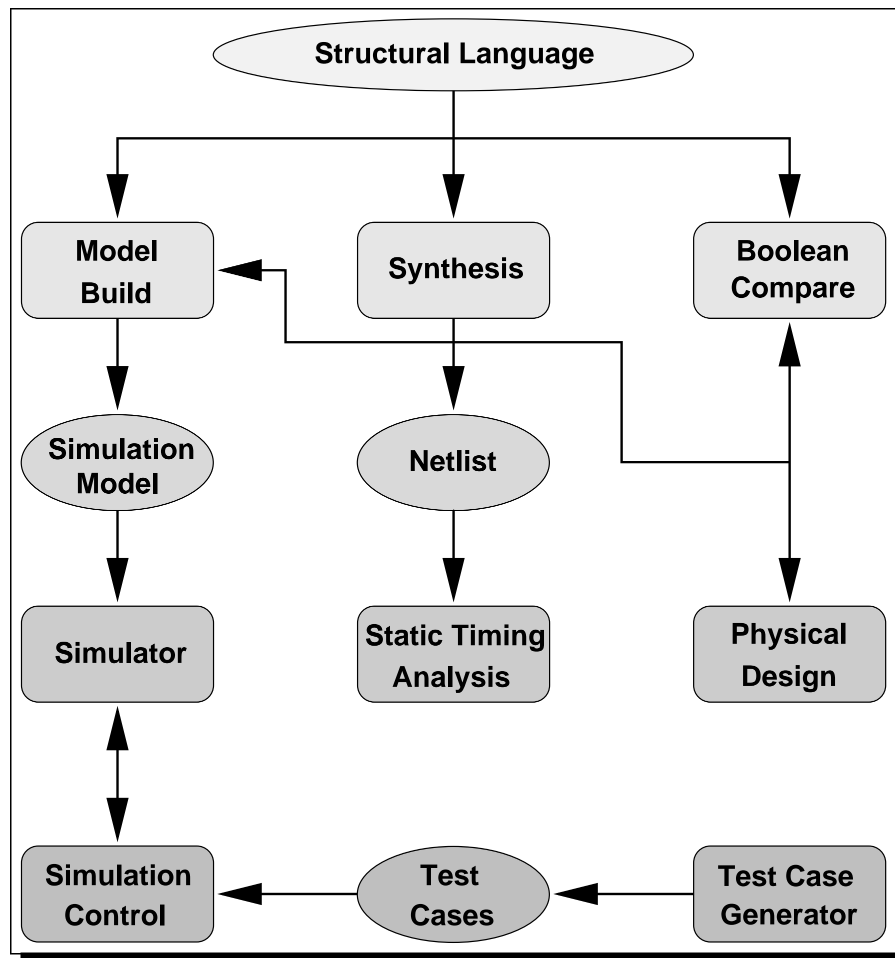


Abbildung 2: Verifikationsprozeß in der Mikroprozessorentwicklung

Abbildung 3 skizziert ein Beispiel für ein Modell einer Schaltung, bei dem die unterschiedlichen Hierarchien entsprechender Abstraktionen von Schaltungskomponenten deutlich werden. Dabei könnte es sich bei PIC um ein Mehrprozessor-System handeln, bestehend aus den Prozessoren P0, P1, ... Die Teilprozessoren bestehen ihrerseits wiederum aus einzelnen funktionalen Bestandteilen EU, CS usw. Diese lassen sich gemäß dem gewünschten Abstraktionsgrad noch weiter zerlegen. In einem zu simulierenden Modell können aber auch mehrere Abstraktionsstufen zugleich[5] auftreten.

Zur Simulation werden hierarchische Modelle zumeist in eine „ebene“ Form überführt. Man bezeichnet diesen Vorgang als *Flattening*. Dabei wird normalerweise die Logik repliziert, also ein vollständiges Flattening vorgenommen (z.B. beim Einsatz von TEXSIM). Gerade für typische SMP-Designs ist es aber günstiger, Referenzen für mehrfach vorhandene Bestandteile und

[5] Solche Modelle heißen mehrstufig (mixed level bzw. multi level).

einen speziellen Evaluationsalgorithmus zu verwenden, der die betreffenden Teile dann auch nur einmal zu evaluieren braucht.

Bei dem von TEXSIM abgeleiteten Simulator MVLSIM [9] wird mittels des auf D. S. ZIKE zurückgehenden sogenannten „Parallel Instance“ Features [7] derart verfahren. Als Resultat erhält man neben unter Umständen dramatisch kleineren Simulationsmodellen (bei vielen großen Teilprozessoren) vor allem einen deutlichen Geschwindigkeitsgewinn (einen nahezu linearen bei großen Booleschen Schaltungen). Dafür ist es jedoch notwendig, die in der Regel unterschiedlichen Inputs und damit auch Outputs der Instanzen durch sogenanntes „Packing“ und „Unpacking“ gesondert zu behandeln, was aber nicht immer möglich ist (Logik zur Speicheradressierung ist dafür das wichtigste Beispiel). In solchen Fällen muß eine serielle Evaluation der betreffenden Teile erfolgen.

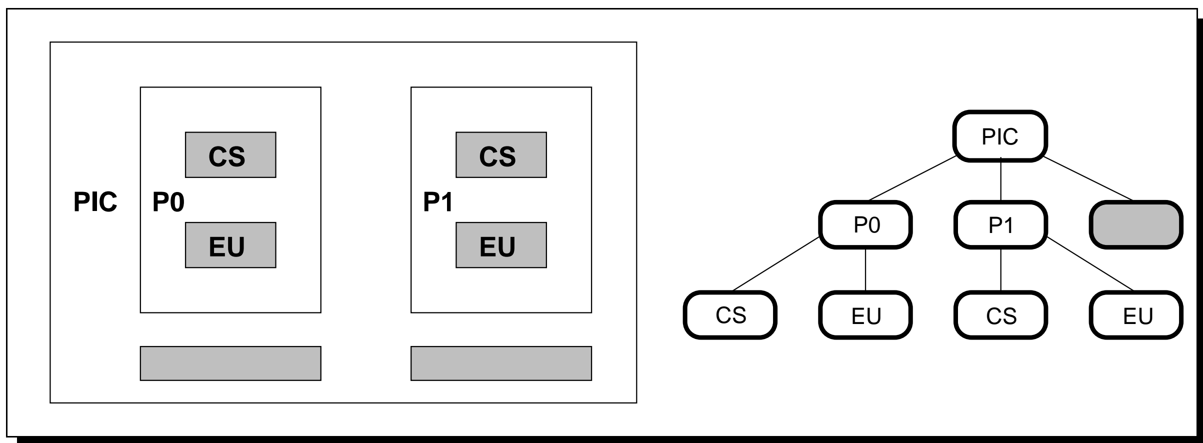


Abbildung 3: Hierarchie einer Schaltung

2.2. Die Modellzeit

Alle Simulationen haben, im Gegensatz etwa zu bloßen Visualisierungen, eines gemein: die fortschreitende Modellzeit als fundamentale Eigenschaft, die aber nicht mit der zur Durchführung der Simulation nötigen Zeit, welche natürlich in einem gewissen Bezug dazu steht, verwechselt werden darf. Eine wesentliche Unterscheidung ergibt sich jedoch hinsichtlich des Einflusses der Zeit auf die Parameter der zugrundeliegenden Modelle. Hängen diese nur von diskreten Zeitwerten ab, so spricht man von *diskreten*, andernfalls von *kontinuierlichen* Modellen und dementsprechend von diskreten und kontinuierlichen Simulationen.

Kontinuierliche Modelle sind zumeist durch Differentialgleichungssysteme mit der Zeit als freier Variablen gegeben — etwa in Wetter- und Klimaprognosen. Da hierfür nur selten exakte Lösungen bekannt sind, müssen diese mit dem Computer numerisch ermittelt werden. Unter

Vorgabe aller nötigen Parameter zu einem festen Startzeitpunkt läßt sich der Modellzustand dann für jeden beliebigen (späteren) Zustand näherungsweise berechnen.

Für die Logiksimulation klassifiziert MEISTER [1] die Möglichkeiten der Modellierung des zeitlichen Verhaltens von Schaltkreisen. Danach sind diese bis auf die unterste Abstraktionsebene diskret.

Ein Modell ist durch miteinander in Wechselwirkung stehende Parameter charakterisiert, deren Werte durch entsprechende Berechnungsverfahren definiert sind. Der Zustand eines Systems zu einem bestimmten Zeitpunkt ist durch die Parameterwerte gegeben. Im diskreten Fall läßt sich daher der zeitliche Einfluß auf das Modellverhalten genauer spezifizieren. Gibt es Modellparameter, deren Berechnung zu jedem Zeitpunkt unabhängig vom vorherigen Systemzustand *abgearbeitet* werden muß, so bezeichnet man solche Modelle bzw. Simulationen als *zeitgesteuert*. Ansonsten spricht man von *ereignisgesteuert*, da sich der Wert eines Parameters dann aus gewissen Parameteränderungen, die man als Ereignisse auffassen kann, ergibt.

In der Logiksimulation dominierte bislang eindeutig die Ereignissteuerung [10], da sich mit entsprechenden Simulatoren noch andere relevante Abstraktionsebenen effizient behandeln lassen. Dies betrifft zum einen alle höheren Stufen, zum anderen aber auch die nächstniederen, bei denen vor allem Fragen wie Signallaufzeiten von Bedeutung sind. Damit können mit ihnen alle Eigenschaften der mit allgemeinen Hardwarebeschreibungssprachen wie VHDL oder Verilog erzeugbaren Modelle simuliert werden.

Nur wenige Entwicklungsteams, insbesondere die einiger sehr großer Konzerne wie IBM, Digital und Intel, leisteten sich bislang den Luxus spezieller zeitgesteuerter Logiksimulatoren (*cycle-based simulation*). Allerdings hat sich in der Praxis deren Leistungsvorteil sehr deutlich gezeigt: TEXSIM beispielsweise ist gegenüber dem ereignisgesteuerten Simulator aus [6] um Größenordnungen schneller [2]. Und der seit 1995 erhältliche zeitgesteuertete Simulator SpeedSim/3 gilt (nach Herstelleraussagen[6]) gegenwärtig als der schnellste kommerziell verfügbare (nicht proprietäre) Logiksimulator, und soll je nach Modell 10 bis 100 Mal schneller sein, als entsprechende ereignisgesteuerte Simulatoren. Auch die Firma Synopsys gibt solche Speed-Up-Werte für ihren 1996 vorgestellten zeitgesteuerten Simulator Cyclop an. Es ist deshalb zu erwarten, daß die Bedeutung der zeitgesteuerten Simulation stark zunehmen wird.

Eine wichtige Eigenschaft von zeitgesteuerten Simulatoren ist die Einschränkung auf Schaltwerke vom HUFFMAN-Typ (HUFFMANN-Sequential-Networks [6]), die durch das Fehlen asynchroner Rückkopplungen gekennzeichnet sind (siehe Abbildung 4). Diese läßt sich jedoch in der Regel durch geeignete Abstraktionen umgehen, die die asynchrone Logik abkapseln.

Zudem existieren *hybride* Simulatoren (z.B. CycleDrive von FRONTLINE), die sowohl zeitgesteuert als auch ereignisgesteuert arbeiten [11]. Bei ihnen wird (nach einer zumeist automatischen Partitionierung in einen asynchronen und einen synchronen Teil) nur die synchrone Logik

[6] nach D. S. ZIKE von der Performance her mit TEXSIM vergleichbar

zeitgesteuert evaluiert, die übrige ereignisgesteuert. Der Nachteil dabei ist, daß der ereignisgesteuerte Simulator die Gesamtgeschwindigkeit negativ beeinflußt: Bei 10 % asynchroner Logik ist so nur eine maximal zehnfach schnellere Simulation möglich (vgl. auch Anhang D).

SpeedSim/3 ist auch ohne die erwähnten Hilfsmittel in der Lage, asynchrone Logik zu evaluieren. Dazu wird nach Herstellerangaben „um die asynchrone Logik herum evaluiert", was zu gewissen Performance-Einbußen führt.

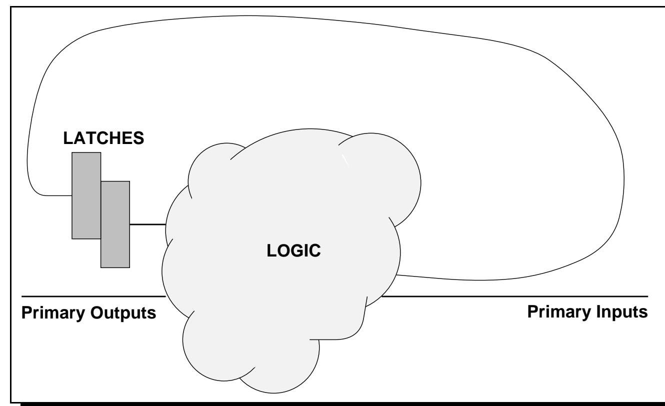


Abbildung 4: Synchrone Schaltungsentwürfe

Erwähnenswert ist noch der Begriff der *compilergesteuerten* Simulation, der im Zusammenhang mit der Logiksimulation gelegentlich verwendet wird. Die damit verbundene Vorstellung ist die, daß zum Compilationszeitpunkt, wenn die Modelldaten in für den verwendeten Simulator notwendige Datenstrukturen überführt werden (Model Build), die Art und Weise der Evaluation vorgegeben wird und nicht dynamisch an sich ändernde Anforderungen angepaßt werden kann.

In [1] meint MEISTER, damit die ereignisgesteuerte Simulation von der zeitgesteuerten unterscheiden zu können. Der Autor der vorliegenden Diplomarbeit vertritt aber die Auffassung, daß bei *allen* (also auch bei ereignisgesteuerten) durch Hardwarebeschreibungssprachen erzeugbaren Modellen deren Simulation im obigen Sinne compilergesteuert erfolgt.

Fast alle heute eingesetzten Logiksimulatoren funktionieren nach dem *Compiled Code*-Prinzip (z.B. dvsim, TEXSIM und MVLSIM), bei denen den oben erwähnten Berechnungsverfahren Maschinencode-Stücke entsprechen, die bei der Evaluierung der jeweiligen Elemente abgearbeitet werden. SpeedSim/3 dagegen arbeitet mit einer proprietären Technik, die vom Hersteller als Boolean Dataflow Engine bezeichnet wird, bei der man aber auch von einer compilergesteuerten Methode sprechen kann. Offenbar hat SpeedSim/3 deshalb keine Branch-Prediction-Probleme, die sonst bei der Simulation sehr großer Modelle auf bestimmten Prozessorarchitekturen auftreten.

2.3. Parallele zeitgesteuerte Logiksimulation

Um die Komplexität der Simulationen meistern zu können, ist Parallelisierung eine geeignete Vorgehensweise. Dafür gibt es drei Methoden: *funktionelle Parallelität*, *Daten-Parallelität* und *modellinhärente Parallelität*. Für die Logiksimulation werden sie z.B. in [1, 12] skizziert.

Der Parallelisierung von TEXSIM liegt die modellinhärente Parallelität zugrunde, da diese die besten Ansatzpunkte zur Reduktion der Komplexität der zeitgesteuerten Logiksimulation verspricht [12]. Hierbei wird das die Schaltung verkörpernde Modell in parallel zu simulierende Teilmodelle, die sogenannten *Blöcke*, zerlegt [8]. Ein solches Zerlegungsverfahren wird als *Partitionierung* bezeichnet und hat sehr großen Einfluß auf die Effizienz der Parallelisierung [8, 13, 14, 15].

Die vorliegende Diplomarbeit wurde im Rahmen eines in Kooperation mit IBM Böblingen am Institut für Informatik der Universität Leipzig bearbeiteten Forschungsprojektes zur Modellpartitionierung im Vorfeld der parallelen Logiksimulation [12] durchgeführt und baut auf den Resultaten eines Berufspraktikums auf [2]. Das Projekt wird von der DFG im Rahmen des Schwerpunktprogrammes „Effiziente Algorithmen für diskrete Probleme und ihre Anwendungen" gefördert.[7]

MEISTER teilt die Parallelisierungsansätze in der Logiksimulation in synchrone und asynchrone Schemata ein [1]. Bei dem, für *parallel*TEXSIM angewendeten, synchronen Simulationsschema verzichtet man auf die Einführung lokaler Zeiten für die Bearbeitung der einzelnen Modellblöcke.

Damit ein voller Takt bei einem Modellblock ohne Rückgriff auf die anderen simulierbar ist, muß die Schaltung so zerlegt werden, daß alle zur Evaluierung eines Schaltelementes während eines Taktschrittes beitragenden Bestandteile in diesem Block enthalten sind. Die zu *parallel*TEXSIM gehörenden Modellpartitionierungswerkzeuge [16] verwenden dafür *fan-in Cones* [8], in denen im obigen Sinne zusammengehörige Elemente gebündelt sind.

Ein derartiger Cone wird in Abbildung 5 verdeutlicht (schattiert).[8] Augenfällig ist dabei auch eine Einteilung der Schaltungselemente (sichtbar von links nach rechts) auf bestimmte Niveaus (Level), die meist als *Levelizing* bezeichnet wird. In Abschnitt 5 wird dies näher (und formaler) betrachtet. Insbesondere auch auf die Überlappung von Cones wird in [8] eingegangen, die Signalschnitte (vgl. Abschnitt 5.6) nach sich zieht, wenn zwei einander überlappende Cones jeweils auf verschiedene Blöcke verteilt werden.

Ein wesentlicher Vorzug der cone-basierten Modellpartitionierung besteht darin, daß der eigentliche Simulationsalgorithmus offensichtlich nicht modifiziert werden muß. Dies war in dieser Diplomarbeit das Grundprinzip und eine unvermeidliche Konsequenz der Praxisrelevanz von TEXSIM: Da dieser in ein sehr komplexes und leistungsfähiges Umfeld von VLSI-Design-Tools

[7] DFG-Aktenzeichen Sp 487/1-1

[8] M_I globale Eingänge, M_O globale Ausgänge, M_L Latche (siehe Abschnitt 5)

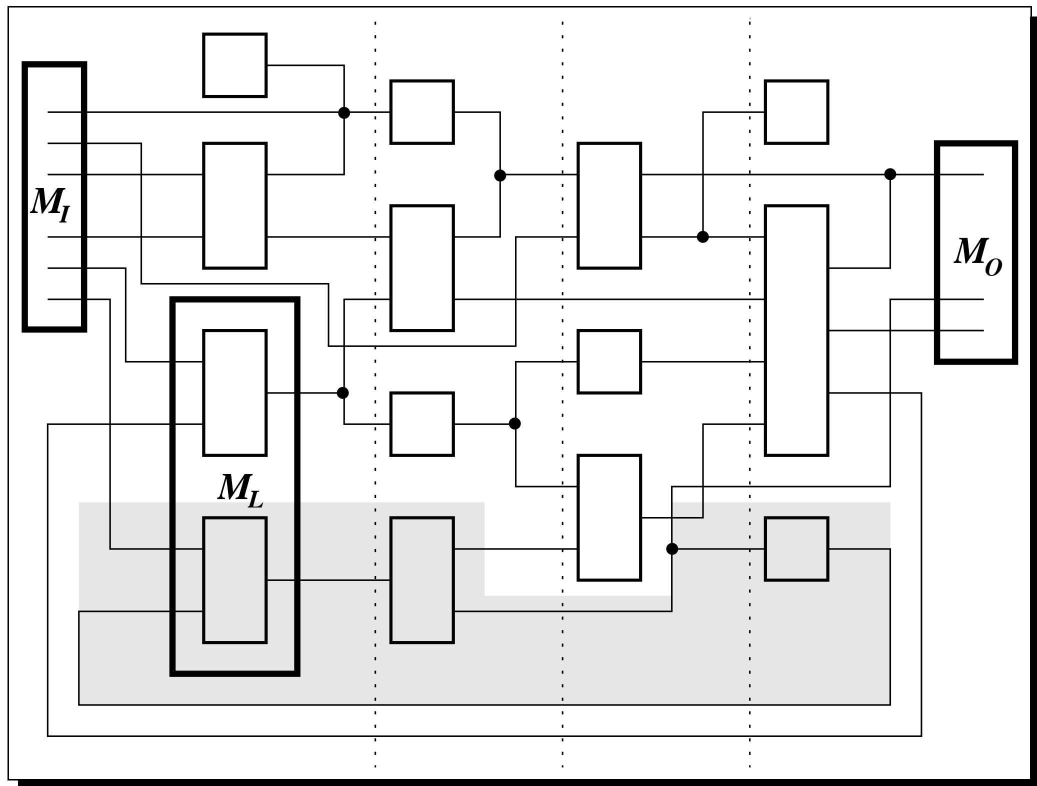


Abbildung 5: Cone in einer synchronen Schaltung

eingebunden ist (siehe Abschnitt 3), kann sein Algorithmus zur Modellevaluation nicht ohne unerwünschten Einfluß auf andere Programme geändert werden.

Nach [1] sind synchrone Parallelisierungsschemata für ereignisgesteuerte Simulatoren nicht geeignet. Bei einem asynchronen Schema sind die Ereignisse mit lokalen Zeitstempeln zu versehen. Die zur Sicherung der Korrektheit der Simulation damit verbundene Problematik der globalen Zeitstempelverwaltung über Synchronisationsprotokolle wird beispielsweise in [4, 17, 18] verdeutlicht.

3. Grundlagen

Im folgenden wird ausgehend von den allgemeineren Betrachtungen zu Simulationen ein konkretes Szenario der Logiksimulation aus der Praxis von IBM (und Motorola) skizziert. Dessen Kenntnis ist zum Verständnis von *parallel*TEXSIM unumgänglich, da dieser Simulator für dieses Umfeld konzipiert und darin entwickelt und getestet wurde. Eine besondere Rolle spielte dabei der Xmon, dem deshalb ein eigener Abschnitt gewidmet ist.

3.1. Der sequentielle Simulator TEXSIM

TEXSIM ist ein durch DAVID S. ZIKE (IBM Austin) entwickelter firmeninterner Logiksimulator, der von IBM und Motorola sehr erfolgreich bei der Entwicklung führender Mikroprozessorarchitekturen (POWER2, ESA/390 und PowerPC) eingesetzt wurde. Unter dem Namen MaxSim ist er 1994 auf der Design Automation Conference (DAC) in San Diego einer breiteren Öffentlichkeit vorgestellt und seitdem in älteren Versionen kommerziell vertrieben worden. In [2] sind Aufbau und Wirkungsweise des Programmes kurz skizziert.

Der Simulator ist, bis auf einige Zeilen Assemblercode mit denen die Cache-Lines einiger Prozessoren als ungültig deklariert werden, in ANSI-C weitgehend portabel programmiert, und es existieren Versionen für verschiedene Plattformen, die alle aus den gleichen Quellen erzeugt werden können. Unterschiede zwischen ihnen ergeben sich vor allem dadurch, daß TEXSIM einige betriebssystemspezifische Eigenschaften ausnutzt, die nicht überall vorhanden sind (z.B. ReXX Subcommand-Handler, dynamisches Linken). Als Simulationsverfahren dient der, durch die Einschränkung auf synchrone Logik in der Gate- und Registerebene (vgl. S. 8) einsetzbare *Clock-Cycle*-Algorithmus (siehe Abschnitt 5).

Es können gemischte Modelle (multi level) zweier IBM-interner struktureller[9] Hardwarebeschreibungssprachen, DSL/1 und BDL/S [6], sowie von (zu deren Funktionsumfang äquivalenten) Verilog- und VHDL-Subsets (auf Gate- und Register-Transfer-Ebene) simuliert werden. Die Sprachcompiler erzeugen dafür als Protos bezeichnete Zwischenstrukturen, die im Wesentlichen die Schaltung repräsentierende Graphen, als Netzwerke oder Netzliste bezeichnet, sind. Sie werden mittels eines Datenbankprogrammes, der Design Automation Database (DA_DB), verwaltet. Über einen auf die DA_DB aufgesetzten Modellcompiler, TEXas BuiLD (TEXBLD), sind aus ihnen dann die eigentlichen Simulationsmodelle zu bilden [16]. Diese enthalten eine Symbol-Tabelle (Namen, Typen, Größen, Wert-Offsets usw.), den Objekt-Code für die Kombinatorische Logik und den Objekt-Code für die Latche.

Abbildung 6 vermittelt einen Einblick in das komplexe Umfeld von Design Tools, in welches TEXSIM eingebettet ist. Dabei ist CORVETTE eine zu TEXSIM kompatible spezielle Simu-

[9] ZIKE charakterisiert in [7] strukturelle Hardwarebeschreibungssprachen im Gegensatz zu verhaltensbasierten als schematisch. Sie enthalten nur Konstrukte mit direkten Hardware-Implementationen wie Booleschen Gleichungen, Multiplexern und Latches und sind in diesem Sinne exakt beschreibend.

lationshardware (Hardware Accelerator) und CORVBLD der zugehörige Compiler. Mit dem BROWSER läßt sich ein Proto grafisch visualisieren. Über TEXSCOPE lassen sich die vom Simulator erzeugten All-Events-Traces (AETs) analysieren. Bei ETE handelt es sich um den Early Timing Estimator. RTPG, AVP und RTX werden im nächsten Abschnitt kurz betrachtet. CDESDB ist der auf die DA_DB aufgesetzte DSL/1-Compiler (von dem es auch eine Stand-Alone-Version gibt) und mit LDBDLS werden BDL/S-Codes in Protos konvertiert.

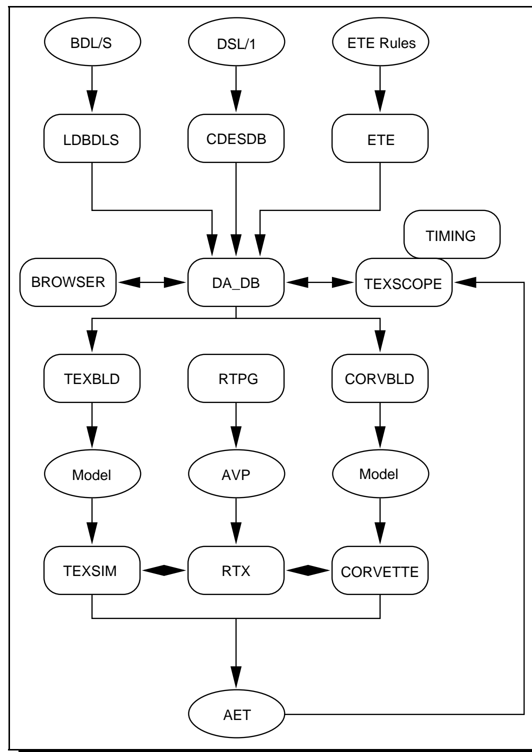


Abbildung 6: DA_DB und TEXSIM

Da TEXBLD den Schaltungselementen Maschinencodesequenzen[10] als Evaluierungsverfahren zuordnet, die für maximal zu erreichende Simulationsgeschwindigkeiten extensiven Gebrauch von Spezifika der zugrundeliegenden Prozessorarchitektur machen, ist es nur für ausgewählte Architekturen verfügbar: IBM POWER, RT, S/370 und S/390, HP PA-RISC und SUN SPARC.[11] Für die Portierung auf weitere Prozessorarchitekturen würde ZIKE nach eigener Aussage jeweils einen Monat Arbeitszeit benötigen.

Nach [7] setzt sich der Modellbildungsprozeß aus folgenden Arbeitsschritten zusammen:

1. Erzeugung eines Netzwerks in einer Datenbank durch den Sprachcompiler

[10] TEXBLD erzeugt direkt (ohne Hilfsprogramme) Objekt-Code.

[11] Da hierfür nur von der Integer-Einheit der Prozessoren Gebrauch gemacht wird, ist die Lauffähigkeit auf allen Versionen von SPARC-Prozessoren trotz deren teilweiser Inkompatibilität gewährleistet.

2. Flattening der Netzwerk-Hierarchie (vgl. Abschnitt 2.1)
3. Optimierung des Netzwerks (siehe dazu auch die Abschnitte über Logiksynthese in [6])
4. Einteilung des Netzwerks in verschiedene Stufen (Levelizing, siehe Abschnitt 5)
5. Strukturelle Analyse und Aufteilung der Logik
6. Erzeugung der Symbol-Tabelle und Wertbelegungen
7. Stufenweise Compilation der Logik:
   (a) Erzeugung des Referenzierungsverlaufs für die Register-Belegung
   (b) Erzeugung der maschinenunabhängigen Code-Sequenzen und anschließende „Peephole“-Optimierung
   (c) Anordnung der Code-Sequenzen (Scheduling)[12]
   (d) Erzeugung des maschinenabhängigen Objekt-Codes

3.2. Logiksimulation mit TEXSIM

Mit TEXSIM können während der Simulation sinnvollerweise nur bestimmte Schaltungselemente, die *Facilities*, kontrolliert, also abgefragt bzw. verändert werden. Dieses sind verschiedene Arten von Arrays [19, 6] und Schaltungssignale verkörpernde Netze [8] bzw. die Vektoren genannten Zusammenfassungen von Einzelnetzen, die durch einen entsprechenden Index charakterisiert sind. Nach außen hin unterscheidet der Simulator nicht zwischen diesen Typen, da sich diese durch eine geeignete Matrix aus *Rows* (Reihen) von Einzelbits darstellen lassen. Abbildung 7 zeigt ihren Aufbau; dabei ist $m \in \{0,\ldots,2^{64}-1\}$ und $n \in \{0,\ldots,2^{16}-1\}$.

Ein einzelnes Netz besteht immer aus einem Bit, d.h., es ist $m = 0$ und $n = 0$. Eine Ausnahme bilden die indizierten Netze, bei denen zwar $n \geq 0$, jedoch nur ein Bit gültig ist. Auch bei Vektoren ist $m = 0$, $n \geq 0$ und es müssen nur wenigstens zwei Bits gültig sein. Der Verantwortung des Nutzers obliegt es, nur auf gültige Bits zuzugreifen.[13] Im Gegensatz zu Netzen bzw. Vektoren ist für Arrays $m \geq 0$ zulässig und alle Bits sind gültig. Die Bits einer Facility können (etwa für Busstrukturen) auch mehr als die üblichen zwei Zustände aufweisen.[14] Dafür sind außer IBM-internen Konventionen Wertebelegungen nach IEEE 1164 zulässig. Neben den bislang genannten Facility-Typen kennt der Simulator noch assoziative und dünn besetze (sparse) Arrays. Bei ihnen kann auf die einzelnen Rows nur sequentiell über spezielle Funktionen zugegriffen werden.

[12] die Ausführungszeit von Codes ist stark von der Reihenfolge der einzelnen Anweisungen abhängig (Stichwort Branch-Prediction)

[13] Alle Facilities werden durch TEXSIM mit Null bzw. mit dem durch eine Startoption festgelegten Wert initialisiert. Daher weisen auch die ungültigen Bits diesen Wert auf und dürfen nicht verändert werden.

[14] Jedoch kann TEXSIM mehrwertige Zustände nicht propagieren.

Rows	Bits									
0	0	1	2	3	4	5	6	7	⋯	n
1	0	1	2	3	4	5	6	7	⋯	n
2	0	1	2	3	4	5	6	7	⋯	n
....										
m	0	1	2	3	4	5	6	7	⋯	n

Abbildung 7: Aufbau einer TEXSIM-Facility

Intern werden die Bits einer Facility auch durch tatsächliche Bits repräsentiert. Dahinter steht der Gedanke, Simulationen auf Register-Transfer-Level besonders effizient zu handhaben. Bei diesen wird nur selten auf einzelne Bits zugegriffen und statt dessen mit der gesamten Facility operiert. Das spiegelt sich auch im Aufbau der Schaltung wider. Auf der Gate-Ebene wiederum überwiegt die Verschaltung einzelner Bits. Deshalb sind in MVLSIM die Bits einer Facility durch einzelne Bytes dargestellt, damit der Zugriff (auch bei der eigentlichen Simulation) auf die einzelnen Facility-Bits wesentlich effizienter erfolgen kann.

Die anhaltende Bedeutung des Gate-Level ergibt sich daraus, daß es neben der Design-Fraktion, die mit Hardwarebeschreibungssprachen arbeitet, auch noch sehr viele Entwickler gibt, die mit grafischen Design-Tools arbeiten [6]. Deren Output ist eine Schaltungsbeschreibung auf Gate-Ebene. Darüber hinaus läßt sich bei der Arbeit mit den Hardwarebeschreibungssprachen der Trend weg vom Register-Transfer-Level hin zur funktionellen Ebene (Behavior) erkennen. Verursacht wird dieser durch die ständig zunehmende Komplexität der Entwicklung.

Vom Functional-Level wird die Schaltung über Synthese-Tools automatisch direkt in den Gate-Level überführt. Per Definition ist damit eine auf dem Functional-Level korrekte Schaltung auch auf dem Gate-Level korrekt. Die Simulation auf der funktionellen Ebene ist aber auf Grund der geringeren Komplexität wesentlich schneller. Jedoch müssen Synthese-Programme (wie das von Synopsys) über eine große Menge an „Intelligenz“ verfügen, weshalb sie erst jetzt praxisreif werden. Somit hat für die unvermeidliche gemischte Simulation von Gate- und Functional-Level ein für die Gate-Ebene optimierter Simulator durchaus seine Berechtigung.

Ausgehend von [2] wird in Abschnitt 3.3 der Einsatz von TEXSIM insbesondere am Beispiel eines Böblinger Monitors (Xmon) knapp umrissen. Demnach läßt sich ein praxisrelevanter Simulationslauf durch das sogenannte ALTER-CLOCK-RETRIEVE Schema aus Abbildung 8 beschreiben, welches typischerweise viele Male abgearbeitet wird. Die dabei auftretenden Arbeitsschritte werden in Tabelle 2 angegeben. Die in [6] geschilderten Einsatzszenarien der Logiksimulation in der Industriepraxis besitzen in ihrer Allgemeinheit auch für die Verwendung von TEXSIM Gültigkeit (siehe auch Abbildung 2).

Zur Überprüfung der Korrektheit der Schaltungsentwürfe dienen die *Test Cases*. Aufgrund seines Haupteinsatzgebietes im Prozessordesign sind Test Cases für TEXSIM im Gegensatz etwa zum Logiksimulator vsim aus [4, 17, 18] keine Stimulivektoren aus Signalbelegungen sondern

Maschinen- oder Microcode-Sequenzen. Allgemein besteht ein Test aus drei Teilen: den initialen Belegungen für Register, Speicher usw., den Test Cases und den Testende-Bedingungen für Register, Speicher usw. Da oftmals bereits existierende Prozessorfamilien weiterentwickelt werden, können die Resultate der Tests auf den realen Prozessoren mit denen der simulierten verglichen werden. So ist es auch auf einfache Weise möglich, Test Cases automatisch zufällig zu erzeugen.

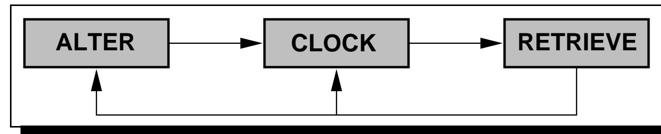


Abbildung 8: Das ALTER-CLOCK-RETRIEVE Schema

ALTER	Setzen von Facilities auf bestimmte Werte
CLOCK	Simulation einer Anzahl von Schaltungszyklen
RETRIEVE	Wertabfrage von Facilities

Tabelle 2: ALTER-CLOCK-RETRIEVE

Wegen des sehr ungünstigen Verhältnisses von simulierter zu realer CPU-Zeit (siehe S. 5) sind der Komplexität der Tests aber schnell Grenzen gesetzt. Deshalb war es auch ein Ziel der Parallelisierung, umfangreichere Tests wie z.B. Teile eines Bootvorganges simulieren zu können. Gegenwärtig dominiert in der Praxis die Simulation sehr vieler kleiner Test Cases.

Bei IBM Austin und im PowerPC-Entwicklungszentrum von IBM, Apple und Motorola in Somerset etwa werden permanent Test Cases automatisch zufällig erzeugt [20, 19]. Dies geschieht mittels des Programmes RTPG (Random Test Program Generator), dem die Eigenschaften der Prozessoren in Form der Design-Spezifikationen „bekannt“ sind, so daß kein Vergleich mit Resultaten auf realen Prozessoren notwendig ist [20]. Die erzeugten Test Cases, AVPs (Architectural Verification Programs) genannt, werden automatisch auf einzelne Rechner eines sehr großen Clusters ($\approx$ 500 Maschinen) verteilt und dort mittels des speziellen Simulationskontrollprogrammes RTX (Random Test Executor) auf ununterbrochen laufenden TEXSIM-Instanzen simuliert (Abbildung 6). Werden bei einem Test Fehler entdeckt, so erhält ein entsprechender Entwickler selbsttätig eine elektronische Mitteilung (eMail) darüber.

Zur Überwachung und Steuerung der Modellsimulation stellt TEXSIM zwei Schnittstellen bereit: einen interaktiven Kommandoprozessor (Anhang C) und ein Programmier-Interface (Anhang A). Sie bieten eine Vielzahl von Funktionen für das ALTER-CLOCK-RETRIEVE Schema und für die Erzeugung von verschiedenen Arten von Ausgabedateien. In der Praxis wird aber aufgrund der Komplexität der Tests fast ausschließlich die Programmierschnittstelle verwendet.

Dies geschieht mittels spezieller Programme, den *User Programs*, die entweder Stapeldateien (Batches) von Anweisungen des interaktiven Kommandoprozessors, ReXX-Programme oder Module in Maschinensprache sind. Letztere werden zur Laufzeit dynamisch an den Simulator angelinkt (run-time linking) und ausgeführt. Damit dies möglich ist, definiert TEXSIM ein API (Application Programming Interface), welches, wie im Unix-Bereich üblich, den Aufrufkonventionen der Sprache C folgt [21].

Die Simulation geschieht somit gemäß eines Client-Server-Konzeptes mit den User Programs als Clienten und TEXSIM als Server (Abbildung 9). Die Clienten lassen sich hinsichtlich der Art und Weise ihrer Abarbeitung durch den Server nach den Kategorien *Unmanaged Client, Managed Client* und *User Exits* klassifizieren [9]:

- **Unmanaged Clients**
  TEXSIM verfügt über einen allgemeinen Mechanismus zum Laden und Ausführen von Programmen. Während der Simulation können damit beliebige Module von Programmen angelinkt und abgearbeitet werden, ohne daß TEXSIM davon besondere Kenntnis erhalten muß. Mit dem Mechanismus wird das mehrfache Linken gleicher Module verhindert und die Möglichkeit gegeben, den Linkvorgang wieder rückgängig zu machen. Auf diese Weise geladene Module heißen *Unmanaged Clients*.

- **Managed Clients**
  Im Gegensatz zu den Unmanaged Clients wird der Start und das Ende von *Managed Clients* von TEXSIM kontrolliert: Über einen speziellen Befehl wird der Client geladen und unmittelbar ausgeführt. Im Rahmen der Beendigung von TEXSIM wird er automatisch terminiert. Durch ihn können weitere Managed und Unmanaged Clients gestartet werden.

- **User Exits**
  Sie werden vom Simulator zu bestimmten Zeitpunkten abgearbeitet und lassen sich in *CYCLEXITs*, *RESETEXITs* und *STOPEXITs* unterteilen. CYCLEXIT-Programme werden nach einem Modelltakt ausgeführt, wobei sich allerdings der Starttakt, das Taktintervall und der Stoptakt der Ausführung festlegen lassen. Ein RESETEXIT wird entweder zum Beginn eines Simulatorresets vor dem Zurücksetzen der Facility-Werte oder direkt nach dem Reset aufgerufen. Die STOPEXIT-Programme werden dagegen nur zum Simulationsende als Teil des Beendigungsprozesses ausgeführt.

Der gesamte Vorgang des Simulationsprozesses besteht nach [7] aus folgenden Schritten:

1. Laden und Initialisieren des Modells
2. Laden des Simulationssteuerprogrammes
3. Abarbeitung des Steuerprogrammes
   (a) Laden des Test Case

(b) Setzen der initialen Modellbelegungen[15]

(c) Start der Simulation

(d) Überwachung des Testendes

(e) Bewertung der Testresultate

(f) Simulationsende oder 3a

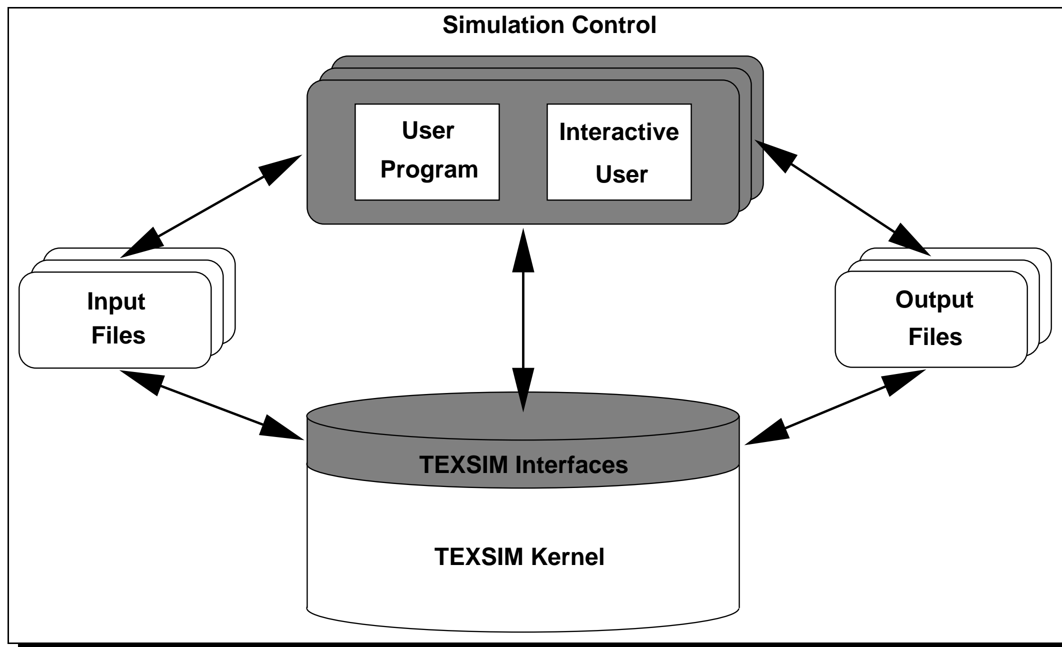


Abbildung 9: Das TEXSIM Client-Server-Prinzip

Der C-Pseudo-Code von DAVID S. ZIKE in Abbildung 12 gibt den groben Rahmen des Programmablaufs von TEXSIM an. Demnach ist die strikte Einhaltung der Reihenfolge des ALTER-CLOCK-RETRIEVE Schemas nicht unbedingt erforderlich; alle Facility-Werte sind zu jedem Zeitpunkt der Programmausführung gültig.[16] Dies wird dadurch erreicht, daß der Simulator die kombinatorische Logik als ungültig deklariert (`logic_is_valid = 0`), falls in eine Logik-Facility geschrieben wird (`putfac`). Sollen dann Werte einer Logik-Facility ausgelesen werden (`getfac`), so erfolgt eine komplette Evaluation der kombinatorischen Logik (bis an die zyklusbegrenzenden Latche; `update_logic`). Für Latches ist dies wegen ihrer um einen Takt verzögerten Wirkung nicht nötig. Die Modellzeit (`current_cycle`) kann mittels eines `clock`-Befehls um eine Anzahl von Takten erhöht werden. Dieser evaluiert die kombi-

[15] Das Laden des Test Case kann als Bestandteil dieses Schrittes betrachtet werden, ist aber aufgrund seiner herausragenden Bedeutung separat aufgeführt.

[16] außer wenn man dies durch eine Option beim Start unterbindet

natorische Logik nur, falls sie ungültig ist, und vermeidet damit unnötige Mehrfachevaluationen. Bei einer Verletzung des ALTER-CLOCK-RETRIEVE Schemas (`putfac clock getfac`), läßt sich das aber nicht verhindern. Die Ausnahme davon bildet natürlich die Vorgehensweise aus Abbildung 10.

```
for (;;)
   {
   do_putfacs ()
   do_getfacs ()
   clock ()
   }
```

Abbildung 10: Grundablauf eines TEXSIM-User-Programs

Zur Identifikation von Facilities erfordern viele API-Funktionen Handles, die *Facility Indizes*. Ein solches Handle läßt sich mittels einer speziellen Funktion (`efsrtsym` bzw. `symbol`) aus dem Namen der Facility ermitteln. Da sich mit einigen Funktionen auch einzelne Teile einer Facility kontrollieren lassen, verlangen diese eine sogenannte Facility-Referenz als Parameter. Dabei handelt es sich um einen Zeiger auf einen der C-Struktur aus Abbildung 11 entsprechenden Speicherbereich, der die betreffenden Facility-Bits eindeutig kennzeichnet. Da TEXSIM für 32-Bit Unix-Syteme (vor allem IBM AIX/6000) entwickelt wurde, werden zwei Schnittstellen für die Facility-Werte angeboten: das Integer- und das Charakter-Interface.

```
struct facref
{
  FACIDX       s;                    /* facility index        */
  char         *facname;             /* ptr to facility name  */
  int          offset;               /* offset of reference   */
  int          length;               /* length of reference   */
  unsigned int row;                  /* row number for arrays */
  unsigned int row_high;             /* row number for arrays */
};
typedef struct facref FACREF;
```

Abbildung 11: C-Struktur Facility-Referenz

Während sich mit den Funktionen des ersteren nur bis zu 32 Bits einer Row auf einmal kontrollieren lassen, können die des Character-Interfaces mehr als 32 Bits als C-String binär, dezimal und hexadezimal mit verschiedenen Werte-Mengen (z.B. nach IEEE 1164) erfassen. Mit dem

Integer-Interface können über mehrere Facility-Referenzen selbstverständlich zwar alle Bits focussiert werden, jedoch ist mehrwertige Logik so nicht umfassend behandelbar. Doch auch beim Charakter-Interface können sich bei entsprechend großen Facilities mehrfache Referenzen nicht vermeiden lassen.

3.3. Der TEXSIM-Client Xmon

Xmon ist die Abkürzung von Xmonitor. Bei diesem Programm handelt es sich um ein, durch das IBM Entwicklungslabor Böblingen genutztes, visuelles Interface für TEXSIM, das auch eine eigene Programmierschnittstelle besitzt. Als Nachfolger für den Monitor des durch TEXSIM ersetzten Logiksimulators aus [6] wird er hauptsächlich bei der Entwicklung von ESA/390-Mikroprozessoren verwendet, ist aber prinzipiell für das Design beliebiger Prozessorarchitekturen nutzbar.

In der Praxis findet die Steuerung der Simulation von Mikroprozessormodellen aufgrund der Komplexität der damit verbundenen Test Cases nicht auf der Ebene von reinen TEXSIM-User-Programmen statt, sondern mittels sogenannter *Monitore*. Diese Monitore sind spezifische Entwicklungen einzelner IBM-Lokationen und werden zumeist auch nur von ihnen eingesetzt, da sich in einem Monitor das jeweilige spezielle Anwendungsprofil, also die Prozessorarchitektur, deutlich widerspiegelt. Monitore sind genau auf die Funktionalität, die die entsprechende Designpraxis erfordert, zugeschnitten. Dadurch sind sie für den Entwickler besonders effizient verwendbar. Ein weiteres Beispiel für ein solches Programm ist RTX (siehe S. 16).

Konkret ist Xmon ein Managed Client von TEXSIM, also als eine Erweiterung des Simulators zu betrachten. Jedoch kann man ihn wegen seiner großen Funktionsvielfalt auch als relativ eigenständiges Programm ansehen. Sein auf die ESA/390-Architektur abgestimmter Leistungsumfang beinhaltet unter anderem Funktionen zur Modellinitialisierung, vordefinierte Formate für Signalwertverläufe (Traces) und Test Cases (Micro- und Maschinencode), ein konfigurierbares Graphical User Interface (GUI) auf Motif-Basis, einen Maschinencode-Disassembler, die Alias-Bildung für Namen von Modellbestandteilen und xmReXX, eine eigene Erweiterung der Sprache ReXX.

Darüber hinaus ist sein besonderes Charakteristikum die standardmäßige Unterstützung von Modellen aus mehreren Prozessoren (MP-Modelle), eine etwa für SMP-Rechner (z.B. typische ESA/390-Computer) sehr sinnvolle Eigenschaft. Da die konkrete Modellbeeinflussung letztlich über das TEXSIM-API erfolgen muß, auch wenn dies der Monitor selber übernimmt, führt das zu einem hierarchischen Namenskonzept für die Modellfacilities, welches schon durch die Logikdesigner festzulegen ist. Beispielsweise `P0, P1, ...` für die verschiedenen Prozessoren, `P0.EU, P1.CS` für Bestandteile von Prozessoren und etwa `PIC` für den gesamten Chipsatz, also dann `PIC.P0.EU, PIC.P1.CS` usw. Um nicht immer mit dem kompletten Namen arbeiten zu müssen, kann man sogenannte *Domains* festlegen, auf die sich ein Namen dann bezieht (etwa `PIC`, so daß `P0.CS` gültig ist). Es ist `P0` eine sogenannte *Subarea* von `PIC` und `CS` eine von `P0` (Abbildung 3).

```
        load_and_initialize_model ();
        logic_is_valid = 0;
        current_cycle  = 0;
        call_control_program_and_wait_for_events ();
        for (;;)
          {
          switch (event)
            {
            case getfac:
              if (type != ARRAY &&
                  type != LATCH &&
                  logic_is_valid == 0)
                {
                update_logic ();
                logic_is_valid = 1;
                }
              return get_value ();

            case putfac:
              if (type != ARRAY)
                logic_is_valid = 0;
              set_value ();
              break;

            case clock:
              call_cycle_exit_programs ();
              apply_queued_putfacs ();
              if (logic_is_valid == 0)
                update_logic ();
              update_read_before_write_arrays ();
              execute_assert_statements ();
              write_all_events_trace ();
              check_buses ();
              update_latches ();
              logic_is_valid = 0;
              current_cycle++;
            }
          }
```

Abbildung 12: Der Programmablauf von TEXSIM

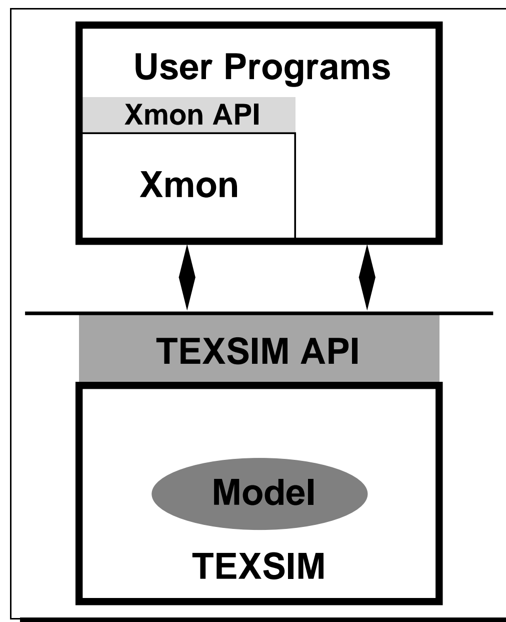


Abbildung 13: Einordnung des Xmon API

Neben seinem eigenen C-API namens *Monitor Model Interface* (mmi) stellt Xmon zudem noch das darunterliegende TEXSIM-API zur Verfügung (siehe Abbildung 13 und [22]). Genau wie TEXSIM verfügt er, neben dem weiterhin vorhandenen TEXSIM Subcommand-Interface, über ein eigenes ReXX Subcommand-Interface, auf das wie bei TEXSIM auch mittels Stapeldateien oder über den interaktiven Kommandoprozessor zugegriffen werden kann. Der Kommandoprozessor ist in das GUI integriert. Außerdem verwendet der Monitor eine Vielzahl von Konfigurationsfiles, z.B. für das User-Interface, die Festlegung abzuarbeitender Test Cases, die Modellinitialisierung und die Formatierungen der Traces.

Wichtig für den Simulationsablauf sind die *Break-Routinen* von Xmon, die nach *jedem* Schaltungszyklus (CLOCK) abgearbeitet werden und festzulegende Facilities abfragen (RETRIEVE), um so etwa Wertverläufe zu protokollieren oder Abbruchkriterien zu überprüfen. Vor allem ergibt sich hierbei das typische Ablaufschema CLOCK-RETRIEVE (Abbildung 8), welches *ständig* durchlaufen wird!

Durch sein Gesamtkonzept und sein GUI ist Xmon besonders für interaktive Simulationen geeignet, bei denen der Logikdesigner aktiv eingreifen kann. Für interaktive Simulationen lohnt sich der Einsatz von *parallel*TEXSIM aufgrund des Partitionierungsoverheads und der Eigenheiten des ALTER-CLOCK-RETRIEVE Schemas gewiß nicht. Im Rahmen der Parallelisierung war die Kombination beider Programme jedoch sehr hilfreich. Der Monitor läßt sich aber auch für vollautomatische Läufe (analog zu RTX) verwenden.

3.4. Voraussetzungen

Diese Diplomarbeit knüpft unmittelbar an ein Berufspraktikum an (siehe S. 10), in dem wichtige Voraussetzungen für die TEXSIM-Parallelisierung geschaffen worden. Die zugehörige Dokumentation [2] erläutert unter anderem, warum C als Implementierungssprache gewählt wurde. Weiterhin wird die Wahl des AIX Parallel Environment (PE) und der Message Passing Library als Parallelisierungsgrundlage und der Verzicht auf Multithreading begründet.

Seitdem der Autor Zugang zu einer neueren Version des PE hat, zu dem auch eine stark verbesserte Dokumentation gehört, ist klar geworden, daß mit PE Multithreading bislang nicht möglich ist. Die Ursache dafür ist die Verwendung nicht threadsicherer Bibliotheken. Dem dürfte aber bald Abhilfe geschaffen werden, weil IBM für die RS/6000 SP inzwischen auch SMP-Rechner auf PowerPC-Basis als Knoten anbietet.

Ein anderer wichtiger Punkt, der in [2] behandelt wird, ist die Programmierpraxis mit ihrer vielfältigen Problematik. So wird die eingesetzte Entwicklungsplattform hinsichtlich Hard- und Software und des Umgangs damit betrachtet.

Wie in Abschnitt 2.3 erwähnt wurde, ist die Beibehaltung des eigentlichen Simulationsalgorithmus eine wesentliche Voraussetzung für die Parallelisierung. Im Rahmen des Praktikums wurde die Durchführbarkeit dieser Methode abgesichert.

Die eigentliche Parallelisierung von TEXSIM mußte, abgesehen von wenigen Ausnahmen, vollkommen selbständig vom Diplomanden vorgenommen werden, da dessen Schöpfer DAVID S. ZIKE in der Regel nur per eMail erreichbar und vielbeschäftigt war.[17] Der Source-Code ist kaum kommentiert und nicht dokumentiert, so daß nur der bloße Code die Ausgangsbasis darstellte.

[17] Während sich der Erfahrungsaustausch per eMail bei Projekten wie beispielsweise Linux bewährt hat, gab es hier das Problem, daß führende Entwickler wie ZIKE täglich dutzende berufliche Mails von den verschiedensten Personen erhalten.

4. Der Logiksimulator *parallel*TEXSIM

4.1. Aufbau

Im Gegensatz zu TEXSIM besteht die parallelisierte Variante aus mehreren, gleichzeitig ablaufenden und miteinander kooperierenden Programmen (Abbildung 14), deren Start über ein spezielles Shellskript erfolgt. Diese Aufteilung spiegelt das für die Parallelisierung gewählte Programmiermodell Master-Slaves wider. In Tabelle 3 sind die einzelnen Komponenten aufgeführt. Der Aufbau von *parallel*TEXSIM entpricht damit genau dem Design-Pattern Master-Slaves aus [23].

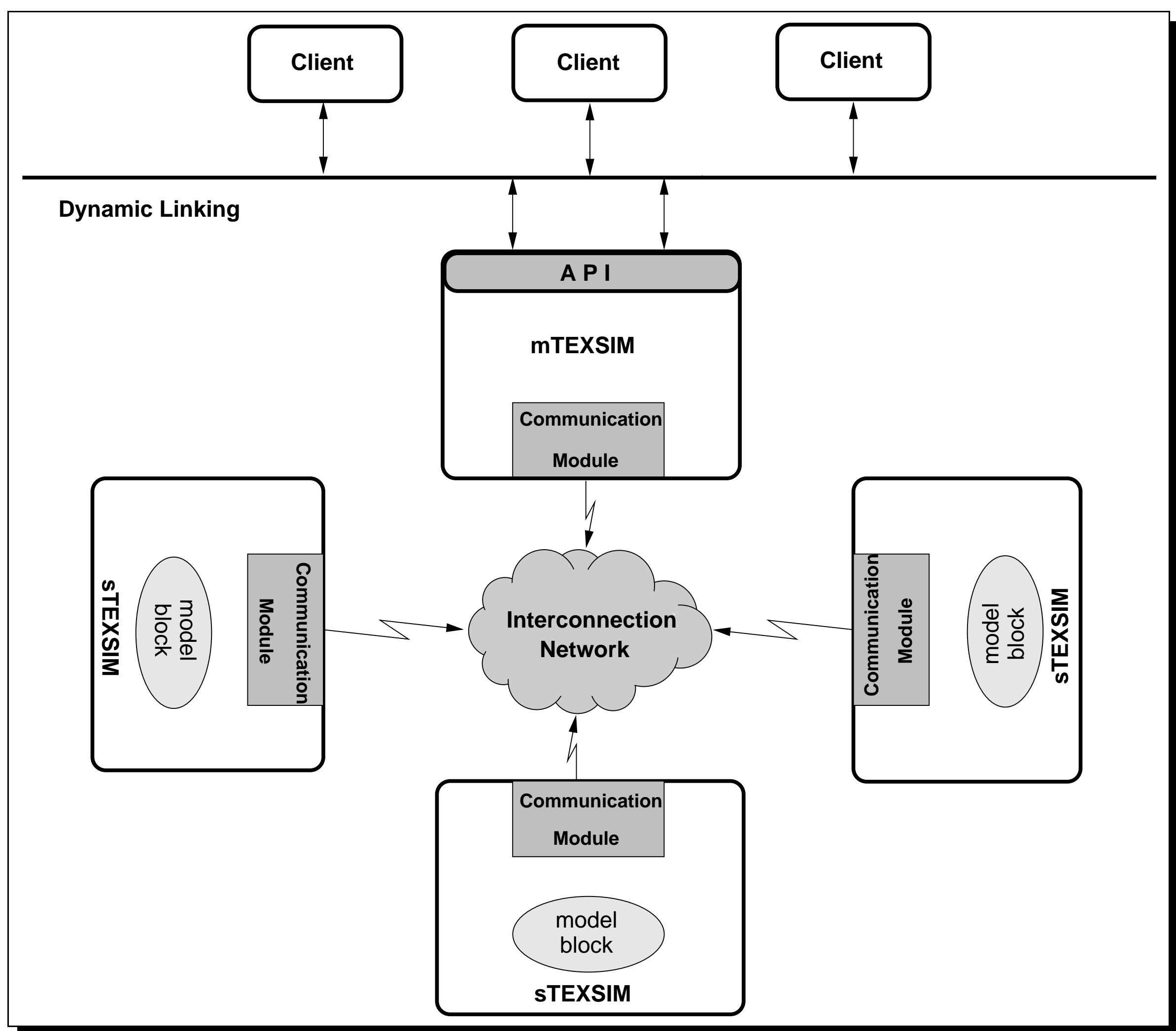


Abbildung 14: Simulation mit *parallel*TEXSIM

Programm	Name	Funktion	Datei
pTEXSIM	parallel TEXSIM	Start	`ptexsim`
mTEXSIM	master TEXSIM	Master	`mtexsim`
sTEXSIM	slave TEXSIM	Slave	`stexsim`

Tabelle 3: Teilprogramme von *parallel*TEXSIM

Die hier zu betrachtende Implementierung basiert auf dem IBM PE (AIX Parallel Environment [24]) mit der MPL (Message Passing Library) als Kommunikationsbibliothek [2]. Dabei wurde jedoch Wert auf leichte Portierbarkeit auf andere parallele Plattformen gelegt. Prinzipiell sind dabei solche verwendbar, die auf Prozessoren und Betriebssystemen basieren, für die TEXSIM (und damit TEXBLD) verfügbar ist.

Gegenwärtig können Workstation-Cluster aus RS/6000-Knoten sowie der IBM-Parallelrechner RS/6000 SP der Programmausführung unter dem Betriebssystem AIX dienen. Bei einem Workstation-Cluster ist eine Verbindung mit dem TCP/IP-Protokoll über einen Ethernet-, Token Ring- oder ATM-Anschluß die Kommunikationsgrundlage (Interconnection Network). Auf der RS/6000 SP gibt es zusätzlich die Möglichkeit einer Hochgeschwindigkeitsverbindung (hohe Bandbreite und geringe Latenzzeit) über verschiedene Entwicklungsstufen des sogenannten *High Performance Switch*. Es handelt sich dabei um ein mit *Wormhole*-Routing arbeitendes Permutationsnetzwerk [25].

Als Entwicklungsplattform diente ein Workstation-Cluster, der in [2] konkret beschrieben ist. Praxisrelevante Simulationen unterschiedlicher Art konnten im IBM-Entwicklungslabor Böblingen auf einer RS/6000 SP mit einem High Performance Switch der 2. Generation (SP2) durchgeführt werden, deren POWER2-Knoten mit jeweils 2 GByte physikalischem RAM ausgestattet waren (siehe Anhang D).

Der Programmablauf von Master und Slaves läßt sich sehr grob in eine Initialisierungs- und eine Aktivitätsphase einteilen, wobei letztere mit einem gemeinsamen Statusprotokoll beginnt. Der gesamte Informationsaustausch zwischen den einzelnen Tasks erfolgt über Nachrichten (Messages), d.h., es wird ein reines Message Passing System (ohne Shared Memory oder ählichem) verwendet. Das PE stellt dabei sicher, daß keine Nachrichten verloren gehen.

Die Hauptaufgabe des Masters mTEXSIM besteht darin, die Schnittstellen von TEXSIM bereitzustellen (also die Server-Funktion für die Clients zu übernehmen) und davon ausgehend die Slaves zu koordinieren. Diese simulieren die ihnen zugeordneten Modellblöcke (vgl. S. 10 und den nächsten Abschnitt) mittels des sequentiellen Clock-Cycle-Algorithmus.

Für den Nachrichtenaustausch müssen die Tasks identifizierbar sein, weshalb PE an sie Task Identifiers (Task IDs) vergibt. Dieses sind Nummern von 0 bis $numtask - 1$, wobei $numtask$ die Anzahl aller Tasks ist. Daher war es naheliegend, die Modellblöcke von dem Slave bearbeiten zu lassen, dessen Task ID gleich der Blocknummer (Block ID) ist. Der Master ist dann Task 0.

```
⟨model file⟩  ::=  ⟨model name⟩.⟨extension⟩
⟨extension⟩   ::=  ⟨target⟩⟨model type⟩
⟨target⟩      ::=  sj | sp | hp | vm | ...
⟨model type⟩  ::=  mod | opt
```

Abbildung 15: Namenskonvention für das TEXSIM-Modell

4.2. Das parallele Simulationsmodell

Die sequentiellen Modelle für TEXSIM enthalten eine Symbol-Tabelle (Namen, Typen, Größen, Wert-Offsets usw.) und den Objekt-Code für die Latche und die kombinatorische Logik komplett in einer einzigen Datei. Für die Simulation können von TEXBLD am Netzwerk diverse Optimierungen vorgenommen werden. Da sich solche optimierten Modelle sehr stark von den entsprechenden nicht optimierten unterscheiden können, werden diese über eine eigene Datei-Extension (`opt` gegenüber `mod`) ausgezeichnet. Weil sich für die einzelnen TEXSIM-Einsatzplattformen die Objekt-Codes unterscheiden, äußert sich das auch in den entsprechenden Dateinamen (`sj`[18] für RS/6000, `vm` für VM, `hp` für HP PA-RISC und `sp` für SUN SPARC) der jeweiligen Modelle (Abbildung 15). Solche Namen sind beispielsweise `BLP.sjmod` oder `BLP.vmopt`.

```
⟨model block 1⟩  ::=  ⟨model name⟩.1.⟨extension⟩
       ⋮                         ⋮
⟨model block n⟩  ::=  ⟨model name⟩.n.⟨extension⟩
```

Abbildung 16: Namenskonvention für die Modellblöcke

Für *parallel*TEXSIM ist durch den Partitionierer [16] ein paralleles (partitioniertes) Modell bereitzustellen. Dieses besteht aus mehreren Dateien: den *Modellblöcken*, den *Signalschnitt-Listen* und einer *Cross-Referenz-Liste*. Die Modellblöcke sind sequentielle TEXSIM-Modelle, die aus bei der Partitionierung geschaffenen Blöcken des zu simulierenden Modells erzeugt werden. Die Anzahl dieser Blöcke ist dabei aufgrund von praktischen Erwägungen (verfügbare Hardware usw.) vorgebbar. Die Modellblöcke müssen zur eindeutigen Identifizierung mit Filenamen nach der Konvention von Abbildung 16 (unter Ausnutzung des *Block Identifiers*[19]) versehen sein.

[18] San Jacinto war während der Entwicklung der Codename für das spätere RS/6000.

[19] Bei einer Partitionierung in n Blöcke werden diese von 1 bis n durchnumeriert. Die einem Block zugeordnete Nummer ist sein Identifier (ID).

Für die Listen-Files gibt es zwei verschiedene Formate: Textfiles (ASCII) und binäre Dateien. Die Textformate von Cross-Referenz-Liste (⟨model name⟩.pxref) und Signalschnitt-Listen (⟨model name⟩.⟨block id⟩.ext) werden in [2] ausführlich erläutert und in Anhang B kurz skizziert. Die binären Listen enthalten die Speicherstrukturen, die bei der Verarbeitung der Text-Listen aufgebaut werden (Tabellen 4 und 5). Statt pxref bzw. ext tragen sie in Analogie zu den Konventionen bei den Modellblöcken die Extension pmod.

Die Text-Files sind dabei als allgemeine Schnittstelle für beliebige als Partitionierer fungierende Programme anzusehen.[20] Generell sollte ein derartiges Programm aber die Erzeugung von pmod-Files für Master und Slaves vorziehen und pxref–Files bzw. ext-Files nur für Entwicklungs- und Debug-Zwecke einsetzen.

Alle Dateien des parallelen Simulationsmodells können komprimiert sein. Zur Kompression bzw. Dekompression dient das Public Domain Programm GNU gzip. Komprimierte Dateien werden durch dieses um die Extension gz erweitert, so daß sie unmittelbar als solche identifizierbar sind.

Der Aufbau und der Inhalt der einzelnen Dateien des parallelen Simulationsmodells ist durch die (nicht disjunkte) Einteilung der TEXSIM-Facilities in *globale, lokale* und *parallele Facilities* sowie in *geschnittene Netze* bedingt. Bei lokalen Facilities handelt es sich um, lokal in den jeweiligen Modellblöcken enthaltene, normale TEXSIM-Facilities. Die globalen Facilities entsprechen den im zugrundeliegenden sequentiellen Modell enthaltenen TEXSIM-Facilities. Den parallelen Facilities entspricht die Aufteilung der globalen Facilities in lokale (Abbildung 17).

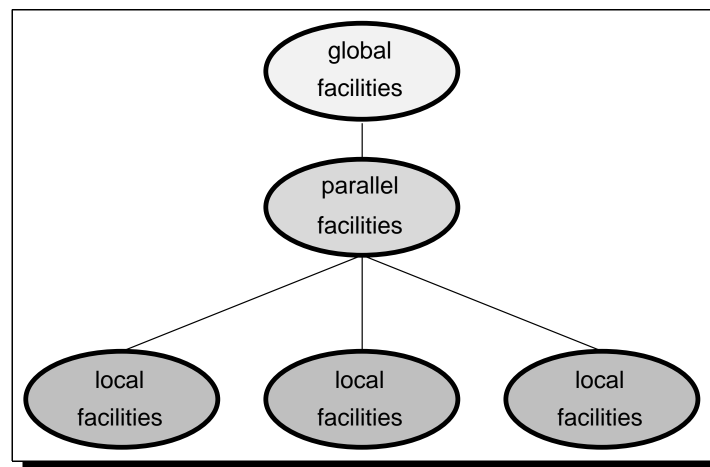


Abbildung 17: Facility-Hierarchie

[20] So ist es denkbar, daß für zu simulierende SMP-Prozessoren die einzelnen Teilprozessoren auf Modellblöcke eines parallelen Modells verteilt werden. Dafür ist eine entsprechende Kenntnis des Prozessors notwendig, so daß hierfür ein allgemeiner Partitionierer vermutlich nicht sinnvoll anwendbar ist.

Filestruktur

Offset	Typ	Bedeutung
		Header
0	U8	Erkennungssignatur für das Fileformat
1	U8	Version des Formats
2	8 × U8	Erzeugungsuhrzeit des Modellblocks
10	8 × U8	Erzeugungsdatum des Modellblocks
18	3 × U32	Erzeugungszeit der Signalschnitt-Liste
30	3 × U32	Erzeugungsdatum der Signalschnitt-Liste
42	U32	Anzahl der Inputs (ip)
46	U32	Anzahl der Outputs (op)
50	U32	Gesamtzahl der Blöcke (nt)
54	U32	Anzahl der globalen Facilities (num)
		Body
58	ip × PARFAC	Daten der Input-Netze
	op × PARFAC	Daten der Output-Netze
	nt × COMFAC	Zuordnung der Input-Netze zu Blöcken
	nt × COMFAC	Zuordnung der Output-Netze zu Blöcken
	num × U32	Lokale Facility Indizes der globalen Facilities

PARFAC

Offset	Typ	Bedeutung
0	U32	Facility Index (Offset)
4	S8	Netz-Index
5	U8	Anzahl der verbundenen Blöcke (num)
6	num × U8	Block IDs

COMFAC

Offset	Typ	Bedeutung
0	U32	Anzahl der Inputs/Outputs (num)
4	num × U32	Offsets der Inputs/Outputs

U8 Unsigned Byte | S8 Signed Byte | U32 Unsigned Word (32 Bits)

Tabelle 4: Format der pmod-Files für die Slaves

Filestruktur		
Offset	Typ	Bedeutung
		Header
0	U8	Erkennungssignatur für das Fileformat
1	U8	Version des Formats
2	8 × U8	Erzeugungsuhrzeit der Cross-Referenz-Liste
10	8 × U8	Erzeugungsdatum der Cross-Referenz-Liste
18	U32	Anzahl der globalen Facilities (fnum)
22	U32	Anzahl der parallelen Facilities (pnum)
		Body
26	fnum × STRING	Namen der globalen Facilities
	fnum × SYMBOL	Daten der globalen Facilities
	pnum × PARFAC	Daten der parallelen Facilities

SYMBOL		
Offset	Typ	Bedeutung
0	U32	Anzahl der parallelen Facilities (num)
4	U32	Anzahl der Rows der globalen Facility (Rows)
8	U32	Länge der globalen Facility (Length)
12	num × U32	Offsets der parallelen Facilities

PARFAC		
Offset	Typ	Bedeutung
0	S8	Netz-Index
1	U8	Anzahl der verbundenen Blöcke (num)
2	U32	Offset der zugehörigen globalen Facility
6	num × U8	Block IDs

STRING		
Offset	Typ	Bedeutung
0	U32	Länge der Zeichenkette (len)
4	len × U8	nullterminierte Zeichenkette

U8	Unsigned Byte	S8	Signed Byte	U32	Unsigned Word (32 Bits)

Tabelle 5: Format der `pmod`-Files für den Master

Durch die Partitionierung können einzelne Elemente eines Vektors auf verschiedene Blöcke verteilt und Modellbestandteile repliziert werden. Deshalb wird einer globalen Facility konkret eine Liste paralleler Facilities zugeordnet, denen jeweils wiederum pro Modellblock maximal eine lokale Facility zugeordnet ist.[21]

Geschnittene Netze sind durch die Partitionierung entstandene globale Schaltungsein- bzw. ausgänge der Modellblöcke, die in der ursprünglichen Schaltung mit dem jeweiligen globalen Eingang bzw. Ausgang eines anderen Blockes verbunden sind.

Die Reihenfolge der globalen Facilities ist in der Cross-Referenz-Liste und den `pmod`-Files identisch, um so über eine globale Ordnung verfügen zu können. Auf Grund ihrer durch die Ordnung gegebenen Position sind die globalen Facilities über diese eindeutig identifizierbar.

Auch für die geschnittenen Netze existiert eine globale Ordnung derart, daß ein Input-Netz eines Modellblockes sich in derselben Position befindet, wie das dazugehörige Output-Netz auf dem anderen Block. Diese Ordnung wird über die passende Reihenfolge der geschnittenen Netze in den jeweiligen Signalschnitt-Listen gewährleistet. Im `pmod`-File existieren dafür sogenannte *Communication Facilities*, die die entsprechende Ordnung verkörpern (siehe Tabelle 4).

In den `pmod`-Files für die Slaves sind Facility-Indizes (vgl. S. 19) in Offset-Form abgespeichert. Ein Facility-Index ist ein zu einem Integer gecasteter Pointer, also letztendlich eine Adresse. Weil diese für alle Facilities aber aufgrund der TEXSIM-internen Implementierung linear aufeinanderfolgen, läßt sich ein Offset zur Startadresse ermitteln und dauerhaft abspeichern.[22] Damit ist es unnötig, die Facility-Namen in den Files zu verwenden (und diese damit aufzublähen). Außerdem spart man sich beim Laden der `pmod`-Files die aufwendige Berechnung des Facility-Indizes und die temporäre Nutzung (Speicheranforderung und -freigabe) des Namens. Sollen die binären Dateien bereits durch einen Partitionierer erzeugt werden, so muß er dazu den internen Aufbau in TEXSIM kennen. Tiefergehende Betrachtungen in diese Richtung würden hier aber zu weit führen.

Über die Unterscheidung in Netze, Vektoren und Arrays hinaus werden bei den globalen Facilities keine genaueren Typspezifikationen in den, die Modellblöcke ergänzenden, Dateien des parallelen Modells vorgenommen. Solche Typen sind z.B. L1- und L2-Arrays, Ein- und Ausgänge an L1- bzw. L2-Latchen usw. Würde man derartige Spezifikationen verwenden, so wären die Formate nicht allgemein genug für alle denkbaren Fälle. Bei neu eingeführten Typen wären Erweiterungen der Formate und Anpassungen an Partitionierer und *parallel*TEXSIM erforderlich. Überdies würde dies den Master und den Partitionierer verkomplizieren. Es hat sich bei der Parallelisierung gezeigt, daß der Verzicht auf genauere Typen problemlos möglich ist.

[21] Da es im Verlauf der Partitionierungsexperimente im Forschungsprojekt Fälle gab und gibt, bei denen nicht alle Teile eines Modells durch die Partitionierung erfaßt werden, darf die Liste auch leer sein.

[22] Für die RS/6000-Plattform unter AIX ließen sich sogar die eigentlichen Adressen aufgrund einer TEXSIM-eigenen Speicherverwaltung dauerhaft verwenden! Das widerspräche aber sämtlichen Prinzipien des modernen Programmdesigns.

4.3. Das Shellskript pTEXSIM

pTEXSIM ist das Programm (Datei: `ptexsim`), welches Master und Slave startet. Es ist als Korn-Shell-Skript implementiert. Unverzichtbar ist ein Shellskript wegen der Notwendigkeit, das PE über Shellvariablen vor dem Programmstart zu konfigurieren, wenn man dies nicht über Startoptionen auf unhandliche Weise bewerkstelligen will. Die Korn-Shell wurde gewählt, weil sie die Standard-Shell für AIX (und generell für Unix System V) ist.

Für ein Shell-Skript spricht weiterhin, daß durch dessen interpretative Abarbeitung keine Performance-Einbußen zu erwarten sind, da die hier durchzuführenden Arbeitsschritte weder zeitkritisch noch zeitintensiv sind. Zudem unterscheidet sich vom Standpunkt des Benutzers die Handhabung eines Skriptes nicht von der eines binären Programmes (Start-, Abbruchmöglichkeiten usw.). Außerdem sind die Möglichkeiten der Shell-Programmierung gerade für derartige Aufgaben optimiert und garantieren eine schnelle Implementierung und leichte Erweiterbarkeit.

Die Notwendigkeit für ein gesondertes Startprogramm ergibt sich daraus, daß das PE bislang nicht in der Lage ist, parallele Tasks dynamisch zu erzeugen. Alle Tasks eines parallelen Programmes werden während des Programmstarts gebildet. Dazu dient das Programm POE (Datei `poe`) aus dem Parallel Environment. Selbstverständlich ist es jedoch möglich, einzelne Tasks separat zu beenden.

Da es sich bei mTEXSIM und sTEXSIM um verschiedene Programme handelt, konnte auch nicht der Weg gewählt werden, daß ein Task anhand seiner Task ID selbst entscheidet, ob er als Master oder Slave fungieren soll. Daß es sich bei Master und Slave um verschiedene Programme handelt, war eine Konsequenz aus deren Komplexität. Allerdings ist mTEXSIM intern so implementiert, auch als normaler TEXSIM fungieren zu können und damit schon weitgehend darauf vorbereitet, als Slave zu arbeiten. Diese Möglichkeit ist vermutlich noch nicht völlig ausgereift und konnte wegen des begrenzten Zeitrahmens nicht ausgiebig getestet werden.

Die Aufgabe von pTEXSIM besteht darin, die notwendigen Umgebungsvariablen zu setzen und dynamisch ein Befehlsskript für POE zu generieren und dieses zu starten. In dem Befehlsskript stehen in jeder Zeile die Aufrufzeilen der parallelen Tasks mit der Reihenfolge gemäß ihrer zu vergebenden Task ID (d.h. mit dem Master in der ersten Zeile und den Slaves nachfolgend). So ist sichergestellt, daß die Task ID eines Slaves mit seiner Block ID übereinstimmt und der Master die ID 0 besitzt. Die Aufrufparameter der einzelnen Tasks ergeben sich aus denen von `ptexsim`. Dabei findet schon eine weitgehende Prüfung auf deren Korrektheit statt.

Eine sehr wichtige Teilaufgabe von pTEXSIM ist die Bestimmung der Anzahl der notwendigen Tasks anhand der Zahl der vorhandenen Modellblöcke. Aufgrund der sehr unterschiedlichen Einsatzplattformen (Parallelrechner oder Workstation-Cluster usw.) ist es möglich, *parallel*TEXSIM zu konfigurieren. Die Übernahme einer derartigen Konfiguration obliegt pTEXSIM. Diese Punkte werden in Abschnitt 4.6 im Rahmen der Beschreibung der praktischen Handhabung etwas genauer betrachtet.

Weil pTEXSIM so konstruiert ist, ein Befehlsskript für ein anderes Programm zu erzeugen, läßt es sich sehr einfach derart erweitern, daß es zusätzlich Jobfiles für Jobverteilungssysteme wie IBMs LoadLeveler erzeugt und an diese den entsprechenden Job übermittelt. Da dem Autor ein derartiges Verteilungssystem nicht dauerhaft zur Verfügung stand, konnte diese Möglichkeit leider nicht ausgenutzt werden.

4.4. Die Slaves sTEXSIM

Die Implementierung der Slaves ist durch verschiedene Aspekte mit großer eigenständiger Bedeutung gekennzeichnet. Neben dem eigentlichen Aufbau spielt auch die Frage der Kommunikation von Master und Slaves sowie der Slaves untereinander eine wesentliche Rolle. Die allergrößte Bedeutung besitzt aber der parallele Clock-Cycle-Algorithmus, denn nur dort ist eine Beschleunigung des gesamten Simulationslaufes möglich.

4.4.1. Die Programm-Struktur

Der Kern eines Slaves wird durch den geringfügig modifizierten Simulator TEXSIM gebildet, der durch ein Rahmenprogramm umschlossen ist. Das Rahmenprogramm (Frame) dient der Zusammenarbeit mit mTEXSIM und den anderen Slaves. Dazu verwendet es die durch die MPL des AIX PE gebotenen Kommunikationsprimitive (Abbildung 18).

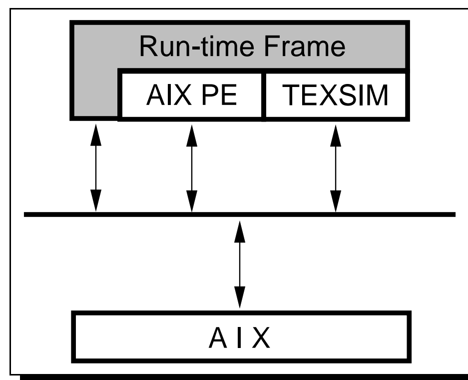


Abbildung 18: Der Aufbau von sTEXSIM

In seiner Initialisierungsphase verarbeitet sTEXSIM die zugeordnete Signalschnitt-Liste und die Cross-Referenz-Liste bzw. das entsprechende pmod-File. Eine dabei gebildete Speicherstruktur ist ein Array aus den lokalen Facility-Indizes der globalen Facilities. Die globalen Facilities, denen im vom jeweiligen Slave bearbeiteten Modellblock keine lokalen Facilities zugeordnet sind, haben hierbei den Facility Index 0. Da der Array-Index einer globalen Facility in den durch

die einzelnen Slaves gebildeten Arrays stets gleich ist (siehe Abschnitt 4.2), können die ihnen zugeordneten lokalen Facilities über diesen Index identifiziert werden.

Will der Master auf eine lokale Facility zugreifen, so schickt er in einer entprechenden Message an den Slave den Array-Index der globalen Facility mit, der von diesem dazu benutzt wird, den lokalen Facility Index zu erhalten und darüber auf die lokale Facility zuzugreifen.

Die geschnittenen Netze werden, getrennt nach Eingängen und Ausgängen, durch zwei Arrays repräsentiert. Diese Arrays enthalten Pointer auf, als *parallele Facilities* bezeichnete, Speicherstrukturen. Eine parallele Facility beinhaltet den Facility Index ihrer zugeordneten lokalen Facility, einen Netz-Index, einen Pointer auf ein Array aus Block IDs und die Anzahl der Elemente des Arrays. Der Netz-Index kodiert entweder den Vektor-Index eines Netzes aus einem Vektor oder das Nichtvorhandensein eines solchen. Im zugeordneten Array aus Block IDs sind die Nummern der Blöcke als vorzeichenlose Bytes gespeichert, mit denen das Netz als (Eingang bzw. Ausgang) verbunden ist.

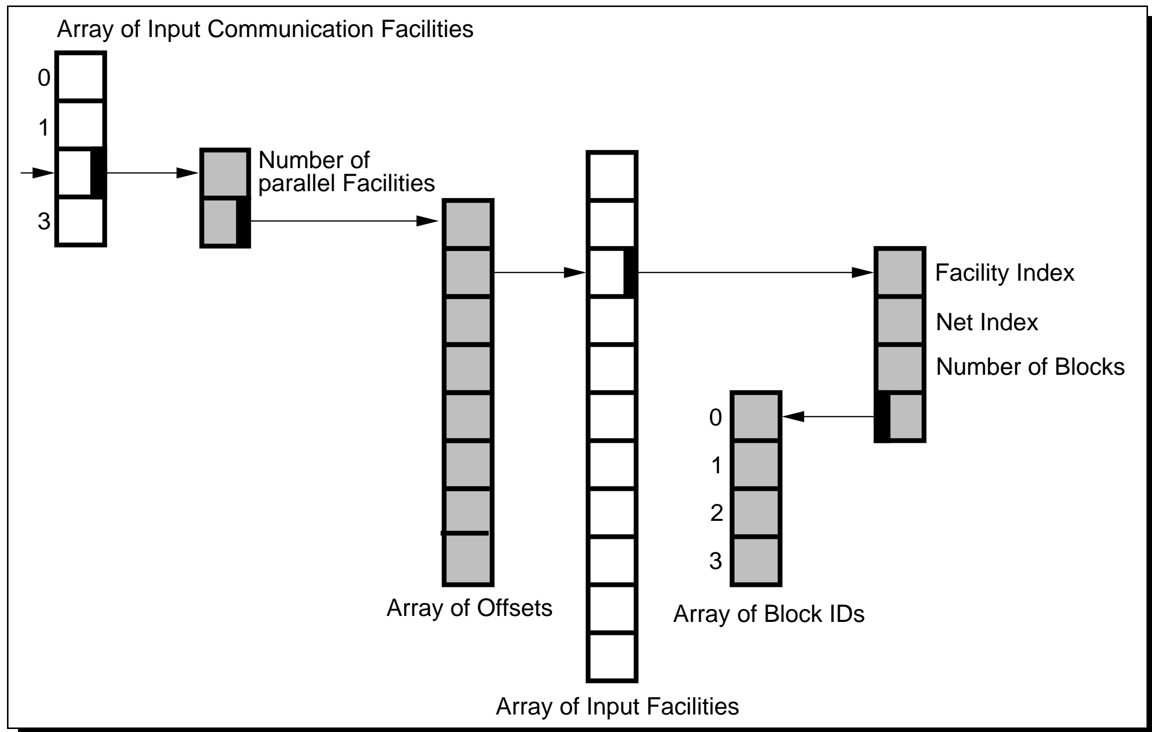


Abbildung 19: Aufbau der Communication Facilities

Die parallelen Facilities sind Bestandteil einer weiteren Speicherstrukturierung: Die Zuordnung der parallelen Facilities zu den einzelnen Blöcken geschieht noch einmal über sogenannte *Communication Facilities*. Es handelt sich dabei um Speicherstrukturen, die genau einem Block zugeordnet sind. Sie enthalten die Anzahl der parallelen Facilities des lokalen Blocks, die mit dem

jeweiligen Block verbunden sind, sowie ein Array aus den Offsets der Pointer auf die parallelen Facilities im jeweiligen Input- oder Output-Array. Die Pointer auf die Communication Facilities werden, wiederum getrennt nach Eingängen und Ausgängen, in zwei Arrays vorgehalten (Abbildung 19). Zu beachten ist, daß aus Gründen der einfacheren Organisation jeder Slave auch für seinen lokalen Block eine Input und eine Output Communication Facility anlegt.[23]

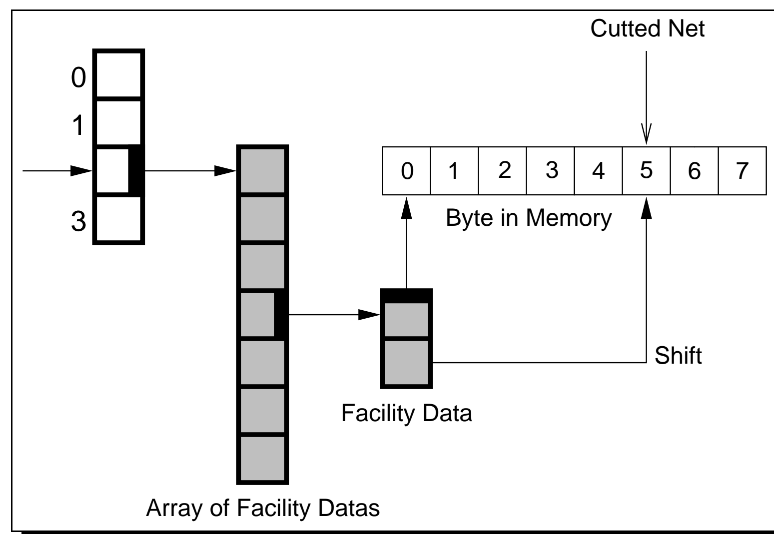


Abbildung 20: Zugriff auf ein geschnittenes Netz

Die Communication Facilities werden benötigt, um den parallelen Clock-Cycle-Algorithmus ausführen zu können, dessen konkrete Implementierung in Abschnitt 4.4.3 dokumentiert ist. Dort ist ein extrem schneller Zugriff auf die die geschnittenen Netze repräsentierenden Speicherstrukturen notwendig. Deshalb wird aus den Informationen der Communication Facilities für jedes Netz ein Pointer auf das es enthaltende Speicherwort und seine Bit-Position darin ermittelt (vgl. S. 14). Diese Daten werden jeweils in einer als *Facility Data* bezeichneten Struktur abgespeichert. Für den Zugriff auf die Strukturen selbst werden, getrennt nach Inputs und Outputs, Arrays aus Pointern auf die Facility Datas gebildet. Diese Arrays werden den einzelnen Blöcken in Form zweier Arrays (Inputs bzw. Outputs) aus Pointern auf diese Arrays derart zugeordnet, daß die einem Block so zugewiesenen Facility Datas diejenigen der zu ihm gehörigen geschnittenen Netze sind (Abbildung 20).

Die Anzahl der Elemente eines Arrays aus Pointern auf Facility Datas braucht nicht noch einmal abgespeichert zu werden, da sie schon in den Communication Facilities vorhanden ist. Die für die Communication Facilities generierten Datenstrukturen werden nämlich trotz ihrer nur temporären Verwendung nicht zeitaufwendig wieder entfernt, da sie insgesamt nur relativ wenig Speicher belegen. Sollte es sich irgendwann einmal aufgrund von Speicherknappheit als

[23]Die Anzahl der parallelen Facilties ist bei diesen jeweils Null und der Array-Pointer der Nullpointer.

nötig erweisen, den von ihnen belegten Speicher wieder verfügbar zu machen, so kann dies problemlos implementiert werden.[24]

Ein genereller Implementierungsgrundsatz bei *parallel*TEXSIM war die Verwendung von Arrays anstelle von verketteten Listen (die beispielsweise bei dvsim [17, 18] extensiv eingesetzt werden), da so über Pointer- und Offset-Arithmetik ein schnellstmöglicher Zugriff auf zu selektierende Elemente möglich ist. Außerdem wird dadurch der Speicherfragmentierung entgegengewirkt. Die Voraussetzung für diese Methodik waren natürlich die fixen Array-Größen, die aber durch die Problemnatur dieses Programms gegeben sind.

Für das Parsen der Signalschnitt- und der Cross-Referenz-Liste wird der in TEXSIM enthaltene Tokenizer verwendet, mit dem die komplett in den Haupspeicher geladenen Listenfiles in Token zerlegt werden und so selbst viele MByte große Dateien extrem schnell verarbeitet werden können.

TEXSIM definiert für einige Unix-Prozeß-Signale (z.B. `SIGTERM`) eigene Behandlungsroutinen (Signal-Handler). Von sTEXSIM werden zusätzlich Pointer auf die Standard-Signal-Handler des POE für die entsprechenden Signale abgespeichert. Damit werden diese Handler dann nach Abarbeitung der TEXSIM-eigenen Handler aufgerufen.

4.4.2. Der Nachrichtenaustausch

Jeder Slave ruft nach Beendigung seiner Initialisierungsphase eine permanent laufende Warteschleife, den sogenannten *Dispatcher*, auf, in der auf Anweisungen vom Master in Form von Master-Slave-Messages gewartet wird. Nach Erhalt einer derartigen Nachricht ruft der Dispatcher eine der Nachricht zugeordnete Handler-Funktion auf bzw. beendet sich und damit den Slave.

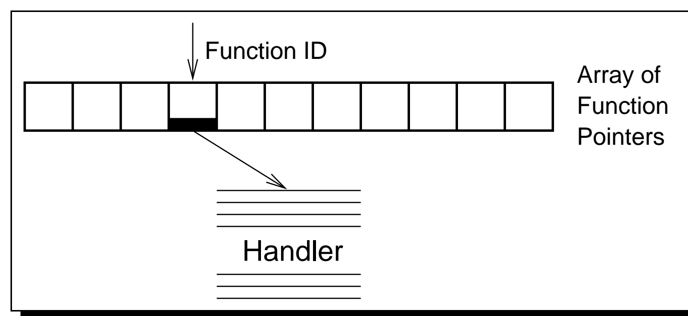


Abbildung 21: Verweis auf eine Message-Handler-Funktion

Die Message-Handler-Funktionen werden vom Dispatcher über eine Funktionstabelle referenziert. Dabei handelt es sich um ein Array aus Pointern auf die Handler-Funktionen. Der ei-

[24]Es ist dann nur die genannte Anzahl der Array-Elemente anderweitig abzuspeichern.

ner Funktion auf diese Weise zugeordnete Index (Funktionsindex) wird als *Function Identifier* (Function ID) bezeichnet und ist eine für Master und Slaves zum Compilationszeitpunkt global festgelegte Eigenschaft.

Die Master-Slave-Messages besitzen alle einen Aufbau aus 32-Bit Integern gemäß Abbildung 22. Das erste Nachrichten-Element ist immer der Funktionsindex, danach folgen die Parameter der Handler-Funktion, so sie welche hat. Die Parameteranzahl ist variabel; der Empfangspuffer des Dispatchers ist so groß gewählt, wie die maximal mögliche Nachrichtengröße es erfordert.

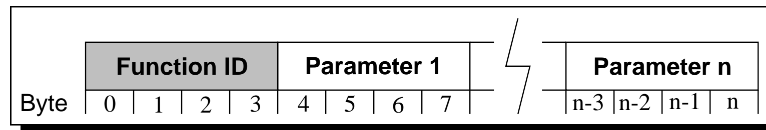


Abbildung 22: Struktur der Master-Slave-Messages

Der Aufruf der entsprechenden Handler-Funktion erfolgt über den Funktionsindex und als Parameter wird ein Pointer auf die gesamte Message übergeben. Somit ist es nicht nötig, eine Nachricht in einem zeitaufwendigen `switch case` oder `if then else` Block zu verarbeiten.

Eine Handler-Funktion kann separat mit dem Master oder den anderen Slaves kommunizieren. *parallel*TEXSIM ist so konstruiert, daß zwischen den Slaves nur kollektives Message-Passing stattfindet, d.h., alle Slaves kommunizieren auf einmal mit allen anderen. Weiterhin kann entweder nur ein Slave Nachrichten an den Master senden, oder der Master und alle Slaves kommunizieren kollektiv. Auf diese Weise ist das Message Passing logisch so aufgebaut, daß keine Art von „Spaghetti-Code in Raum und Zeit" entsteht, ohne daß dies Performance-Nachteile mit sich bringt. Solch eine vorteilhafte Methode war aber nur möglich, weil die Natur des Problems dies zuläßt.

Eine besondere Handler-Funktion übermittelt dem Master auf dessen Anforderung (im Rahmen eines initialen Statusprotokolls) eine Statusmeldung. Dabei handelt es sich um einen Integer, in dem kodiert ist, ob die Initialisierungsphase fehlerfrei ablief (Code 0) oder welcher Fehlertyp auftrat (Code > 0).

Tritt aber ein Fehler nach Beendigung der Initialisierungsphase auf, so wird nach der Terminierung der eigentlichen Simulatorinstanz der `mpc_stopall()`-Befehl der MPL von der betroffenen Instanz von sTEXSIM ausgeführt, der die Terminierung der anderen Tasks an den entsprechenden Handler des POE delegiert.

Nach erfolgreicher Beendigung des Statusprotokolls weist der Master die Slaves an, ihrer Task-Gruppe und ihre Kommunikationsvektoren zu bilden. Über eine kollektive MPL-Routine wird dann die Task-Gruppe der Slaves gebildet. Anschließend wird im Rahmen einer kollektiven Operation der Slaves die über alle Modellblöcke maximale Anzahl der Outputs eines Blocks an einen anderen bestimmt. Davon ausgehend werden die Kommunikationsvektoren gebildet.

4.4.3. Der parallele Clock-Cycle-Algorithmus

Der parallele Clock-Cycle-Algorithmus wird von einem Slave nach Erhalt einer Master-Slave-Message ausgeführt, die spezifiziert, daß entweder das gesamte Modell zu „clocken" ist, d.h. es sollen volle Taktzyklen simuliert werden, oder die nur das „Clocken" der kombinatorischen Logik, nicht aber der Latches, anweist.[25] Der Aufruf der parallelen Clock-Cycle-Routine enthält dann die Anzahl der zu simulierenden Zyklen (macht nur beim „Clocken" des Gesamtmodells Sinn) und einen Pointer auf die entsprechende sequentielle Simulationsroutine (`clockc()` bzw. `cyclec()`).

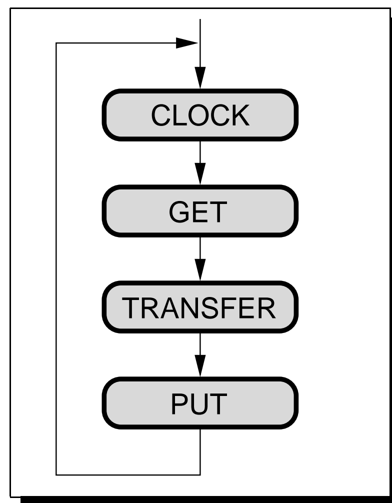


Abbildung 23: Implementierung des parallelen Clock Cycle

Abbildung 23 zeigt den Ablauf des parallelen Clock-Cycle-Algorithmus. Am Beginn wird der Schritt CLOCK ausgeführt, bei dem ein Zyklus auf den einzelnen Modellblöcken mittels des sequentiellen Clock-Cycle-Algorithmus simuliert wird. Danach werden in GET die Werte der geschnittenen Netze aus den Blöcken ausgelesen und in den Kommunikationsvektor geschrieben. Dieser wird dann im Schritt TRANSFER auf die anderen Slaves verteilt. Nach dem Datentransfer wird der Inhalt des Kommunikationsvektors im Schritt PUT ausgelesen und in die sequentiellen Modellblöcke geschrieben. Anschließend ist entweder ein neuer paralleler Zyklus zu bearbeiten oder der parallele Clock-Cycle beendet sich mit einer Synchronisation von Master und Slaves.

Die konkrete Implementation des Schrittes TRANSFER basiert auf einer kollektiven Operation der Slaves, die eine gemeinsame logische Gruppe von Tasks bilden. Bei der Operation handelt es sich um den `mpc_index()`-Befehl der MPL (Abbildung 24). Dieser entspricht dem `alltoall()`-Befehl des Message Passing Interface [26].

[25] Indem man nur die kombinatorische Logik „clockt", kann man die Logik von einem ungültigen Status in einen gültigen überführen (vgl. S. 18).

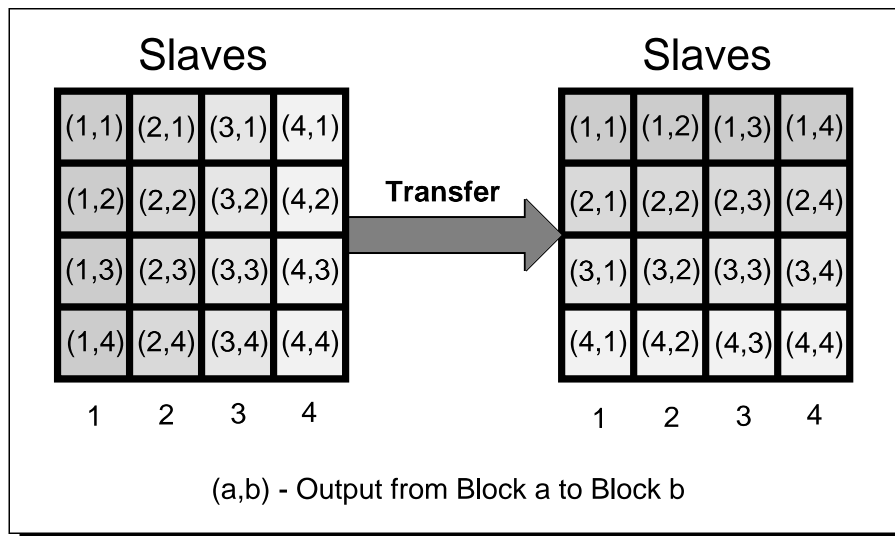


Abbildung 24: Datentransfer beim parallelen Clock Cycle

Die Verwendung solch einer eleganten und einfachen Operation ist aber nur möglich, da die Menge der zwischen zwei Modellblöcken zu transferierenden Daten relativ gering ist ($<$ 50 KByte). Ihr Vorteil liegt darin, daß zu ihrer Implementierung direkt auf unterliegende Kommunikations-Schichten zugegriffen werden kann. Somit sollte sie effizienter sein, als äquivalente Konstruktionen mit Funktionen aus der MPL. Das haben verschiedene Experimente bestätigt.

Der `mpc_index()`-Befehl transformiert auf jeder sTEXSIM-Instanz den sogenannten *Communication Vector* aus Abbildung 25. Es handelt sich dabei um ein Integer-Array, das in gleichgroße Einheiten unterteilt ist, die einem Block zugeordnet sind. In den Integern solch einer Einheit sind vor der Ausführung von `mpc_index()` die Ausgangsnetze der jeweiligen Instanz an den entsprechenden Block aufeinanderfolgend als Bits in den Integern eingetragen. Nach `mpc_index()` befinden sich dort in gleicher Weise die Eingangsnetze, die die Instanz von dem der Einheit zugeordneten Modellblock erhält.

Die Anzahl der Integer-Elemente in einer Einheit ist auf allen Instanzen von sTEXSIM gleich und ergibt sich aus der Maximalzahl der Ausgänge, die ein Block an einen anderen Block zu liefern hat. Die Reihenfolge der Einzelnetze in einer Einheit ergibt sich aus der Reihenfolge der Facility Datas (und damit aus der Reihenfolge der Communication Facilities). Diese Reihenfolge wiederum ist in folgendem Sinne über alle Instanzen global: Die Reihenfolge der Ausgangsnetze auf der Instanz, die die Ausgänge liefert, ist gleich der Reihenfolge der Eingangsnetze auf der Instanz, die die Eingangsnetze erhält.

Diese Reihenfolge ergibt sich aus der Verarbeitung der Signalschnitt-Liste bzw. des zugehörigen `pmod`-Files (siehe S. 30), die sich in Form der Anordnung der jeweiligen Pointer in den Pointer-Arrays widerspiegelt (vgl. Abschnitt 4.4.1). Somit können in den Schritten GET bzw.

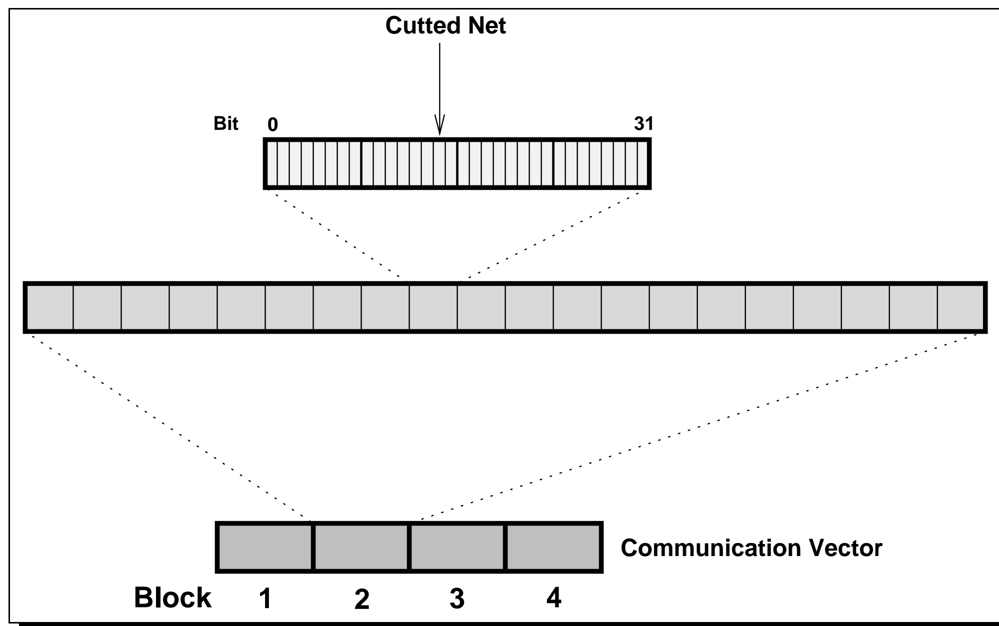


Abbildung 25: Aufbau eines Communication Vector

PUT die entsprechenden Netze aus dem Modellblock ausgelesen bzw. in den Block geschrieben werden, indem man einfach die Facility Datas gemäß ihrer Anordnung abarbeitet. So spart man nicht nur Zeit für die Netzidentifizierung sondern auch für für den eigentlichen Datentransfer der Netzbelegungen, weil keine zusätzlichen Informationen zur Netzkennzeichnung übertragen werden müssen.

4.5. Der Master mTEXSIM

4.5.1. Aufbau

mTEXSIM basiert auf den stark modifizierten und um einen parallelen Kern erweiterten Quellen von TEXSIM. Abbildung 26 symbolisiert den strukturellen Aufbau des Masters. Die Änderungen betreffen im wesentlichen die Schnittstellen. Da mTEXSIM so konzipiert ist, je nach Startmodus als Master oder als normaler TEXSIM zu fungieren, verwenden die Schnittstellenroutinen entweder Funktionen des sequentiellen oder des parallelen Kerns (Kernels). Allerdings ist eine solche Unterscheidung nicht bei allen Schnittstellenroutinen notwendig.

Ebenso wie bei den Slaves wird das eigentliche Message-Passing in einem *Communication Module* abgekapselt (Abbildung 14). Die Funktionen des Master-Moduls tragen in der Regel `get_` und die des Slave-Moduls `put_` als Namenspräfix.

Die zugrundegelegten modifizierten TEXSIM-Sourcen werden gleichzeitig auch für sTEXSIM verwendet. Dies geschieht über entsprechende Präprozessoranweisungen im Quelltext und ein daran angepasstes Makefile. So ist der normale TEXSIM weiterhin aus den Sourcen erzeugbar, und die Weiterentwicklung aller drei Programme läßt sich gemeinsam betreiben.

Wie bei sTEXSIM auch wird zum Parsen der Cross-Referenz-Liste der sehr schnelle Tokenizer verwendet, für den sich keine Notwendigkeit zur Anpassung ergab. Zum Laden der komprimierten Files des parallelen Modells wird (genauso wie bei sTEXSIM) das Programm `gzip` gestartet, das die dekomprimierten Daten über eine Pipe an mTEXSIM (bzw. sTEXSIM) liefert.

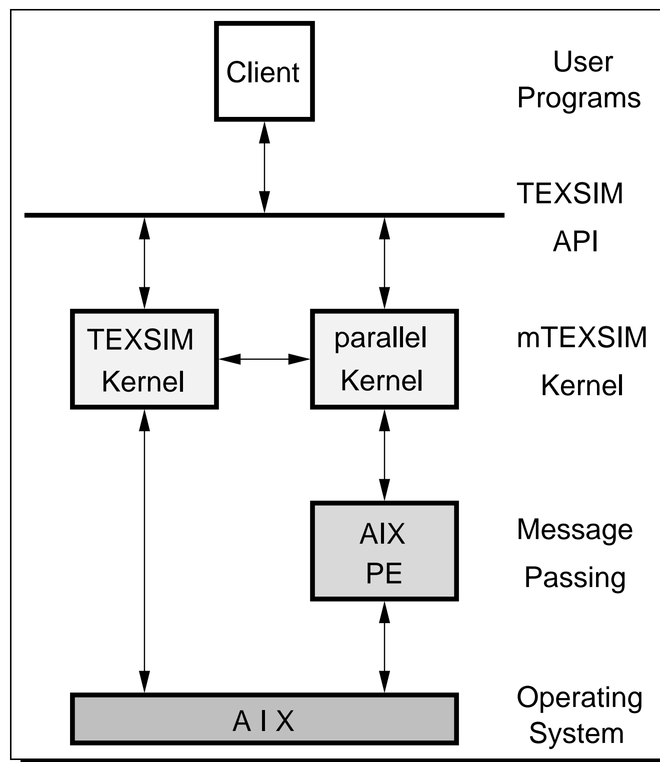


Abbildung 26: Der Aufbau von mTEXSIM

Die folgenden Betrachtungen beziehen sich nur noch auf einen als Master gestarteten mTEXSIM.

Nach seiner Initialisierungsphase führt der Master mit allen Slaves ein Fehlerprotokoll durch, bei dem alle Task ihren Fehlerstatus an den Master übermitteln. Liefern Tasks Fehlermeldungen, so beendet sich mTEXSIM mit einer Auflistung der den Tasks zugeordneten Fehlercodes. Jedes Mal, bevor der Master sich beendet, übermittelt er an alle Slaves eine Message, die diese anweist, sich ebenfalls zu beenden. Im Gegensatz zu den Slaves, braucht er deshalb bei die Beendigung erzwingenden Fehlern nicht die MPL-Funktion `mpc_stopall()`, um alle Tasks nach Abschluß der Initialisierungsphase zu terminieren.

Bei der Verarbeitung der Cross-Referenz-Liste bzw. des pmod-Files werden Speicherstrukturen gemäß Abbildung 27 aufgebaut. Es handelt sich dabei um zu den üblichen TEXSIM-Facilities äquivalente Datenstrukturen, die die Aufteilung dieser in globale, parallele und lokale Facilities (vgl. Abschnitt 4.2) berücksichtigen.

Die Facility-Namen werden aufeinanderfolgend abgespeichert. Verschiedene parallele Facilities können den gleichen Facility Namen besitzen. Die die parallelen Facilities verkörpernden Speicherstrukturen beinhalten deshalb einen Pointer auf den entsprechenden Namensanfang. Damit diese Pointer-Zuweisung möglich ist, wird in einem vorhergehenden Schritt der Facility Index der globalen Facility ermittelt und abgespeichert. Über diesen ist dann der Name erfaßbar.

Weiterhin sind den parallelen Facilities lokale Facilities zugeordnet. Diesem Umstand wird mit einem Pointer auf ein Array aus Block IDs Rechnung getragen, in welchem die IDs der Blöcke eingetragen sind, auf denen sich die entsprechenden lokalen Facilities befinden. Die jeweilige Blockanzahl ist ebenso mit in der Struktur enthalten.

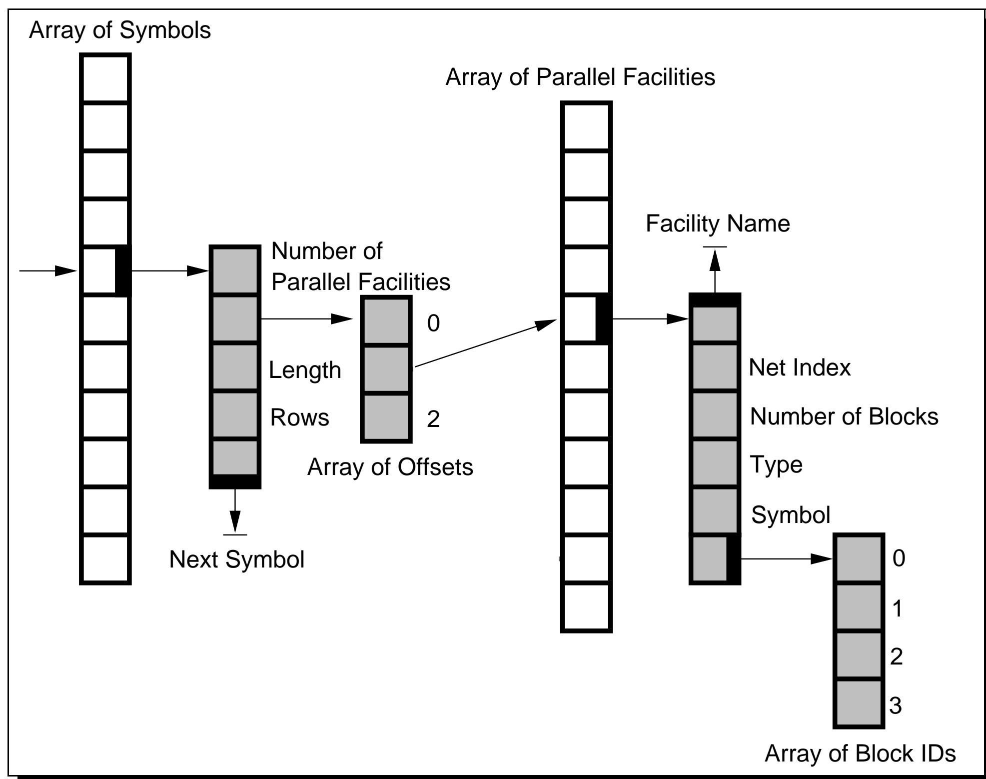


Abbildung 27: Struktur der Facilities

Da es Einzelnetze mit (indizierte Netze) und ohne Index gibt, ist eine diesbezügliche Typunterscheidung in der Datenstruktur enthalten. Darüber hinaus kodiert der Netz-Index entweder

den Vektor-Index eines Teilnetzes aus einem Vektor, den Maximalindex eines Vektors oder die Nichtexistenz eines Indizes.

Für den Zugriff auf die parallelen Facilities existiert ein Array aus Pointern auf die beschriebenen Speicherstrukturen. Nach dessen Erzeugung lassen sich die globalen Facilities aus den parallelen bilden. Dazu werden die parallelen Facilities mit gleichem Facility-Namen durch das Abspeichern ihrer Offsets im Pointer-Array in jeweils einem Array zusammengefaßt.

Die Anordnung der Pointer im Pointer-Array folgt dabei der in der Cross-Referenz-Liste vorgegeben Ordnung der parallelen Facilities. Für den Zugriff auf lokale Facilities wird dann dem jeweiligen Slave der Offset des Pointers im Pointer-Array übermittelt. Dieser dient so der eindeutigen Identifizierung der parallelen Facilities in allen Tasks von *parallel*TEXSIM (siehe auch Abschnitt 4.4.1).

Eine globale Facility ist durch eine Speicherstruktur verkörpert, die einen Pointer auf ein derartiges Array enthält. Mit den Informationen in den Strukturen der so zugewiesenen parallelen Facilities lassen sich dann die Anzahl der Rows und der Bits in einer Row (siehe Abschnitt 3.2) einer globalen Facility ermitteln.

Für Arrays müssen diese Daten von einem Modellblock angefordert werden. Weil Arrays bei der Partitionierung nicht aufgeteilt werden, gibt es für sie nur eine zugehörige parallele Facility (aber eventuell mehrere lokale Facilities). Hier wird dann immer auf den ersten Block aus dem Array mit den entsprechenden Block IDs zugegriffen. Da die Anzahl der Arrays in einem Modell in der Regel sehr klein ist, ergibt sich nur ein unwesentlicher Overhead durch das nötige Message-Passing.

Bei allen anderen globalen Facilities ist die Anzahl der Rows stets 0. Ebenso wie Arrays haben Einzelnetze nur eine zugehörige parallele Facility und die Anzahl der Bits in einer Row ist 0 für nichtindizierte Netze bzw. gleich dem Netz-Index.

Ist ein Vektor durch die Partitionierung aufgeteilt worden, so sind der entsprechenden globalen Facility soviele parallele Facilities zugeordnet, wie der Vektor Einzelnetze besitzt. Der Vektor-Index der gemäß dem entsprechenden Offset-Array letzten parallelen Facility liefert dann die Vektorlänge. Für Vektoren, die nicht aufgeteilt werden, existiert wiederum nur eine zugehörige parallele Facility, deren Netz-Index die Vektorlänge kodiert.

Neben dem Pointer auf das Offset-Array, der Anzahl von Rows und der Anzahl von Bits in einer Row beinhaltet die einer globalen Facility entsprechende Speicherstruktur noch die Anzahl der Elemente des Offset-Arrays und einen Pointer auf eine andere globale Facility, der für das Facility-Hashing benötigt wird.

Zum Zugriff auf die globalen Facilities existiert ein Array aus Pointern auf die jeweiligen Speicherstrukturen. Darüber hinaus existiert eine Hash-Table aus Pointern auf globale Facilities auf der Basis der Hash-Funktion, die sich schon bei TEXSIM bestens bewährt hat. So ist es möglich, über den Facility-Namen einen Pointer auf die Speicherstruktur der globalen Facility zu erhalten. Kollisionen, also Facility-Namen mit gleichem Hash-Index, werden über den erwähnten Pointer auf eine andere globale Facility aufgelöst.

Zur Erzeugung der Hash-Table werden die globalen Facilities gemäß ihrer Anordnung im Pointer-Array abgearbeitet. Tritt eine Kollision auf, so äußert sich dies darin, daß die dem Hash-Index entsprechende Stelle in der Hash-Table schon belegt ist. Dann wird der Pointer dort durch den Pointer auf die gerade bearbeitete globale Facility ersetzt, und der Pointer in der Speicherstruktur der aktuellen globalen Facility verweist auf die globale Facility, die vor dem Austausch die Stelle in der Hash-Table belegt hatte. Auf diese Weise bilden die beim Hashing jeweils miteinander kollidierenden globalen Facilities eine verkettete Liste, die beim Ermitteln des Facility Indizes aus dem Facility Namen durch die API-Funktion `efsrtsym()` so lange abgearbeitet werden muß, bis die Namen übereinstimmen.

4.5.2. Die Schnittstellen

Da *parallel*TEXSIM weitgehend kompatibel zu TEXSIM ist, sind auch alle seiner Schnittstellen implementiert. Neben der Verarbeitung sämtlicher Eingabe-Dateien können auch alle Arten von Ausgabe-Dateien erzeugt werden. Die Programmierschnittstellen lassen sich in ein C-API und ein Subcommand-Interface unterteilen (Abschnitt 3.2).

Von den Schnittstellenroutinen werden oftmals Funktionen des parallelen Kernels aufgerufen, die den Großteil der notwendigen Fähigkeiten implementieren und den gleichen Namen mit einem „_“ als Präfix tragen (z.B. ruft `efsrtsym()` die parallele Version `_efsrtsym()` auf). Die Ursache hierfür liegt im Entwicklungsprozeß der Parallelisierung begründet.

Wie in [2] erläutert, wurden parallelisierte Varianten an den sequentiellen Kernel angefügt (bildlich gesprochen wurde so ein Rahmenprogramm angebaut), die dann in einem späteren Schritt mit dem sequentiellen Kernel verschmolzen wurden (sequentieller und paralleler Kernel wurden miteinander verzahnt). Die dort aufgeführten Begründungen für diese Vorgehensweise haben sich im Verlauf der Entwicklung als voll gerechtfertigt erwiesen:

- die eigentliche Parallelisierung ließ sich relativ schnell durchführen und Probleme auch im Zusammenhang mit der Partitionierung konnten frühzeitig erkannt werden;
- parallele und sequentielle Versionen der API-Funktionen konnten solange gleichzeitig ausgeführt und direkt miteinander verglichen werden, wie der Master auch das sequentielle Modell simuliert hatte;
- der gesamte Entwicklungsprozeß war auf diese Weise gut strukturiert und damit besser organisierbar.

Die Facility-Hierarchie (global, parallel und lokal) spiegelt sich auch bei der Implementierung der Schnittstellenroutinen wider. Einige der dabei wesentlichen Aspekte sollen im folgenden betrachtet werden.

Ist einer globalen Facility nur eine parallele Facility zugeordnet, kann das Abfragen ihres Zustandes direkt an eine lokale Facility weitergeleitet werden, weil sich diese alle im gleichen Zustand befinden. Praktischerweise wird dabei die gemäß des Arrays der Block IDs erste genommen. Soll aber ihr Zustand verändert werden, so sind diese Änderungen an allen der parallelen Facility zugeordneten lokalen Facilities vorzunehmen.

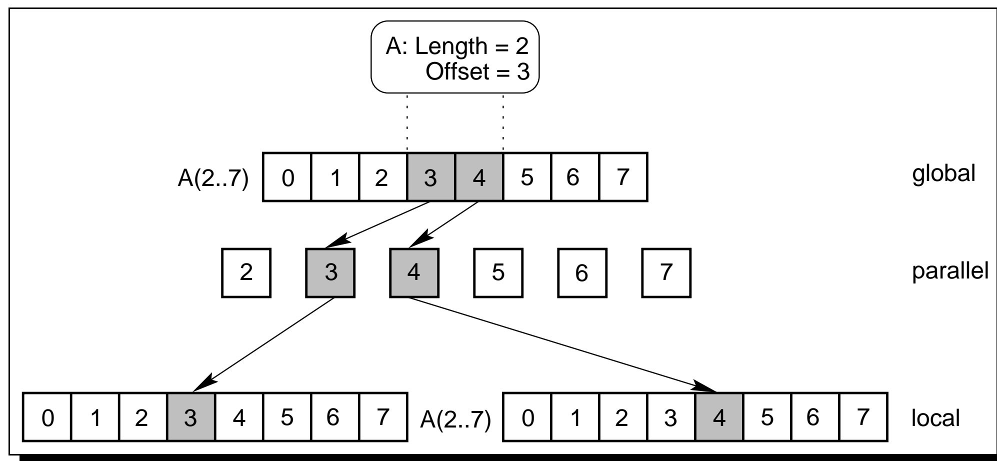


Abbildung 28: Zugriff auf eine aufgeteilte Facility

Komplizierter wird es, wenn eine globale Facility über mehrere parallele Facilities verfügt. Es handelt sich dabei um einen durch die Partitionierung aufgeteilten Vektor aus Einzelnetzen. Bei der Zustandsabfrage werden die Einzelinformationen der parallelen Facilities so zusammengefügt, daß sie die Gesamtinformation über die globale Facility ergeben. Bei der Abfrage der parallelen Facilities wird wiederum die gemäß dem jeweiligen Array aus Block IDs erste lokale Facility verwendet. Für die Änderung des Zustandes der globalen Facility muß die Information dann in Einzelinformationen zerlegt und jeweils an die zugehörigen parallelen Facilities und damit an die dementsprechenden lokalen Facilities weitergeleitet werden.

Abbildung 28 zeigt das Vorgehen am Beispiel eines Vektors A, der aus 6 Einzelnetzen mit Indizes aus $\{2, \ldots, 7\}$ besteht.[26] Dabei ist eine Facility Referenz (vgl. S. 19) auf A mit Offset 3 und Länge 2 dargestellt. Jedem Einzelnetz entspricht eine parallele Facility, der wiederum lokale Facilities (von denen jeweils eine dargestellt ist) zugeordnet sind.

Bei den Routinen des Integer-Interfaces wird das Aufteilen bzw. das Zusammensetzen der Informationen sehr effizient über geeignete Schiebeoperationen bzw. über binäres Oder durchgeführt. Für die Funktionen des Character-Interfaces sind temporäre Umwandlungen in binäre Strings erforderlich, falls diese noch nicht vorlagen, an denen dann äquivalente Operationen vorgenommen werden. Dabei ist auch mehrwertige Logik berücksichtigt.

[26] Es hätten auch nicht alle Netze aufeinanderfolgen müssen.

4.6. Praktische Handhabung

Die praktische Handhabung von *parallel*TEXSIM gestaltet sich weitgehend ohne Unterschied zu TEXSIM. Einige Dinge, die es es dabei jedoch naturgemäß zu beachten gibt, sollen in diesem Abschnitt angesprochen werden.

4.6.1. Der Programmstart

Der Aufruf in einer Kommandozeile erfolgt vollkommen analog:[27]

`ptexsim ⟨model name⟩ [⟨option⟩] ⟨subcommand⟩ [⟨option⟩]`

Bis auf eine sind alle Programmoptionen zulässig. Die Option `-host` wird abgewiesen, weil sie dazu dient, den statisch angelinkten TEXSIM unter der Kontrolle eines anderen Simulators ablaufen zu lassen. Eine solche (sehr selten verwendete) Möglichkeit konnte aus Zeitgründen nicht implementiert werden. Zusätzlich gibt es weitere Optionen. Mit `-nopartex` verhält sich der Simulator wie TEXSIM, d.h., er arbeitet sequentiell. Dieses Feature befindet sich aber noch in einem experimentellen Zustand und sollte nicht verwendet werden. Die Optionen `-pxref` bzw. `-ext` weisen Master bzw. Slave an, keine binären Listenfiles zu verwenden und auch keine zu erzeugen. Werden `-holdsims` bzw. `-holdaets` nicht angegeben, so werden die durch den Simulator erzeugten Message-Files (Extension `sim`) bzw. die All-Events-Trace-Files (Extension `aet`) der einzelnen TEXSIM-Instanzen nach Beendigung des Programmlaufs gelöscht. Für die selteneren Typen von Ausgabe-Dateien sind die von den Slaves erzeugten Files stets durch den Nutzer zu löschen. Generell werden die die Erzeugung von bestimmten Ausgabe-Dateien anfordernden Startoptionen an die Slaves weitergegeben. Weil die verschiedenen Typen von Ausgabe-Dateien eines Simulationslaufes sich nur hinsichtlich ihrer Extension unterscheiden, ansonsten aber den Modellnamen tragen, haben diejenigen der TEXSIM-Instanzen

`⟨model name⟩.⟨block id⟩.⟨extension⟩`

als Namensschema. Dies gilt auch für die durch den Simulator erzeugten binären Listen-Files.

Die Option `-msg` schaltet normalerweise die Ausgabe der Simulationsprotokolle auf der Standardausgabe ein, die sonst nur in die Message-Files geschrieben werden (falls dies nicht explizit mittels einer entsprechenden Start-Option abgeschaltet wird). Von `ptexsim` wird stattdessen aber nur eine Copyright-Meldung geliefert. Der Verzicht auf die Protokollausgabe liegt in der Pufferung (Standard-Puffergröße 4096 bzw. 8192 KB) aller Daten, die an die Standardausgabe geliefert werden, begründet, die vom PE zur Steigerung der Performance vorgenommen wird. Würde die Ausgabe zugelassen, könnte das sehr schnell zum Pufferüberlauf und damit zum Programmabbruch führen. Eine manuelle Entleerung des Puffers durch *parallel*TEXSIM kann nur als synchrone Operation aller parallelen Tasks stattfinden, so daß wiederum aus Performance-Gründen darauf verzichtet wurde.

[27] Der Modellname ist dabei durch den Partitionierer vorgeben.

Das ist auch der Grund dafür, daß es nicht sinnvoll möglich ist, den interaktiven Kommandoprozessor zu benutzen. Dafür ergibt sich aber auch keine Notwendigkeit, da *parallel*TEXSIM nicht für die interaktive Simulation geeignet und gedacht ist (vgl. Abschnitt 3.3).

Die vorliegende Version von *parallel*TEXSIM kann noch keine All-Events-Traces (AETs), also Signal-Wert-Verläufe, im eigentlichen Sinne erzeugen. Vielmehr werden nur bei Verwendung der entsprechenden Start-Optionen solche durch die Slaves erzeugt. In der Praxis werden diese aber für die für *parallel*TEXSIM anvisierten Einsatzzwecke nicht häufig gebraucht. Für eine spätere Version ist die volle Unterstützung der AETs vorgesehen. Dafür schlägt der Autor vor, die von den Slaves erzeugten AET-Files zu mergen.

Da die Überprüfung der Startparameter weitgehend durch das Startskript `ptexsim` erfolgt, gibt es im Fehlerfall keine Message-Files, die sonst die Fehlermeldung beinhalten, da dann ein Programmstart gar nicht erst erfolgt. Eine Ausnahme bildet die Option `-host`, die nur zur einer Warnung führt und nicht verwendet wird. Die anderen Ausnahmen ergeben sich aus Fehlern, die vom Skript nicht abgefangen wurden und dann durch Master und/oder Slaves moniert werden. Das Startskript meldet Fehler immer über die Standardausgabe.

4.6.2. Die Programmkonfiguration

Die Blöcke des zu simulierenden Modells werden von `ptexsim` in den in der Umgebungsvariablen `SIMIN` eingetragenen Verzeichnissen gesucht. Ist `SIMIN` nicht gesetzt, so wird im aktuellen Verzeichnis gesucht. Dieses ist mit in die Variable aufzunehmen, falls sich die Modellblöcke dort befinden und sie gesetzt wird. Die Modellblöcke sollten nicht über mehrere Verzeichnisse verteilt werden, obwohl dies unter gewissen Umständen möglich ist![28] Werden nicht mindestens zwei Modellblöcke gefunden, so beendet sich *parallel*TEXSIM mit einer Fehlermeldung (außer wenn `-nopartex` als Startoption gewählt wird).

Auch für die Cross-Referenz- und die Signalschnitt-Listen muß entsprechend den Modellblöcken verfahren werden. Es empfiehlt sich aus Gründen der besseren Übersichtlichkeit, Modellblöcke und Listen-Files in getrennten Verzeichnissen zu halten. Zur Beschleunigung von mehrfachen Läufen über einem parallelen Modell erzeugt *parallel*TEXSIM binäre Listen (Extension `pmod`), die im durch die Umgebungsvariable `SIMOUT` definierten Verzeichnis abgespeichert werden. Ist die Variable nicht gesetzt, so wird dafür das aktuelle Verzeichnis verwendet. Die Erzeugung einer solchen Datei erfolgt standardmäßig, falls sie im `SIMIN`-Pfad (bzw. im aktuellen Verzeichnis) nicht gefunden werden kann. Im Falle ihrer Existenz erfolgt deren Verwendung anstatt eventuell vorhandener nicht binärer, außer wenn man das über eine entsprechende Startoption (s.o.) unterbindet. Ändern sich die Modellblöcke (mit beibehaltenem Modellnamen), so erkennt *parallel*TEXSIM, daß eventuell noch vorhandene binäre Listen-Dateien nicht dazu passen und beendet sich mit einer Fehlermeldung. Da der Partitionierer die

[28] Auf deren genaue Angabe hier wird verzichtet, da sich bei ihrer Ausnutzung leicht unerwünschte Nebenwirkungen ergeben könnten.

nicht binären Listen-Files zusammen mit den Modellblöcken bereitstellt, erfolgt bei Ihnen keine Überprüfung. Aus diesem Grund sind Cross-Referenz- und Signalschnitt-Listen gemeinsam und nicht separat durch (aktuellere) Versionen zu ersetzen.

Binäre und nicht binäre Listen-Dateien können in komprimierter Form vorliegen, und für nicht binäre ist das sehr zu empfehlen. Sie tragen dann zusätzlich die Extension `gz`, da sie das GNU `gzip`-Format haben. Dadurch ist es möglich, sie mit `gzip` bzw. `gunzip` selbst zu packen bzw. zu entpacken. Liegt ein Listen-File sowohl in komprimierter als auch in unkomprimierter Form vor, so verwendet *parallel*TEXSIM die unkomprimierte Version. Der Simulator erzeugt immer komprimierte binäre Listen-Dateien. Zu beachten ist außerdem, daß die pmod-Files intern eine Versionsnummer gespeichert haben. Dadurch wird gewährleistet, daß auch nur solche Dateien mit einem zum jeweiligen *parallel*TEXSIM kompatiblen Format geladen bzw., falls dies möglich ist, mit einer kompatiblen Laderoutine verwendet werden.

Alle Ausgabe-Dateien werden im durch die Umgebungsvariable `SIMOUT` definierten (bzw. im aktuellen) Verzeichnis erzeugt. Sämtliche Eingabe-Dateien werden im `SIMIN`-Pfad (bzw. im aktuellen Directory) gesucht. Die Werte dieser Variablen gelten für Master und Slaves, und auch das aktuelle Verzeichnis ist bei ihnen gleich. Das muß beachtet werden, wenn man nicht mit einem bei allen beteiligten Knoten gleichen Filesystem arbeitet! Eine derartige Konfiguration (z.B. mittels NFS oder AFS) ist aufgrund der besseren Wartbarkeit sehr zu empfehlen. Soll AFS eingesetzt werden, so sind dafür einige Änderungen am PE vom Systemverwalter vorzunehmen, die in den Handbüchern beschrieben werden.

`ptexsim` erzeugt aus seinen Startparametern im Ausgabe-Pfad ein für `poe` notwendiges Befehlsskript mit `.partex` als Namen. Weil es sich damit um eine nach Unix-Konvention verdeckte Datei handelt, dürfte es kaum Überschneidungen mit anderen Files gleichen Namens geben. Damit `ptexsim` ausgeführt werden kann, muß sich die Korn-Shell mit `/bin/ksh` aufrufen lassen.

Da es für die parallele Einsatzplattform viele Konfigurationsmöglichkeiten gibt (vor allem für die IBM RS/6000 SP), läßt sich *parallel*TEXSIM daran anpassen. Dies geschieht über ein Shellskript mit Namen `ptexsim.cfg`, welches sich im Shell-Suchpfad befinden muß. Prinzipiell ist es damit möglich, mehrere Konfigurationen mittels verschiedener Skripts zu realisieren. Ein einfaches Beispiel (Workstation-Cluster, Ethernet und Internet-Protokoll) für ein solches Konfigurationsskript bietet Abbildung 29. Für die Erstellung eines Konfigurationsskripts sind Kenntnisse in der Handhabung des AIX Parallel Environments notwendig. `ptexsim` selber sollte nicht modifiziert werden! Selbstverständlich ist `ptexsim.cfg` im Filesystem als ausführbar zu deklarieren.

Am Beispiel von Xmon soll noch die Verwendung von *parallel*TEXSIM mit einem Monitor (siehe Abschnitt 3.3) kurz angeschnitten werden. Xmon wird über ein Shellskript namens `rxmon` (run Xmon) gestartet. Dieses ist auf ein File mit Namen `prxmon` zu kopieren.[29] Darin ist `texsim` durch `ptexsim` zu ersetzen und um die zusätzlich gewünschten Startoptionen zu

[29] Damit paßt sich der Filename im Stil den Konventionen bei *parallel*TEXSIM an.

```
######################################################################
#                      parallel T E X S I M                           #
######################################################################
#                    U S E R - configuration                          #
######################################################################

# the POE environment variables
export MP_HOSTFILE=host.list;           # the host list
export MP_RESD=no;                      # no ressource manager
export MP_EUILIB=ip;                    # IP protocol
export MP_EUIDEVICE=en0;                # ethernet

# the TEXSIM environment variables
export SIMIN=.:$HOME/models:$HOME/simin # input directories
export SIMOUT=$HOME/simout              # the output directory

######################################################################
```

Abbildung 29: `ptexsim.cfg`-Beispiel für Workstation-Cluster

ergänzen. Da `prxmon` auch die Umgebungsvariablen `SIMIN` und `SIMOUT` setzt, dürfen sie in `ptexsim.cfg` nicht belegt werden. Ein solches Konfigurationsskript, welches gleichzeitig auch auf eine mögliche RS/6000 SP-Installation zugeschnitten ist, wird in Abbildung 30 wiedergegeben. Nicht zu vergessen ist, daß alle von Xmon benutzten Pfade auf allen Knoten identisch sein müssen. Große Sorgfalt ist wiederum nötig, wenn man kein gemeinsames Filesystem nutzt.

Aufgrund der durch mTEXSIM und sTEXSIM zu bearbeitenden enormen Datenmengen, die für höchstmögliche Laufzeit-Effizienz permanent im Arbeitsspeicher gehalten werden, sind dem Nutzer vom Systemverwalter entsprechende Limits zu setzen.[30] Andernfalls kann es sein, daß der dem Nutzer zur Verfügung stehende maximale Arbeitsspeicher nicht ausreichend ist. Die notwendigen Werte hängen vom zu simulierenden Modell, der Blockanzahl und dem verwendeten Partitionierungsalgorithmus ab.

Für maximale Performance von *parallel*TEXSIM muß man beachten, daß für den parallelen Clock-Cycle-Algorithmus der langsamste Task die Gesamtgeschwindigkeit bestimmt. Die bei einem parallelen Lauf verwendeten Prozessorknoten sollten also möglichst gleich durch andere Programme ausgelastet werden. Dafür hat man eine geeignete Host Liste zu verwenden. Besser ist es jedoch, dafür auf Jobverteilungsprogramme wie IBMs LoadLeveler zurückzugreifen, die automatisch die bestmögliche Lastverteilung ermöglichen.

TEXSIM bietet die Möglichkeit des sogenannten Checkpointing. Hierbei wird der aktuelle Zu-

[30] In der Korn Shell oder der GNU Bourne Again Shell können diese über den Befehl `ulimit` abgefragt werden.

```
######################################################################
#                        parallel T E X S I M                        #
######################################################################
#                       U S E R - configuration                      #
######################################################################

export SP_NAME=ldsphcws                  # DNS name of RS/6000 SP

export MP_CONTROL_WORKSTATION=$SP_NAME   # control workstation name
export MP_RESD=yes                       # use ressource manager
export MP_EUILIB=us                      # user space protocol
export MP_EUIDEVICE=css0                 # high performance switch
export MP_RMPOOL=2                       # use pool 2
```

Abbildung 30: `ptexsim.cfg`-Beispiel für Xmon und RS/6000 SP

stand des Modells in einem File abgespeichert, so daß er jederzeit wiederhergestellt werden kann. Im parallelen Fall werden in Analogie zu den AET-Files nur für die einzelnen Slaves solche Files erzeugt. Für die Wiederverwendung in *parallel*TEXSIM spielt das keine Rolle. Soll aber der Modellzustand in anderen Simulatoren über ein Checkpoint-File wiederhergestellt werden, so funktioniert das momentan daher noch nicht. Ein solcher Fall ist aber in dem für *parallel*TEXSIM angedachten Einsatzfeld nicht sehr häufig gegeben. Für eine spätere Version ist die volle Checkpoint-Unterstützung aber vorgesehen. Möglicherweise ist dafür die gleiche Strategie wie bei den AET-Files sinnvoll: das Mergen der Files der einzelnen Blöcke.

5. Der Clock-Cycle-Algorithmus

Zum Verständnis der folgenden Abschnitte ist die Betrachtung von Abbildung 5 sehr hilfreich. In ihnen werden die bislang nur verbal beschriebenen Algorithmen für den sequentiellen und den parallelen Clock-Cycle-Algorithmus auf der Grundlage mathematischer Modelle formal spezifiziert.

5.1. Das strukturelle Hardware-Modell

Als theoretische Grundlage für weitere Betrachtungen und Begriffsbildungen wird eine abstrakte Beschreibung eines Hardware-Designs als einfache Anpassung eines bereits vorhandenen Hardware-Modells für synchrone Schaltungen auf Gate- bzw. Registerebene angegeben, welches ausschließlich die rein strukturellen Aspekte der Hardware beinhaltet.

Es handelt sich dabei um eine geringfügige Modifikation des in [8] eingeführten Modells. Hierfür sei $\Gamma^{+}_{G}(x) = \{y \mid (x,y) \in \mathcal{R}\}$ die Menge der Nachfolger und $\Gamma^{-}_{G}(x) = \{y \mid (y,x) \in \mathcal{R}\}$ die Menge der Vorgänger von $x \in X$ für einen beliebigen gerichteten Graphen $G = (X, \mathcal{R})$.

Definition 5.1 (strukturelles Hardware-Modell sHWM)
Es seien M_E, M_L und M_S paarweise disjunkte, nichtleere endliche Mengen, M_I und M_O nichtleere disjunkte Teilmengen von M_S sowie $M_B = M_E \cup M_L$ und $M_R \subseteq (M_B \times M_S) \cup (M_S \times M_B)$. $\mathcal{M} = (M_I, M_O, M_E, M_L, M_S, M_R)$ ist ein strukturelles Hardware-Modell, *wenn der gerichtete bipartite Graph $G(\mathcal{M}) = (M_B, M_S, M_R)$ die folgenden Bedingungen (i)–(iii) erfüllt.*

(i) Es gilt $\left\{ s \mid s \in M_S \wedge \Gamma^{-}_{G(M)}(s) = \emptyset \right\} = M_I$.

(ii) Es gilt $\left\{ s \mid s \in M_S \wedge \Gamma^{+}_{G(M)}(s) = \emptyset \right\} = M_O$.

(iii) Jeder Kreis (Zyklus) in $G(\mathcal{M})$ enthält ein Element von M_L.

Die Elemente von M_B heißen *Boxen*, die von M_S *Signale*, die von M_R *Netze*, die von M_L *Latche* und die von M_E *Logikelemente*. M_I ist die Menge der *globalen Eingänge* und M_O die Menge der *globalen Ausgänge*. In der Literatur (z.B. in [6, 2]) trifft man auch auf die Begriffe PI (*Primary Input*) und PO (*Primary Output*), die sich nicht mit den hier angegebenen Begriffen des globalen Schaltungsein- und ausgangs decken. PIs bzw. POs sind solche Elemente von M_I bzw. M_O, die eine Schaltung nach außen hin bereitstellt (z.B. bei Mikroprozessoren die Pins). Demgegenüber existieren in der Praxis auch solche Signale, die innerhalb der Hardware nur mit einem Ein- bzw. Ausgang einer Box verbunden sind, auf die aber auch nicht von der Umgebung der Hardware aus zugegriffen werden kann. Eine solche Unterscheidung der globalen Ein- und Ausgänge ist hier jedoch nicht relevant, könnte aber leicht eingeführt werden: Man erweitert

die Struktur $\mathcal{M}$ aus Definition 5.1 einfach um zwei Mengen $M_{PI} \subseteq M_I$ (die PIs) und $M_{PO} \subseteq M_O$ (die POs), die auch leer sein können.

Abgesehen von der Endlichkeit der Mengen unterscheidet sich Definition 5.1 in (i) und (ii) (und damit auch bei der Festlegung von M_B) von der des Hardware-Modells aus [8]. Die Festlegung, daß ein Element x aus M_I oder M_O keine Box ist ($x \notin M_B$), sondern immer ein Signal ($x \in M_S$), entspricht auch der Situation bei TEXSIM.

Folgerung 5.1
Es gelten die Voraussetzungen von Definition 5.1. Dann existiert für jedes $s \in M_S$ ein $b \in M_B$ mit $(s,b) \in M_{\mathcal{R}}$ oder $(b,s) \in M_{\mathcal{R}}$.

Beweis: Aus (i) und (ii) von Definition 5.1 ergibt sich wegen $M_I \cap M_O = \emptyset$ und $M_I, M_O \neq \emptyset$ die Behauptung. □

5.2. Das Box-Levelizing

Um den *Clock-Cycle*-Algorithmus (Abschnitt 5.4) anwenden zu können, müssen die Boxen auf bestimmte Niveaumengen aufgeteilt werden [8]. Diese Aufteilung wird *Levelizing* genannt und im folgenden Abschnitt definiert.

Hierfür sei $\widetilde{\Gamma_G^+}(M) = \{y \mid \exists x (x \in M \ \wedge \ y \in \Gamma_G^+(x))\}$ die Nachfolgermenge sowie $\widetilde{\Gamma_G^-}(M) = \{y \mid \exists x (x \in M \ \wedge \ y \in \Gamma_G^-(x))\}$ die Vorgängermenge von $M \subseteq X$ für einen beliebigen gerichteten Graphen $G = (X, \mathcal{R})$.

Definition 5.2 (Box-Levelizing, Box-Niveau-Menge)
Es sei $\mathcal{M} = (M_I, M_O, M_E, M_L, M_S, M_{\mathcal{R}})$ ein strukturelles Hardware-Modell mit zugehörigem bipartiten gerichteten Graphen $G(\mathcal{M}) = (M_B, M_S, M_{\mathcal{R}})$. Für $k \in \mathbb{N}$ wird die volle k-te Box-Niveaumenge $\widehat{\mathfrak{N}_k}(M_B) \subseteq M_B$ *mit $k \in \mathbb{N} \setminus \{0\}$ wie folgt rekursiv definiert:*

$$b \in \widehat{\mathfrak{N}_0}(M_B) :\Longleftrightarrow b \in M_L \vee \left(b \in M_E \Rightarrow \Gamma^-_{G(\mathcal{M})}(b) \subseteq M_I\right) \tag{5.1}$$

$$b \in \widehat{\mathfrak{N}_k}(M_B) :\Longleftrightarrow b \in M_L \vee \left(b \in M_E \Rightarrow \widetilde{\Gamma^-_{G(\mathcal{M})}}\left(\Gamma^-_{G(\mathcal{M})}(b)\right) \subseteq \bigcup_{l \in \{0,\ldots,k-1\}} \widehat{\mathfrak{N}_l}(M_B)\right). \tag{5.2}$$

Es sei $\mathfrak{N}_0(M_B) := \widehat{\mathfrak{N}_0}(M_B)$ die 0-te Box-Niveaumenge von M_B *und für $k \in \mathbb{N} \setminus \{0\}$ sei*

$$\mathfrak{N}_k(M_B) := \widehat{\mathfrak{N}_k}(M_B) \setminus \widehat{\mathfrak{N}_{k-1}}(M_B) \tag{5.3}$$

die k-te Box-Niveaumenge von M_B. *Die Menge $\mathfrak{L}(M_B) = \{\mathfrak{N}_k(M_B)\}_{k \in \mathbb{N}} \setminus \{\emptyset\}$ heißt* Levelizing von M_B.

Mit (5.1) ist gewährleistet, daß das Niveau 0 von den Latches, den Boxen ohne Eingänge und den direkt nur an globalen Inputs angeschlossenen Boxen gebildet wird. Der folgende Satz besagt, daß durch (5.2) und (5.3) der Forderung Rechnung getragen wird, daß die Eingangssignale einer Box aus einem Niveau größer 0 *nur* mit Boxen kleinerer Niveaus verbunden sein sollen. Vorher wird noch eine Folgerung angegeben, die die vollen Box-Niveaumengen näher charakterisiert.

Folgerung 5.2 (Isotonie)
Es gelten die Voraussetzungen von Definition 5.2. Dann ist $\left(\widehat{\mathfrak{N}_k}(M_B)\right)_{k\in\mathbb{N}}$ *eine* isotone Folge, *d.h., es gilt für* $k \in \mathbb{N}$ *die Beziehung* $\widehat{\mathfrak{N}_k}(M_B) \subseteq \widehat{\mathfrak{N}_{k+1}}(M_B)$.

Beweis: Sei $b \in \widehat{\mathfrak{N}_k}(M_B) \setminus M_L$. Wegen (5.2) gilt also

$$\widetilde{\Gamma^-_{G(M)}}\left(\Gamma^-_{G(M)}(b)\right) \subseteq \bigcup_{m\in\{0,\ldots,k-1\}} \widehat{\mathfrak{N}_m}(M_B) \subseteq \bigcup_{m\in\{0,\ldots,k\}} \widehat{\mathfrak{N}_m}(M_B).$$

Mit (5.2) heißt das aber $b \in \widehat{\mathfrak{N}_{k+1}}(M_B)$. Für $b \in M_L \cap \widehat{\mathfrak{N}_k}(M_B)$ ist $b \in \widehat{\mathfrak{N}_{k+1}}(M_B)$ mit (5.2) offensichtlich. Somit gilt $\widehat{\mathfrak{N}_k}(M_B) \subseteq \widehat{\mathfrak{N}_{k+1}}(M_B)$ wegen $M_B = M_L \cup M_E$. □

Der folgendende Satz verschärft die Folgerung dahingehend, daß aus ihm unmittelbar folgt, daß die vollen Box-Niveaumengen M_B vollständig ausschöpfen, d.h., M_B ist ihr Grenzwert. Wegen der Endlichkeit von M_B gibt es also ein $k \in \mathbb{N}$ mit $\widehat{\mathfrak{N}_k}(M_B) = M_B$.

Satz 5.1 (Niveau-Mengen-Zerlegung)
Sei $\mathcal{M} = (M_I, M_O, M_E, M_L, M_S, M_R)$ *ein strukturelles Hardware-Modell mit zugehörigem bipartiten gerichteten Graphen* $G(\mathcal{M}) = (M_B, M_S, M_R)$. *Das Levelizing* $\mathfrak{L}(M_B)$ *ist eine Zerlegung von* M_B*, d.h., es gilt*

$$M_B = \bigcup_{k\in\mathbb{N}} \mathfrak{N}_k(M_B)$$

und für $k, l \in \mathbb{N}$ *mit* $k \neq l$ *ist* $\mathfrak{N}_k(M_B) \cap \mathfrak{N}_l(M_B) = \emptyset$.

Beweis: (a.) O.B.d.A. seien $k, l \in \mathbb{N}$ mit $l > k$, also insbesondere $k \neq l$. Nun sei $b \in \mathfrak{N}_k(M_B)$. Wegen (5.3) gilt somit $b \in \widehat{\mathfrak{N}_k}(M_B)$. Damit folgt aus (5.3) aber auch, daß $b \notin \mathfrak{N}_l(M_B)$ gilt. Folglich ist $\mathfrak{N}_k(M_B) \cap \mathfrak{N}_l(M_B) = \emptyset$.

(b.) Trivialerweise gilt

$$\bigcup_{k\in\mathbb{N}} \mathfrak{N}_k(M_B) \subseteq M_B = M_L \cup M_E.$$

Sei $b \in M_L$. Dann folgt aus (5.1), daß $b \in \mathfrak{N}_0(M_B)$ gilt. Für $b \in M_E$ mit $\Gamma^-_{G(M)}(b) \subseteq M_I$ folgt aus (5.1), daß $b \in \mathfrak{N}_0(M_B)$ gilt. Daher sei nun $b_0 \in (M_B \setminus \mathfrak{N}_0(M_B)) \cap M_E$. Angenommen, es ist

$b_0 \notin \bigcup_{k \in \mathbb{N}} \mathfrak{N}_k(M_B)$. Dann gibt es kein $k \in \mathbb{N}$ mit $b_0 \in \mathfrak{N}_k(M_B)$ und $b_0 \in \widehat{\mathfrak{N}_k}(M_B)$. Damit gilt für $k \in \mathbb{N} \setminus \{0\}$ stets

$$\widetilde{\Gamma^-_{G(M)}}\left(\Gamma^-_{G(M)}(b_0)\right) \not\subseteq \bigcup_{l \in \{0,\ldots,k-1\}} \widehat{\mathfrak{N}_l}(M_B).$$

Nun sei $b_1 \in \widetilde{\Gamma^-_{G(M)}}\left(\Gamma^-_{G(M)}(b_0)\right)$, also insbesondere $b_1 \notin \bigcup_{l \in \{0,\ldots,k-1\}} \widehat{\mathfrak{N}_l}(M_B)$. Da dies für alle $k \in \mathbb{N} \setminus \{0\}$ gilt, ist dann

$$b_1 \notin \bigcup_{k \in \mathbb{N}} \widehat{\mathfrak{N}_k}(M_B).$$

Wiederholt man dieses Verfahren rekursiv, so erhält man eine Folge $b_0, b_1, b_2, \ldots$ aus $M_E \setminus \bigcup_{k \in \mathbb{N}} \widehat{\mathfrak{N}_k}(M_B)$. Wegen der Endlichkeit von M_E muß die Folge entweder abbrechen oder periodisch werden (d.h., sie beschreibt Kreise in $G(\mathcal{M})$). Im Fall der Periodizität folgt aus (iii) von Definition 5.1, daß ein $l \in \mathbb{N}$ derart existiert, daß $b_l \in \mathfrak{N}_0(M_B)$ gilt, da oben schon gezeigt wurde, daß die Elemente von M_L in $\mathfrak{N}_0(M_B)$ liegen. Das hieße aber $b_l \in \bigcup_{k \in \mathbb{N}} \widehat{\mathfrak{N}_k}(M_B)$, was nicht sein kann. Somit muß die Folge abbrechen.

Es gibt also ein $m \in \mathbb{N}$ mit $\widetilde{\Gamma^-_{G(M)}}\left(\Gamma^-_{G(M)}(b_k)\right) = \emptyset$, d.h. $\Gamma^-_{G(M)}(b_m) \subseteq M_I$ wegen (i) von Definition 5.1. Das bedeutet aber mit (5.1) dann $b_m \in \widehat{\mathfrak{N}_0}(M_B) \subseteq \bigcup_{k \in \mathbb{N}} \widehat{\mathfrak{N}_k}(M_B)$, was nicht sein kann. Deshalb muß die Annahme $b_0 \notin \bigcup_{k \in \mathbb{N}} \mathfrak{N}_k(M_B)$ falsch sein.

Folglich ist $M_B \subseteq \bigcup_{k \in \mathbb{N}} \mathfrak{N}_k(M_B)$, also $M_B = \bigcup_{k \in \mathbb{N}} \mathfrak{N}_k(M_B)$. □

In Teil (b.) des Beweises wird wesentlich von der Endlichkeit des Graphen und der Tatsache Gebrauch gemacht, daß Latche in jedem Kreis (Zyklus) des Graphen enthalten sind. Der Satz von der Niveau-Mengen-Zerlegung liefert die mathematische Entsprechung der anschaulichen Vorstellung von der Einteilung der Boxen auf disjunkte Niveau-Mengen (*Level*), das gemeinhin als *Levelizing* bezeichnet wird.

Die nächste Definition nutzt das Box-Levelizing für eine analoge Begriffsbildung für die Signale, die sich wiederum anschaulich begründen läßt.

Definition 5.3 (Signal-Levelizing, Signal-Niveau-Menge)
Es sei $\mathcal{M} = (M_I, M_O, M_E, M_L, M_S, M_R)$ ein strukturelles Hardware-Modell mit zugehörigem bipartiten gerichteten Graphen $G(\mathcal{M}) = (M_B, M_S, M_R)$. Für $k \in \mathbb{N}$ wird die volle k-te Signal-Niveaumenge $\widehat{\mathfrak{N}_k}(M_S) \subseteq M_S$ *mit $k \in \mathbb{N} \setminus \{0\}$ wie folgt rekursiv definiert:*

$$s \in \widehat{\mathfrak{N}_0}(M_S) :\Longleftrightarrow s \in M_I \tag{5.4}$$

$$s \in \widehat{\mathfrak{N}_k}(M_S) :\Longleftrightarrow \Gamma^-_{G(M)}(s) \subseteq \widehat{\mathfrak{N}_{k-1}}(M_B). \tag{5.5}$$

Es sei $\mathfrak{N}_0(M_S) := \widehat{\mathfrak{N}_0}(M_S)$ *die* 0-te Signal-Niveaumenge von M_S *und für* $k \in \mathbb{N} \setminus \{0\}$ *sei*

$$\mathfrak{N}_k(M_S) := \widehat{\mathfrak{N}_k}(M_S) \setminus \widehat{\mathfrak{N}_{k-1}}(M_S) \tag{5.6}$$

die k-te Signal-Niveaumenge von M_S. *Die Menge* $\mathfrak{L}(M_S) = \{\mathfrak{N}_k(M_S)\}_{k\in\mathbb{N}} \setminus \{\emptyset\}$ *heißt* Levelizing von M_S.

Der Nutzen obiger Definition ergibt sich aus einer einfachen Folgerung des Satzes von der Niveau-Mengen-Zerlegung.

Folgerung 5.3 (Signal-Mengen-Zerlegung)
Sei $M = (M_I, M_O, M_E, M_L, M_S, M_R)$ *ein strukturelles Hardware-Modell mit zugehörigem bipartiten gerichteten Graphen* $G(M) = (M_B, M_S, M_R)$. *Das Levelizing* $\mathfrak{L}(M_S)$ *ist eine Zerlegung von* M_S*, d.h., es gilt*

$$M_S = \bigcup_{k\in\mathbb{N}} \mathfrak{N}_k(M_S)$$

und für $k,l \in \mathbb{N}$ *mit* $k \neq l$ *ist* $\mathfrak{N}_k(M_S) \cap \mathfrak{N}_l(M_S) = \emptyset$.

Beweis: (a.) O.B.d.A. seien $k,l \in \mathbb{N}$ mit $l > k$, also insbesondere $k \neq l$. Nun sei $s \in \mathfrak{N}_k(M_S)$. Wegen (5.6) gilt somit $s \in \widehat{\mathfrak{N}_k}(M_S)$. Damit folgt aus (5.6) aber auch, daß $s \notin \mathfrak{N}_l(M_S)$ gilt. Folglich ist $\mathfrak{N}_k(M_S) \cap \mathfrak{N}_l(M_S) = \emptyset$.

(b.) Trivialerweise gilt

$$\bigcup_{k\in\mathbb{N}} \mathfrak{N}_k(M_S) \subseteq M_S.$$

Sei daher $s \in M_S$. Gilt $\Gamma^-_{G(M)}(s) = \emptyset$, so folgt aus (5.5) und (5.6) dann $s \in \bigcup_{k\in\mathbb{N}} \mathfrak{N}_k(M_S)$. Nun sei $\Gamma^-_{G(M)}(s) \neq \emptyset$ und $b \in \Gamma^-_{G(M)}(s)$. Nach dem Satz von der Niveau-Mengen-Zerlegung gibt es dann genau ein $l \in \mathbb{N}$ mit $b \in \mathfrak{N}_l(M_B)$. Aufgrund der Endlichkeit aller hier betrachteten Mengen gibt es deshalb für alle $b \in \Gamma^-_{G(M)}(s)$ ein $m \in \mathbb{N}$, so daß $b \in \widehat{\mathfrak{N}_m}(M_B)$ gilt. Somit gilt $\Gamma^-_{G(M)}(s) \subseteq \widehat{\mathfrak{N}_m}(M_B)$. Mit (5.5) heißt das aber $s \in \widehat{\mathfrak{N}_{m+1}}(M_S)$. Wegen (5.6) existiert daher ein $l' \in \{0,\ldots,m+1\}$ mit $s \in \mathfrak{N}_{l'}(M_S)$. Insgesamt ergibt sich damit

$$M_S \subseteq \bigcup_{k\in\mathbb{N}} \mathfrak{N}_k(M_S),$$

woraus die Behauptung folgt. □

5.3. Das funktionelle Hardware-Modell

5.3.1. Definition des funktionellen Hardware-Modells

Die funktionelle Beschreibung des Hardware-Modells soll durch geeignete Bewertungsfunktionen des zugehörigen bipartiten gerichteten Graphen erreicht werden. Seien M_B, M_S und M_R nichtleere Mengen mit $M_R \subseteq (M_B \times M_S) \cup (M_S \times M_B)$ und $M_B \cap M_S = \emptyset$. Weiterhin ist $W = \{0,1\}$ und

$$\mu : M_R \to W \quad (5.7)$$

$$\nu : (M_B \cup M_S) \to W \cup W^{(\mathbb{N})} \quad (5.8)$$

$$\nu_B : M_B \to W^{(\mathbb{N})} \quad (5.9)$$

$$\hat{\nu}_B : \left[W^{(\mathbb{N})}\right] \times M_S \to W \quad (5.10)$$

$$\nu_S : M_S \to W \quad (5.11)$$

$$\hat{\nu}_S : M_B \to W^{(\mathbb{N})} \quad (5.12)$$

$$\hat{\nu}_I : M_I \to W \quad (5.13)$$

seien eindeutige Abbildungen sowie

$$(5.14)$$

$$\iota : M_S \to \mathbb{N} \quad (5.15)$$

eine injektive Abbildung. Dabei seien $\hat{\nu}_B$ und $\hat{\nu}_I$ beliebig und für $b \in M_B$, $s \in M_S$ und $i \in \mathbb{N}$ sowie für den gerichteten bipartiten Graphen $G := (M_B, M_S, M_R)$ gelte[31]

$$(\hat{\nu}_S(b))_i = \begin{cases} \mu(s,b) & \text{falls } s \in \Gamma_G^-(b) \text{ und } \iota(s) = i \\ 0 & \text{sonst} \end{cases} \quad (5.16)$$

$$(\nu_B(b))_i = \begin{cases} \hat{\nu}_B(\hat{\nu}_S(b), s) & \text{falls } s \in \Gamma_G^+(b) \text{ und } \iota(s) = i \\ 0 & \text{sonst} \end{cases} \quad (5.17)$$

$$\nu_S(s) = \begin{cases} \hat{\nu}_I(s) & \text{falls } s \in M_I \\ \bigvee_{b \in \Gamma_G^-(s)} \mu(b,s) & \text{sonst} \end{cases} \quad (5.18)$$

$$\nu|_{M_B} = \nu_B \quad (5.19)$$

$$\nu|_{M_S} = \nu_S \quad (5.20)$$

$$\mu(s,b) = \nu_S(s) \quad (5.21)$$

$$\mu(b,s) = (\nu_B(b))_{\iota(s)} \,. \quad (5.22)$$

[31] hierbei werde mit $(\cdot)_i$ der Funktionswert einer Folge an der Stelle i bezeichnet, d.h., es ist die i-te Komponente

Offensichtlich (wegen der Endlichkeit des Graphen) werden durch (5.16) und (5.17) und damit auch durch (5.19) Elemente aus $W^{(\mathbb{N})}$ definiert. Mit den obigen Bewertungen wird der funktionelle Aspekt eines Hardware-Modells beschrieben, wie im Anschluß an die folgende, dies festhaltende, Definition erläutert wird.

Definition 5.4 (funktionelles Hardware-Modell fHWM)
Sei $M = (M_I, M_O, M_E, M_L, M_S, M_R)$ ein strukturelles Hardware-Modell mit zugehörigem gerichteten bipartiten Graphen $G := G(M) = (M_B, M_S, M_R)$. Weiter sei W ein endlicher Abschnitt von $\mathbb{N}$. Dann heißt $M_F = (M, \mu, \nu)$ funktionelles Hardware-Modell *von M, wenn die Gleichungen (5.7) – (5.22) für $b \in M_B$, $s \in M_S$ sowie $i \in \mathbb{N}$ erfüllt sind. W ist die* Menge der M_F zugrundegelegten Wahrheitswerte.

Im allgemeinen wird $W = \{0, 1\}$ eine ausreichende Festlegung sein, sofern man nicht Simulatoren für mehrwertige Logik betrachtet. TEXSIM beispielsweise kann mehrwertige Zustände nicht propagieren. Die Disjunktion in Gleichung (5.18) bedarf bei $|W| \neq 2$ natürlich einer gesonderten Beachtung. Für die Betrachtungen hier sei der Einfachheit halber stets $|W| = 2$ angenommen (auch weil dies für TEXSIM ausreichend ist). Die Klasse aller fHWM eines sHWM ist eine (endliche) Menge, weil ein sHWM eine Struktur aus endlichen Mengen ist.

5.3.2. Zur Interpretation des funktionellen Hardware-Modells

Die in Abschnitt 5.1 erwähnte Modifikation der Definition des Hardware-Modells aus [8] machte sich wegen Gleichung (5.17) notwendig. Mit (5.19) und (5.20) wird ν zu einer Knotenbewertung. Die eineindeutige Indizierung (5.15) der Signale dient nur als Hilfskonstrukt für (5.16),(5.17) und (5.22) und kann daher völlig beliebig sein.

Der Grundgedanke des fHWM besteht darin, daß Eingangstupeln einer Box durch ihre *Transferfunktion* (Schaltfunktion) Ausgangstupel zugeordnet werden, abhängig davon, wie die Eingänge belegt sind. Da Eingangs- und Ausgangstupel von Box zu Box völlig unterschiedliche Dimensionen haben können, wurden abzählbar unendliche Tupel verwendet, die nur an endlich vielen Positionen von Null verschiedene Komponenten haben (die Elemente von $W^{(\mathbb{N})}$).

Die Transferfunktion einer Box wird in (5.17) definiert. Mit $\hat{\nu}_S(b)$ wird dabei auf die Eingänge der Box b zurückgegriffen, deren Belegungen durch (5.16) gegeben sind. Dort werden wiederum die entsprechenden Netzbewertungen ausgenutzt, die in (5.21) festgelegt werden: Die Bewertungen der Kanten (Netze) von Signalen zu Boxen ergeben sich einfach aus der des jeweiligen Signales. Eine Bewertung eines Signales ist die Disjunktion der Bewertungen der Netze, die von Boxen an dieses Signal gehen, falls es kein globaler Eingang ist. Das entspricht unmittelbar der Interpretation eines Signals als Kontaktpunkt von Leiterbahnen. Die dafür nötige Kantenbewertung wird mit (5.22) gerade durch die entsprechenden Transferfunktionen der jeweiligen Boxen gegeben. Daß die Definition damit überhaupt sinnvoll ist, sichert die Synchronität der Schaltung, d.h. (iii) von Definition 5.1.

Die Besonderheit der globalen Eingänge, über deren Belegung letztendlich das Verhalten der gesamten Schaltung kontrollierbar ist, spiegelt sich in (5.21) durch die Verwendung des beliebigen $\hat{v}_I$ wider.

5.3.3. Hardware-Zustände und das funktionelle Hardware-Modell

Mit dem strukturellen und dem funktionellen Hardware-Modell sind zwei wesentliche Aspekte einer synchronen Schaltung auf der Abstraktionsebene des Gate Level und des Register Transfer Level formal erfaßt. Das Verhalten einer Schaltung hängt von den Zuständen ihrer Elemente ab. Dies gilt vor allem für Latche, deren Einfluß auf die Funktion einer Schaltung, nämlich die Signalverzögerung um einen Takt, durch das fHWM noch nicht zwingend abgedeckt ist. Zustände lassen sich innerhalb des fHWMs als Belegungen der Boxen und Signale darstellen.

Üblicherweise wird ein realer Schaltkreis initialisiert. Bei Mikroprozessoren beispielsweise werden dafür bestimmte Signale an die Input-Pins angelegt. Die formale Beschreibung hierfür ist schon im fHWM enthalten (das regeln $\hat{v}_I$ von Definition 5.4 sowie die Latch-Transferfunktionen).

Jedoch ist es in der Logiksimulation aus vielerlei Gründen auch wichtig (vgl. Abschnitt 3), ganz konkrete Zustände von Elementen inmitten der Schaltung während des Simulationsprozesses setzen zu können. Derartige Setzungen nennt man *Stimuli*. So werden z.B. Test Cases direkt in bestimmte Arrays geschrieben. Diese Situation kann innerhalb des fHWMs formalisiert werden, indem man auch die Stimuli, die bei einer konkreten Simulation verwendet werden, über die Belegungsfunktionen beschreibt.

Somit deckt sich die hier verwendete Bezeichnung (funktionelles Hardware-)Modell nicht mit der in der Praxis (vgl. Abschnitt 2.1) üblichen. Dort wird dieser Begriff nur für die, die Hardware verkörpernde, Datenstruktur verwendet. Hier wird die Bezeichnung natürlich auch im Sinne eines mathematischen *Modells* gebraucht. Es ist daher nicht unangebracht, statt vom fHWM von einem Simulations-Modell zu sprechen.

Damit ist nur noch die Frage der adäquaten Darstellung der Latche offen. Ihr funktionelles Verhalten ist davon abhängig, wie sie selbst belegt sind. Die an den Latchen als Eingang anliegenden *clock*-Signale sind dabei einfach normale Signale aus M_S, die in den Belegungsfunktionen derart auftreten, daß sich eine Latch-Ausgangsbelegung erst dann ändert, wenn ein zugehöriges clock-Signal mit 1 belegt ist. Insbesondere wird dabei auch nur auf Eingänge zugegriffen, wenn das clock-Signal gesetzt ist, so daß nicht auf undefinierte Signale zugegriffen wird.

Das ist eine Forderung an das fHWM, die nicht formalisiert vorliegt (einerseits aus Zeitgründen, andererseits um die Darstellung nicht noch mehr auszudehnen), die aber für alle formalen Betrachtungen als gegeben angenommen wird!

Insgesamt läßt sich daher feststellen, daß mit dem funktionellen Hardware-Modell die Simulation synchroner Schaltungen auf der Abstraktionsebene der Logik, d.h. auf Gate- und Registerebene, formal beschreibbar ist. Diese Aussage wird im nächsten Abschnitt mit der Formulierung eines konkreten Simulationsalgorithmus, dem Clock-Cycle-Algorithmus, noch untermauert.

5.4. Der sequentielle Clock-Cycle-Algorithmus

Definition 5.5 (Evaluation, Box-Evaluation, Signal-Evaluation)
Es sei $M_F = (M, \mu, \nu)$ *ein funktionelles Hardware-Modell des strukturellen Hardware-Modells* $M = (M_I, M_O, M_E, M_L, M_S, M_R)$. *Für* $b \in M_B$ *heißt dann* $(b, \nu(b))$ *die* Box-Evaluation von b in M_F *und für* $s \in M_S$ *heißt dann* $(s, \nu(s)$ *die* Signal-Evaluation von s in M_F. *Die Menge*

$$E_{M_F}(\mathfrak{M}) = \{(b, \nu(b)) \mid b \in \mathfrak{M}\}$$

ist die Box-Evaluation von $\mathfrak{M}$ in M_F *mit* $\emptyset \neq \mathfrak{M} \subseteq M_B$ *und die Menge*

$$E_{M_F}(\mathfrak{M}) = \{(s, \nu(s)) \mid s \in \mathfrak{M}\}$$

ist die Signal-Evaluation von $\mathfrak{M}$ in M_F *mit* $\emptyset \neq \mathfrak{M} \subseteq M_S$. $E_{M_F}(M_B)$ *ist die* Box-Evaluation von M_F *und* $E(M_S)$ *ist die* Signal-Evaluation von M_F. *Als* Evaluation von M_F *wird die Menge* $E(M_F) = E(M_B) \cup E(M_S)$ *bezeichnet.*

Die Box- bzw. die Signal-Evaluation repräsentiert gewissermaßen den Zustand eines Schaltungselementes und mithin die Evaluation des gesamten funktionellen Hardware-Modells den Zustand der Schaltung zu einem bestimmten Zeitpunkt.

Definition 5.6 (sequentieller Clock-Cycle-Algorithmus)
Es gelten die Voraussetzungen von Definition 5.5 und es sei $n \in \mathbb{N}$. *Das Verfahren in Abbildung 31 heißt dann der sequentielle* Clock-Cycle-Algorithmus. *Mit* $\mathfrak{C}_{seq}(M_F, n)$ *wird sein Resultat in Abhängigkeit vom Parameter* n *bezeichnet.*

Daß durch Abbildung 31 tatsächlich ein Algorithmus definiert wird, ist (wegen der Endlichkeit der Mengen) klar, da alle Schritte offensichtlich konstruktiv sind und das Verfahren terminiert (vgl. Bemerkung vor dem Satz von der Niveau-Mengen-Zerlegung).

Durch die Anwendung des Algorithmus auf ein fHWM wird die funktionelle Logiksimulation mathematisch modelliert. Konkret wird damit der Schritt CLOCK des ALTER-CLOCK-RETRIEVE Schemas (vgl. Abschnitt 3) beschrieben, wobei der Parameter n die Anzahl der Zyklen angibt.

Satz 5.2
Es gelten die Voraussetzungen von Definition 5.5. Dann gilt $E_{M_F}(M_F) = \mathfrak{C}_{seq}(M_F, 1)$.

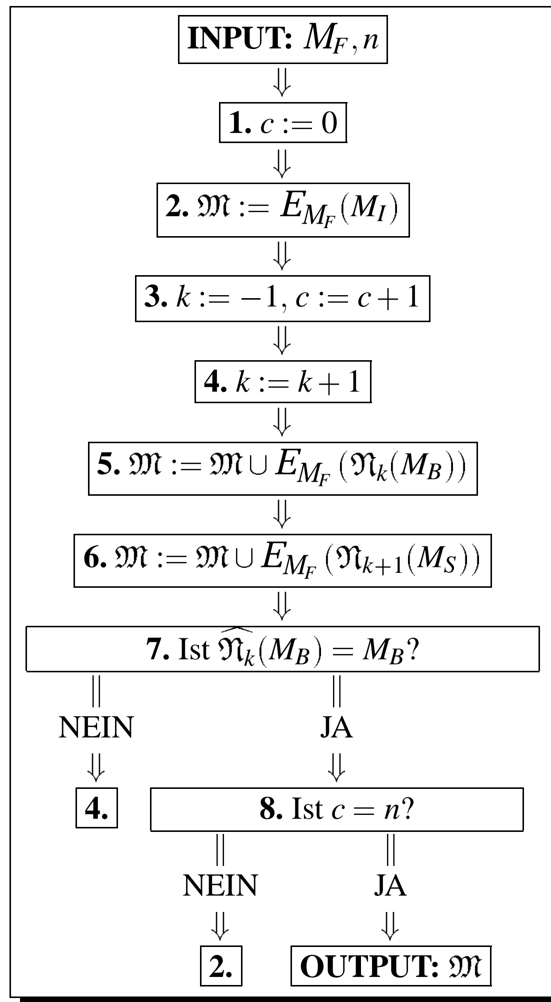


Abbildung 31: Der sequentielle Clock-Cycle-Algorithmus

Beweis: Im Schritt **2.** werden die Signal-Evaluationen der Elemente von M_I gebildet. Dann liefert der Algorithmus wegen **5.** offensichtlich die Box-Evaluation für die Boxen aus $\mathfrak{N}_0(M_B)$, da dafür aufgrund von (5.16) – (5.22) keine vorherigen Box-Evaluationen für die einzelnen Box-Evaluationen — dies trifft auch auf die Elemente von M_L zu — notwendig sind (vgl. (5.1) von Definition 5.2 und die sich an die Definition anschließende Bemerkung). Wegen **6.** liefert der Algorithmus für die Signale aus $\mathfrak{N}_1(M_S)$ dann die Signal-Evaluation.

Liefere der Algorithmus bei $k \in \mathbb{N}$ nun für die Boxen aus den Mengen $\mathfrak{N}_0(M_B), \ldots, \mathfrak{N}_k(M_B)$ die Box-Evaluation und für die Signale aus $\mathfrak{N}_0(M_S), \ldots, \mathfrak{N}_{k+1}(M_S)$ die Signal-Evaluation und gelte $\widehat{\mathfrak{N}_k}(M_B) \neq M_B$. Aufgrund von Folgerung 5.2 sind die 0- bis $(k-1)$-ten Box-Niveau-Mengen Teilmengen von $\widehat{\mathfrak{N}_k}(M_B)$ und wegen (5.3) von Definition 5.2 ist die k-te Box-Niveaumenge sowieso eine Teilmenge von $\widehat{\mathfrak{N}_k}(M_B)$. Dann liefert der Algorithmus wegen **5.** die Box-Evaluation von $\mathfrak{N}_{k+1}(M_B)$, weil mit (5.16) – (5.22) dafür nur noch Box-Evaluationen aus $\widehat{\mathfrak{N}_k}(M_B)$ für die einzelnen Box-Evaluationen notwendig sind. Wegen **6.** liefert der Algorithmus für die Signale aus $\mathfrak{N}_{k+2}(M_S)$ dann die Signal-Evaluation.

Da nach dem Satz von der Niveau-Mengen-Zerlegung die Box-Menge M_B die Vereinigung der einzelnen Box-Niveaus ist, liefert der Algorithmus somit die Box-Evaluation von M_F. Nach der Folgerung über die Signal-Mengen-Zerlegung liefert er ebenso die Signal-Evaluation von M_F. Damit ist die Behauptung gezeigt. □

Die Grundidee des sequentiellen Clock-Cycle-Algorithmus besteht darin, die einzelnen Schaltungselemente erst dann zu evaluieren, wenn alle speisenden Elemente schon evaluiert wurden. Dafür bietet sich in natürlicher Weise die Reihenfolge gemäß des Levelizing an. Die Evaluierung geschieht so lange, bis man wieder an die Latche, also das 0-te Niveau gelangt — dann ist ein Zyklus abgeschlossen. Die im nächsten Zyklus benötigten Eingangsbelegungen der Latche hat man so schon im aktuellen ermittelt.

5.5. Die Cone-Darstellung

In [8] werden die Begriffe *fan in-Cone* und *fan out-Cone* definiert, die die Boxen charakterisieren, die die Evaluation einer bestimmten Box beeinflussen, bzw. die Boxen, deren Evaluation von einer bestimmten Box beeinflußt wird. Diese Definitionen werden jetzt an die Darstellung in der vorliegenden Arbeit angepaßt.

Definition 5.7 (*fan in*-Cone, *fan out*-Cone)
Sei $M = (M_I, M_O, M_E, M_L, M_S, M_R)$ ein strukturelles Hardware-Modell mit zugehörigem gerichteten bipartiten Graphen $G(M) = (M_B, M_S, M_R)$. Der fan in-Cone $co_I(b)$ *eines Elementes $b \in M_B \cup M_O$ wird durch die folgenden Bedingungen (i) und (ii) definiert. Der* fan out-Cone $co_O(b)$ *eines Elementes $b \in M_B \cup M_I$ wird durch die folgenden Bedingungen (iii) und (iv) definiert.*

(i) Es ist $b \in co_I(b)$.

(ii) Für $b' \in M_E$ gilt $\widetilde{\Gamma^{+}_{G(M)}}\left(\Gamma^{+}_{G(M)}(b')\right) \cap co_I(b) \neq \emptyset \Longrightarrow b' \in co_I(b)$.
Zudem gilt $\emptyset \neq \Gamma^{+}_{G(M)}(b') \cap \{b\} \subseteq M_O \Longrightarrow b' \in co_I(b)$.

(iii) Es ist $b \in co_O(b)$.

(iv) Für $b' \in M_E$ gilt $\widetilde{\Gamma^{-}_{G(M)}}\left(\Gamma^{-}_{G(M)}(b')\right) \cap co_O(b) \neq \emptyset \Longrightarrow b' \in co_O(b)$.
Zudem gilt $\emptyset \neq \Gamma^{-}_{G(M)}(b') \cap \{b\} \subseteq M_I \Longrightarrow b' \in co_O(b)$.

Die Anpassung der Definitionen erfolgte durch die zweiten Teile von (ii) bzw. (iv), die sich durch die Änderung (s. Abschnitt 5.1) des zugrundegelegten Hardware-Modells erforderlich machten. Es wird damit ausgedrückt, daß, falls b ein globaler Ausgang bzw. Eingang ist, die direkt an b angeschlossenen Boxen im *fan in-* bzw. *fan out*-Cone von b liegen.

Da ein gewisser Box-Typ, neben den bislang bekannten Latchen, für die cone-basierte Modellpartitionierung eine besondere Rolle spielt, soll er zunächst definiert werden.

Definition 5.8 (Assert-Box, Sending-Unit-Box)
Es gelten die Voraussetzungen von Definition 5.7. Gilt für $b \in M_E$ dann $\Gamma^{+}_{G(M)}(b) = \emptyset$, so wird b Assert-Box *genannt. Gilt dagegen für $b \in M_E$ dann $\Gamma^{-}_{G(M)}(b) = \emptyset$, so wird b* Sending-Unit-Box *genannt.*

Assert-Boxen sind also solche Boxen, die keine Ausgänge haben; Sending-Unit-Boxen haben dagegen keine Eingänge. In einer realen Schaltung werden diese wohl kaum vorkommen — für Simulationszwecke können sie aber gewisse (Meta-)Funktionen, die über die eigentliche Clock-Cycle-Simulation hinausgehen, übernehmen. Die Assert-Boxen können der Überprüfung von

Bedingungen (Assertion Violations) dienen. Sending-Unit-Boxen könnten nur zu bestimmten Zeitpunkten, also konkreten Cycles, ihre Wertbelegung ändern, und so Updates von Latchen anstelle der eigentlichen clock-Signale erzwingen.

Wegen der zyklusbegrenzenden Funktion der globalen Ausgänge, der Assert-Boxen und der Latche werden daher in [8] deren *fan in*-Cones betrachtet.[32] Die folgende Definition (nach [8]) ist dadurch motiviert.

Definition 5.9 (Cone, Cone-Menge, Cone-Kopf)
Es gelten die Voraussetzungen von Definition 5.7 und M_A sei die Menge der zugehörigen Assert-Boxen. Ein Cone $co(x)$ *ist ein* fan in-*Cone* $co_I(x)$ *mit* $x \in M_O \cup M_A \cup M_L$. *Die zum strukturellen Hardware-Modell* $\mathcal{M}$ *gehörige* Cone-Menge $Co(\mathcal{M})$ *ist die Menge der Cones, d.h.*

$$Co(\mathcal{M}) = \{c \mid \exists x (x \in M_O \cup M_A \cup M_L \Longrightarrow c = co(x))\}.$$

Die Elemente von $M_O \cup M_A \cup M_L$ *werden* Cone-Köpfe *genannt.*

Der nächste Satz zeigt die Bedeutung der Cone-Menge: Jede Box ist in mindestens einem Cone enthalten.

Satz 5.3 (Cone-Darstellung)
Es gelten die Voraussetzungen von Definition 5.9. Dann gilt $M_B = \bigcup_{c \in Co(\mathcal{M})} c$.

Beweis: Offensichtlich gilt $\bigcup_{c \in Co(\mathcal{M})} c \subseteq M_B$. Zu zeigen ist also $M_B \subseteq \bigcup_{c \in Co(\mathcal{M})} c$. Sei daher $b_0 \in M_B$ und $\mathfrak{M}_0 := \widetilde{\Gamma^+_{G(\mathcal{M})}}\left(\Gamma^+_{G(\mathcal{M})}(b_0)\right)$. Ist $\mathfrak{M}_0 = \emptyset$, so gilt $\mathfrak{M} := \Gamma^+_{G(\mathcal{M})}(b_0) \subseteq M_O$. Ist $\mathfrak{M} = \emptyset$, so ist $b_0 \in M_A$, also $co(b_0) \in Co(\mathcal{M})$. Folglich ist $b_0 \in \bigcup_{c \in Co(\mathcal{M})} c$. Daher sei $s \in \mathfrak{M}$. Dann gilt $b_0 \in co(s)$, also $b_0 \in \bigcup_{c \in Co(\mathcal{M})} c$. Daher gelte jetzt $\mathfrak{M}_0 \neq \emptyset$ und $b_1 \in \mathfrak{M}_0$. Ist $b_1 \in M_L$, so gilt $b_0 \in co(b_1)$ und mithin $b_0 \in \bigcup_{c \in Co(\mathcal{M})} c$. Nun sei $b_1 \notin M_L$ und $\mathfrak{M}_1 := \widetilde{\Gamma^+_{G(\mathcal{M})}}\left(\Gamma^+_{G(\mathcal{M})}(b_1)\right)$. Für $\mathfrak{M}_1 = \emptyset$ ergibt sich wie eben $b_1 \in \bigcup_{c \in Co(\mathcal{M})} c$. Bei $\mathfrak{M}_1 \neq \emptyset$ sei $b_2 \in \mathfrak{M}_1$.

Auf diese Weise erhält man eine Folge $b_0, b_1, \ldots$. Wegen (iii) von Definition 5.1 muß es aber ein $k \in \mathbb{N}$ derart geben, daß $b_k \in M_L$ gilt. Somit muß die Folge abbrechen, und es gilt $b_0 \in co(b_k)$, also $b_0 \in \bigcup_{c \in Co(\mathcal{M})} c$. Daraus folgt die Behauptung. □

[32] ausgenommen die Assert-Boxen

5.6. Cone-basierte Modellpartitionierung

Die Modellpartionierung gemäß [8] ist die Grundlage der in dieser Arbeit vorgestellten praktischen Umsetzung der Parallelisierung der Logiksimulation am speziellen Beispiel von TEXSIM.

Mit der folgenden Definition (nach [8]) wird die Partitionierung auf die Grundlage des Cone-Begriffs (Abschnitt 5.5) gestellt.

Definition 5.10 (Partitionierung, Partition)
Es seien U und V nichtleere Mengen. Dann ist eine eindeutige Abbildung $\Phi : U \to V$ eine Partitionierung von U bezüglich V. *Eine* Partition Ψ_Φ von U bezüglich Φ *ist die Menge der Fasern der Partitionierung Φ über V ohne die leere Menge, d.h., es ist*

$$\Psi_\Phi = \left\{\Phi^{-1}(v) \mid v \in V\right\} \setminus \{\emptyset\} = \left\{\Phi^{-1}(v) \mid v \in \Phi(U)\right\}.$$

Durch eine Partitionierung ergibt sich in natürlicher Weise eine (nicht notwendig disjunkte) Aufteilung der Hardware in Teil-Blöcke. Die nächsten Definitionen formalisieren die Aufteilung — zunächst den strukturellen Aspekt und danach den funktionellen.

Definition 5.11 (struktureller Modellblock)
Es sei $\mathcal{M} = (M_I, M_O, M_E, M_L, M_S, M_R)$ ein strukturelles Hardware-Modell mit zugehörigem bipartiten Graphen $G(\mathcal{M}) = (M_B, M_S, M_R)$. Weiterhin sei Φ eine surjektive[33] *Partitionierung von $Co(\mathcal{M})$ bezüglich einer nichtleeren endlichen Menge $\mathfrak{B}$ mit $\Psi_\Phi = \left\{B_1, \ldots, B_{m_b}\right\}$, wobei $|\mathfrak{B}| = m_b$ gelte. Für $k \in \{1, \ldots, m_b\}$ sei $\overline{B_k} = \bigcup_{c \in B_k} c$ sowie $M_{E,k} = M_E \cap \overline{B_k}$ und $M_{L,k} = M_L \cap \overline{B_k}$. Weiterhin seien*

$$M_{S,k} = \left\{s \mid s \in M_S \wedge \left(\Gamma^-_{G(M)}(s) \cup \Gamma^+_{G(M)}(s)\right) \cap \overline{B_k} \neq \emptyset\right\}$$

sowie $M_{R,k} = M_R \cap \left((M_{B,k} \times M_{S,k}) \cup (M_{S,k} \times M_{B,k})\right)$. Mit $G_k(\mathcal{M}) = (M_{B,k}, M_{S,k}, M_{R,k})$, wobei $M_{B,k} = M_{E,k} \cup M_{L,k}$ gilt, sei dann

$$M_{I,k} = \left\{s \mid s \in M_S \ \wedge \ \Gamma^-_{G_k(M)}(s) = \emptyset\right\}$$

und

$$M_{O,k} = \left\{s \mid s \in M_S \ \wedge \ \Gamma^+_{G_k(M)}(s) = \emptyset\right\} .$$

Die strukturellen Hardware-Modelle $\mathcal{M}_k = (M_{I,k}, M_{O,k}, M_{E,k}, M_{L,k}, M_{S,k}, M_{R,k})$ heißen dann die strukturellen Modellblöcke von $\mathcal{M}$ bezüglich Φ *und die jeweiligen* zugehörigen bipartiten gerichteten Graphen *sind die Graphen $G_k(\mathcal{M}) = (M_{B,k}, M_{S,k}, M_{R,k})$.*

[33] dann ist $\emptyset \notin \Psi_\Phi$

Bei $M_{R,k}$ werden nur diejenigen Verbindungen von Boxen und Signalen aus $G(M)$ übernommen, die Boxen und Signale aus $M_{B,k}$ und $M_{S,k}$ betreffen. So werden neue Graphen $G_k(M)$ erhalten, die aufgrund der Festlegung von $M_{I,k}$ und $M_{O,k}$ offensichtlich die Bedingungen (i) – (iii) aus Definition 5.1 erfüllen, weshalb mit den M_k tatsächlich strukturelle Hardware-Modelle vorliegen.

Durch die nächste Folgerung wird klar, daß es sich bei den strukturellen Modellblöcken tatsächlich um eine Aufteilung des Hardware-Modells handelt.

Folgerung 5.4
Es gelten die Voraussetzungen von Definition 5.11. Dann gilt $M_B = \bigcup_{k=1}^{m_b} M_{B,k}$, $M_S = \bigcup_{k=1}^{m_b} M_{S,k}$ *und* $M_R = \bigcup_{k=1}^{m_b} M_{R,k}$.

Beweis: Es gilt wegen Definition 5.11 und dem Satz über die Cone-Darstellung

$$\begin{aligned}\bigcup_{k=1}^{m_b} M_{B,k} &= \bigcup_{k=1}^{m_b} [M_{E,k} \cup M_{L,k}] = \bigcup_{k=1}^{m_b} [(M_E \cap \overline{B_k}) \cup (M_L \cap \overline{B_k})] = \bigcup_{k=1}^{m_b} [(M_E \cup M_L) \cap \overline{B_k}] \\ &= (M_E \cup M_L) \cap \bigcup_{k=1}^{m_b} \overline{B_k} = M_B \cap \bigcup_{k=1}^{m_b} \overline{B_k} = M_B \cap \bigcup_{k=1}^{m_b} \bigcup_{c \in B_k} c = M_B \cap \bigcup_{c \in \bigcup_{k=1}^{m_b} B_k} c \\ &= M_B \cap \bigcup_{c \in Co(M)} c = M_B \cap M_B = M_B.\end{aligned}$$

Mit Folgerung 5.1 ergibt sich zusätzlich

$$\begin{aligned}\bigcup_{k=1}^{m_b} M_{S,k} &= \bigcup_{k=1}^{m_b} \left\{ s |\ s \in M_S \wedge \left(\Gamma^-_{G(M)}(s) \cup \Gamma^+_{G(M)}(s) \right) \cap \overline{B_k} \neq \emptyset \right\} \\ &= \left\{ s \;\middle|\; s \in M_S \wedge \left(\Gamma^-_{G(M)}(s) \cup \Gamma^+_{G(M)}(s) \right) \cap \left[\bigcup_{k=1}^{m_b} \overline{B_k} \right] \neq \emptyset \right\} \\ &= \left\{ s \;\middle|\; s \in M_S \wedge \left(\Gamma^-_{G(M)}(s) \cup \Gamma^+_{G(M)}(s) \right) \cap M_B \neq \emptyset \right\} \\ &= \{ s \mid s \in M_S \} \\ &= M_S.\end{aligned}$$

Für $b \in M_B$ existiert ein $k \in \{1, \ldots, m_b\}$ mit $b \in M_{B,k}$. Nun sei $(b,s) \in M_R$. Dann ist $b \in \overline{B_k} \cap \Gamma^-_{G(M)}(s)$, also $s \in M_{S,k}$ und somit $(b,s) \in M_{R,k}$. Gelte jetzt $(s,b) \in M_R$. Dann ist $b \in \overline{B_k} \cap \Gamma^+_{G(M)}(s)$, also $s \in M_{S,k}$ und somit $(s,b) \in M_{R,k}$. Hieraus ergibt sich $M_R = \bigcup_{k=1}^{m_b} M_{R,k}$. □

Durch die Partitionierung ergeben sich Signalschnitte, die bei der parallelen Simulation zu berücksichtigen sind.

Definition 5.12 (**(k,l)-te Signalschnitt-Menge, (k,l)-te Update-Funktion**)
Es gelten die Voraussetzungen von Definition 5.13 und es seien $k,l \in \mathbb{N}$. Dann heißt $S_{k,l} := M_{I,k} \cap M_{O,l}$ die (k,l)-te Signalschnitt-Menge. *Für $s \in S_{k,l}$ sei $U_{k,l}(s) := \nu_l(s)$. $U_{k,l} : M_{S,k} \to W$ ist die (k,l)*-te Update-Funktion.

Somit ist $(S_{k,1}, \ldots, S_{k,m_b})$ das Äquivalent der zum k-ten Block gehörenden Signalschnitt-Liste.

Die strukturellen Modellblöcke induzieren auf natürliche Weise ihnen entsprechende funktionelle Hardware-Modelle, die im folgenden zu definierenden funktionellen Modellblöcke.

Sei daher $M_F = (M_F, \mu, \nu)$ ein fHWM mit zugehörigem bipartiten Graphen $G(M)$ und gelten die Voraussetzungen von Definition 5.11. Dann sind die Gleichungen (5.7) – (5.22) mit $G := G(M)$ erfüllt. Daher seien

$$\mu_k : M_{R,k} \to W \tag{5.23}$$

$$\nu_k : (M_{B,k} \cup M_{S,k}) \to W \cup W^{(\mathbb{N})} \tag{5.24}$$

$$\nu_{B,k} : M_{B,k} \to W^{(\mathbb{N})} \tag{5.25}$$

$$\nu_{S,k} : M_{S,k} \to W \tag{5.26}$$

$$\hat{\nu}_S : M_{B,k} \to W^{(\mathbb{N})} \tag{5.27}$$

eindeutige Abbildungen mit

(5.28)

$$\left(\hat{\nu}_{S,k}(b)\right)_i = \begin{cases} \mu_k(s,b) & \text{falls } s \in \Gamma^-_{G_k(M)}(b) \text{ und } \iota(s) = i \\ 0 & \text{sonst} \end{cases} \tag{5.29}$$

$$\left(\nu_{B,k}(b)\right)_i = \begin{cases} \hat{\nu}_B\left(\hat{\nu}_{S,k}(b), s\right) & \text{falls } s \in \Gamma^+_{G_k(M)}(b) \text{ und } \iota(s) = i \\ 0 & \text{sonst} \end{cases} \tag{5.30}$$

$$\nu_{S,k}(s) = \begin{cases} \hat{\nu}_I(s) & \text{falls } s \in M_{I,k} \cap M_I \\ U_{k,l}(s) & \text{falls } s \in M_{I,k} \setminus M_I \\ \bigvee\limits_{b \in \Gamma^-_{G_k(M)}(s)} \mu_k(b,s) & \text{sonst} \end{cases} \tag{5.31}$$

$$\nu_k|_{M_{B,k}} = \nu_{B,k} \tag{5.32}$$

$$\nu_k|_{M_{S,k}} = \nu_{S,k} \tag{5.33}$$

$$\mu_k(s,b) = \nu_{S,k}(s) \tag{5.34}$$

$$\mu_k(b,s) = (\nu_{B,k}(b))_{\iota(s)} \tag{5.35}$$

für $b \in M_{B,k}$ und $s \in M_{S,k}$ sowie $i \in \mathbb{N}$.

Definition 5.13 (funktionelle Modellblöcke)
Es gelten die Voraussetzungen von Definition 5.11 und $M_1,\ldots,M_{m_b}$ seien die strukturellen Modellblöcke von M bezüglich Φ mit den zugehörigen bipartiten Graphen $G_k(M)$. Weiterhin sei $M_F = (M,\mu,\nu)$ ein funktionelles Hardware-Modell von M. Dann heißen für $k \in \{1,\ldots,m_b\}$ die funktionellen Hardware-Modelle[34] $M_{F,k} = (M_k,\mu_k,\nu_k)$ funktionelle Modellblöcke von M_F *bezüglich Φ, wenn die Gleichungen (5.23) – (5.35) für $b \in M_{B,k}, s \in M_{S,k}$ und $i \in \mathbb{N}$ erfüllt sind.*

Die nächste Folgerung zeigt, daß die Definition der Modellblöcke tatsächlich sinnvoll ist.

Folgerung 5.5
Es gelten die Voraussetzungen von Definition 5.11. Dann gilt $E_{M_F}(M_F) = \bigcup_{k=1}^{m_b} E_{M_{F,k}}(M_{F,k})$.

Beweis: Es gilt mit Folgerung 5.4 dann

$$\begin{aligned}\bigcup_{k=1}^{m_b} E_{M_{F,k}}(M_{B,k}) &= \bigcup_{k=1}^{m_b} \{(b,\nu(b)) \mid b \in M_{B,k}\} = \left\{(b,\nu(b)) \,\middle|\, b \in \bigcup_{k=1}^{m_b} M_{B,k}\right\}\\ &= \{(b,\nu(b)) \mid b \in M_B\}\\ &= E_{M_F}(M_B)\end{aligned}$$

und

$$\begin{aligned}\bigcup_{k=1}^{m_b} E_{M_{F,k}}(M_{S,k}) &= \bigcup_{k=1}^{m_b} \{(s,\nu(s)) \mid s \in M_{S,k}\} = \left\{(s,\nu(s)) \,\middle|\, s \in \bigcup_{k=1}^{m_b} M_{S,k}\right\}\\ &= \{(s,\nu(s)) \mid s \in M_S\}\\ &= E_{M_F}(M_S).\end{aligned}$$

Somit gilt

$$\begin{aligned}E_{M_F}(M_F) &= E_{M_F}(M_B) \cup E_{M_F}(M_S)\\ &= \left(\bigcup_{k=1}^{m_b} E_{M_{F,k}}(M_{B,k})\right) \cup \left(\bigcup_{k=1}^{m_b} E_{M_{F,k}}(M_{S,k})\right)\\ &= \bigcup_{k=1}^{m_b} \Big(E_{M_{F,k}}(M_{B,k}) \cup E_{M_{F,k}}(M_{S,k})\Big)\\ &= \bigcup_{k=1}^{m_b} E_{M_{F,k}}(M_{F,k}).\end{aligned}$$

□

[34] diese sind wegen (5.23) – (5.35) und Definition 5.4 offensichtlich fHWMs

5.7. Der parallele Clock-Cycle-Algorithmus

Der in [8] verbal angegebene parallele Clock-Cycle-Algorithmus wird in diesem Abschnitt auf der Grundlage des funktionellen Hardware-Modells betrachtet.

Definition 5.14 (paralleler Clock-Cycle-Algorithmus)
Es gelten die Voraussetzungen von Definition 5.13 und es sei $n \in \mathbb{N}$. *Das durch Abbildung 32 gegebene Verfahren heißt der* parallele Clock-Cycle-Algorithmus. *Mit* $\mathfrak{C}_{par}(M_{F,1}, \ldots, M_{F,m_b}, n)$ *wird sein Resultat in Abhängigkeit vom Parameter* n *bezeichnet.*

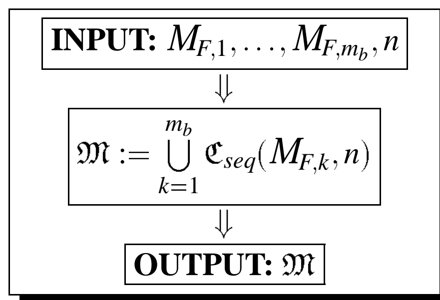


Abbildung 32: Der parallele Clock-Cycle-Algorithmus

Aus dem nächsten Satz ergibt sich die Äquivalenz zum sequentiellen Clock-Cycle-Algorithmus.

Satz 5.4
Es gelten die Voraussetzungen von Definition 5.14. Dann gilt

$$E_{M_F}(M_F) = \mathfrak{C}_{par}(M_{F,1}, \ldots, M_{F,m_b}, 1).$$

Beweis: Folgerung 5.5 liefert zusammen mit Satz 5.2 die Behauptung. □

6. Zusammenfassung

Das Ziel dieser Diplomarbeit war es, eine erste lauffähige parallele Variante des IBM-internen sequentiellen Logiksimulators TEXSIM zu entwerfen und zu implementieren. Ausgangspunkt war eine spezielle Methodik der Partitionierung der zu simulierenden Hardware-Modelle, deren Untersuchung und praktische Umsetzung in einem Forschungsprojekt bearbeitet wird. Das zu entwickelnde parallele Logiksimulationssystem sollte der experimentellen Überprüfung und Bewertung der Untersuchungen zur Modellpartitionierung dienen. IBM Böblingen verknüpft damit konkret die Hoffnung auf ein für den Praxiseinsatz taugliches internes Produkt.

6.1. Ergebnisse

Durch den Einsatz von *parallel*TEXSIM im IBM-Entwicklungslabor Böblingen bei Verifikationsszenarien, die bei der Entwicklung von verschiedenen, durch IBM produzierten, Mikroprozessoren auftraten, hat sich folgendes gezeigt: Mit dem durch den Diplomanden geschaffenen parallelen Logiksimulationssystem liegt ein weitgehend praxisreifes Programm vor, das nur wenig Ansatzmöglichkeiten zu seiner Optimierung liefern dürfte und bis auf einige Ausnahmen voll zu TEXSIM kompatibel ist. In diesem Sinne wurde damit das Ziel der Arbeit mehr als erfüllt.

Aber *parallel*TEXSIM kann nur so gut sein, wie die Daten, die ihm eingespeist werden. Hier heißt das also, nur so gut wie die Partitionierung es ermöglicht. Der Praxiseinsatz hat eine vom Autor seit längerer Zeit vertretene Vermutung eindeutig bestätigt: Die parallele Kommunikation überwiegt gegenüber der parallelen Last. Davon ausgehend konnte im Forschungsprojekt die Partitionsbewertung vom Schwerpunkt Last auf die Kommunikation umgestellt werden. Damit ließen sich in ersten Experimenten gute Beschleunigungswerte bei parallelen Läufen ermitteln (Anhang D).

Für die Berücksichtigung aller designspezifischen Eventualitäten sind aber im Forschungsprojekt noch eine Vielzahl von Untersuchungen und Arbeiten notwendig. Die Weiterentwicklung der Partitionierungswerkzeuge zur Praxistauglichkeit erfolgt momentan in anderen Diplomarbeiten. Wichtige Punkte sind dabei auch deren Parallelisierung und die Ausnutzung modellspezifischer Eigenschaften.

In [2] ist die Methode der Parallelisierung durch Anfügen von Rahmenprogrammen und entsprechende Instanzenbildung angedeutet. Insgesamt läßt sich im Rahmen dieser Diplomarbeit feststellen, daß ein solcher Weg eine praktische und angemessene Vorgehensweise ist. Während der eigentliche Kern der Parallelisierung, also der parallele Clock-Cycle-Algorithmus, in vergleichsweise kurzer Zeit realisiert werden konnte, war für die Vielzahl der API-Funktionen jedoch einige Anpassungsarbeit erforderlich, da es intern in TEXSIM aus Geschwindigkeitsgründen kaum Kapselungen gibt. Für Programme mit weniger umfangreichen Schnittstellen dürfte sich diese Parallelisierungsmethode daher als ideal erweisen.

Im Anschluß an die Arbeiten des Entwurfs von *parallel*TEXSIM wurde ein formales Modell für das funktionelle Verhalten einer Logikschaltung entwurfen, auf dessen Grundlage dann der sequentielle und der parallele Clock-Cycle-Algorithmus beschrieben werden konnte. Die entprechenden Darstellungen sind aufgrund der Kürze der zur Verfügung stehenden Zeit sehr knapp gehalten und geben noch sehr viel Raum für tiefgründigere Untersuchungen vor allem im Hinblick auf die Latche. Es handelt sich hierbei jedoch nur um einen Zusatz zur eigentlichen Diplomarbeit.

6.2. Schlußfolgerungen

Wie schon in [2] und Abschnitt 3.2 festgestellt wurde, ist mit der zugrundegelegten Parallelisierungsstrategie nur für den Teilschritt CLOCK des ALTER-CLOCK-RETRIEVE Schemas prinzipiell ein Performance-Gewinn möglich, während ALTER und RETRIEVE stets mit einem Overhead verbunden sind. Dieser Overhead läßt sich aber verringern, wenn man die Wertänderungen von Facilities durch das Simulationskontrollprogramm in eine Warteschlange einordnet. Der Inhalt der Warteschlange wird erst dann en bloc an die einzelnen Slaves übertragen, wenn es zu einer Wertabfrage bzw. zum CLOCK kommt.

Eine solche Vorgehensweise ist allerdings aufwendig zu implementieren und konnte aufgrund des begrenzten zur Verfügung stehenden Zeitrahmens nicht berücksichtigt werden. Zudem war aufgrund der Problematik des ALTER-CLOCK-RETRIEVE Schemas das Einsatzgebiet des parallelisierten Programmes auf Fälle mit extrem dominierenden CLOCK anvisiert (Regression Tests [6]), so daß dieser Nachteil kaum von Bedeutung sein dürfte (siehe dazu auch Anhang D).

Während (aufgrund der vermutlichen relativen Konstanz des Schrittes TRANSFER) beim parallelen Clock Cycle ein schnelleres Verbindungsnetzwerk kaum spürbare Verbesserungen für CLOCK bringen wird, kann jedoch das volle ALTER-CLOCK-RETRIEVE Schema davon profitieren. Mittlerweile ist ein High Performance Switch mit der vierfachen Geschwindigkeit des für die experimentellen Untersuchungen eingesetzten verfügbar. Je nachdem, wie stark im konkreten Fall ALTER und RETRIEVE ausgeprägt sind, können mit diesem noch deutliche Performance-Steigerungen erreichbar sein.

6.3. Ausblick

Die aufgrund des „Parallel Instance“ Features und der Gate-Level-Optimierung (vgl. Abschnitt 2.1) wünschenswerte Übertragung der Parallelisierungsstrategie auf den von TEXSIM abgeleiteten IBM-internen Simulator MVLSIM [9] sollte für zweiwertige Modelle aufgrund der weitgehenden Übereinstimmung der Schnittstellen mit moderatem Aufwand möglich sein. Bei den zweiwertigen Modellen handelt es sich um solche, bei denen genau wie durch TEXSIM keine

Propagierung mehrwertiger Zustände[35] erfolgt, weshalb von einem Partitionierer keine Signale mit mehrwertigen Zuständen geschnitten werden können.

Für mehrwertige Modelle ist der parallele Clock-Cycle-Algorithmus zu modifizieren. Dabei ist für jedes geschnittene Signal ein Byte zu lesen, zu transferieren und zu schreiben, was beim Message Passing das Datenaufkommen enorm vergrößert.[36] In diesem Fall ist es deshalb vorstellbar, daß die gewählte Parallelisierungsstrategie nicht sinnvoll anwendbar sein könnte. Die Mikroprozessorentwicklung ist aber noch eindeutig durch die Simulation zweiwertiger Modelle geprägt.

Daß die Parallelisierung von MVLSIM mit nur geringen Änderungen an den nicht direkt auf TEXSIM basierenden Sourcen von mTEXSIM und sTEXSIM möglich sein sollte, wird dadurch erhärtet, daß es problemlos möglich war, zum Ende der Diplomarbeit die Quellen der aktuellsten TEXSIM-Version mit dem Code von *parallel*TEXSIM zu verschmelzen. An den Quellen von MVLSIM sind jedoch zahlreiche Änderungen analog den bei TEXSIM durchgeführten (und damit auf festgelegte Art und Weise) zu vollziehen. Außerdem sind für gute Resultate die Auswirkungen des „Parallel Instance“ Features (siehe S. 7) bei der Partitionierung zu berücksichtigen.

Die Anzahl paralleler Tasks (≤ 32), mit denen *parallel*TEXSIM bislang verwendet wird,[37] liegt noch im Bereich der für SMP möglichen Werte, wenngleich aufgrund heuristischer Überlegungen optimale Werte bei ca. 16 Modellblöcken und damit einer noch selten durch SMP unterstützten Prozessorzahl zu erwarten sind. Durch verbesserte Busstrukturen (wie etwa der UPA-Bus von Sun [27]) sollten aber auf vielen Hardware-Architekturen bis zu 32 Prozessoren sinnvoll unterstützt werden können.

Wenn man den Aussagen der Firma SpeedSim Glauben schenkt, so lassen sich mittels SMP und einer (im Unterschied zur hier bislang zugrundegelegten rein graphentheoretischen) modellspezifischen Partitionierung exzellente Beschleunigungen erreichen. Ihr zeitgesteuerter Simulator SpeedSim/3 soll auf SMP-Maschinen der Firma Sun für bis zu 8 Prozessoren einen nahezu linearen Speed-Up erreichen. Dafür wird (anders als bei *parallel*TEXSIM) ein speziell auf SMP zugeschnittener Simulationsalgorithmus verwendet und das Modell „per Hand“ in sogenannte Modell Segments zerlegt.

Ein interessanter Gesichtspunkt der parallelen Compiled Code-Simulation ist der unter bestimmten Umständen theoretisch mögliche *superlineare* Speed-Up. Durch die Aufteilung des Modells und damit des Simulationscodes in kleinere Teile kann es sein, daß die Simulationscodes vollständig in die Secondary Caches der einzelnen Prozessoren passen. In diesem Fall ist die Simulation der einzelnen Modellblöcke in der Regel deutlich schneller als die der äquivalenten Teile im Ursprungsmodell.

[35] MVLSIM läßt sich in einem solchen Modus betreiben und arbeitet dann effizienter.

[36] Es sind 7 verschiedene Logikzustände möglich, so daß wenigstens mit einer Verdreifachung zu rechnen ist, wenn man für den Transfer nur jeweils drei Bits zur Kodierung verwendet.

[37] Die interne Implementierung ermöglicht bis zu 127 Slaves.

Zusammenfassung

In der vorliegenden Diplomarbeit wird der Entwurf, die Implementierung und die praktische Handhabung eines durch den Diplomanden entwickelten parallelen Logiksimulationssystems beschrieben. Als Ausgangspunkt der Arbeit diente ein firmeninterner sequentieller, mit dem Clock-Cycle-Algorithmus arbeitender Logiksimulator, der vor allem von IBM erfolgreich bei der Entwicklung führender Mikroprozessorarchitekturen (z.B. POWER2, ESA/390 und PowerPC) eingesetzt wurde. Der Parallelisierung liegt eine Methode der Modellpartitionierung zugrunde, deren praktische Umsetzung im Rahmen eines von der DFG und IBM Böblingen geförderten Forschungsprojektes erfolgte. Die Diplomarbeit wurde in enger Kooperation mit den Mitarbeitern dieses Forschungsprojektes im Anschluß an ein IBM-Berufspraktikum des Diplomanden bearbeitet.

Darüber hinaus wird auf der Basis eines funktionellen Hardware-Modells eine formale Beschreibung des sequentiellen und des parallelen Clock-Cycle-Algorithmus gegeben.

Literatur

[1] Gerd Meister. A Survey on Parallel Logic Simulation. Technical Report 14, Universität des Saarlandes, Institut für Informatik, 1993.

[2] Denis Döhler. Parallelisierung des Logiksimulators TEXSIM — Praktische Vorarbeiten auf Basis des AIX Parallel Environment. Studienarbeit, Universität Leipzig, Institut für Informatik, Dezember 1995.

[3] F. Mattern and H. Mehl. Diskrete Simulation — Prinzipien und Probleme der Effizienzsteigerung durch Parallelisierung. *Informatik-Spektrum*, (12):198–210, 1989.

[4] Stefan Winterstein-Theobald. Implementierung des Time-Warp-Algorithmus für den verteilten VHDL-Simulator DVSIM. Diplomarbeit, Universität Saarbrücken, Fachbereich Informatik, September 1995.

[5] P. Marwedel. *Synthese und Simulation von VLSI-Systemen*. Carl Hanser Verlag, 1993.

[6] Wilhelm G. Spruth. *The Design of a Microprocessor*. Springer Verlag, 1989.

[7] David S. Zike. Cycle-based Simulation in IBM. Lecture in Workshop “Parallel Logic Simulation” (Universität Leipzig, Institut für Informatik), February 1996.

[8] Klaus Hering, Rainer Haupt, and Thomas Villmann. Cone-basierte, hierarchische Modellpartitionierung zur parallelen compilergesteuerten Logiksimulation beim VLSI-Design. Technical Report 13, Universität Leipzig, Institut für Informatik, 1995.

[9] Jeannette Sutherland and Vlad Zavadsky. *MVLSIM Multivalue Simulation System — User’s Guide*. IBM Microelectronics Division Austin, TX, March 1996.

[10] Chung-Chih Tung and Cary Ussery. FACE-OFF: Cycle-based vs. event-driven Simulation. *Computer Design’s ASIC DESIGN*, pages A14–A17, August 1994.

[11] Lisa Maliniak. Partitioning is key too using cycle-based simulation in event-driven verification. *Electronic Design*, pages 31–32, May 1996.

[12] Klaus Hering. Partitionierungsalgorithmen für Modelldatenstrukturen zur parallelen compilergesteuerten Logiksimulation (Projekt). Technical Report 5, Universität Leipzig, Institut für Informatik, 1995.

[13] Christian Sporrer. *Verfahren zur Schaltungspartitionierung für die parallele Logiksimulation*. Verlag Shaaker Aachen, 1995.

[14] Klaus Hering, Rainer Haupt, and Thomas Villmann. An Improved Mixture of Experts Approach for Model Partitioning in VLSI-Design Using Genetic

Algorithms. Technical Report 14, Universität Leipzig, Institut für Informatik, 1995.

[15] Klaus Hering, Rainer Haupt, and Thomas Villmann. Hierarchical Strategy of Model Partitioning for VLSI-Design Using an Improved Mixture of Experts Approach. In *Proc. of PADS'96*, pages 106–113, 1996.

[16] Robert Reilein. Parallelisierung des Logiksimulators TEXSIM — Realisierung der Protozerlegung auf DA_DB-Basis. Studienarbeit, Universität Leipzig, Institut für Informatik, September 1996.

[17] Jeanine Weißenfels. Entwurf und Implementierung eines verteilten VHDL-Simulators. Diplomarbeit, Universität Saarbrücken, Fachbereich Informatik, Mai 1994.

[18] Franz-Josef Simmel. Entwurf und Implementierung eines verteilten VHDL-Simulators. Diplomarbeit, Universität Saarbrücken, Fachbereich Informatik, April 1994.

[19] Charles H. Malley and Max Dieudonné. Logic Verification Methodology for PowerPC Microprocessors. In *Proc. of 32nd ACM/IEEE Design Automation Conference*, 1995.

[20] A. Aharon, A. Bar-David, B. Dorfman, E. Gofman, M. Leibowitz, and V. Schwartzburd. Verification of the IBM RISC System/6000 by a dynamic biased pseudo-random test program generator. *IBM Systems Journal*, 30(4), 1991.

[21] David S. Zike. *The TEXSIM and CORVETTE Simulation Interface*. AWD Austin, Oktober 1992.

[22] Harald Gerst. *Xmon — A Simulation Monitor based on TEXSIM and X/Motif for small pieces of Hardware up to MP systems with n processors — User's Guide*. Logic Design Support Böblingen, October 1995.

[23] Frank Buschmann. The Master-Slave Pattern. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Programm Design*, pages 133–142. Addison Wesley, 1995.

[24] M. Snir, P. Hochschild, and D. D. Frye. The Communication Software and Parallel Environment of the IBM SP2 (PE). *IBM Systems Journal*, 34(2):185–204, 1995.

[25] Stunkel et al. The SP2 Communication Subsystem. Technical report, IBM T.J. Watson Research Center, August 1994.

[26] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, June 1995.

[27] M. Clauß. Architektur des UPA-Busses von Sun. *iX*, (10):110–116, 1996.

Anlagenverzeichnis

Anlage A Das TEXSIM-API

Anlage B Die Text-Listenfiles

Anlage C Das TEXSIM-Subcommand-Interface

Anlage D Performance-Messungen

Anlagen

A Das TEXSIM-API

Die durch *parallel*TEXSIM noch nicht unterstützten Funktionen sind über einen entsprechenden Kommentar gesondert gekennzeichnet und liefern bei ihrer Verwendung Fehlermeldungen.

```
typedef int FACIDX;
typedef int ROWIDX;
typedef int BUSIDX;
typedef int SIMID;
typedef int (FUNCTION) (char*);
struct facref
{
  FACIDX       s;                  /* facility index        */
  char         *facname;           /* ptr to facility name  */
  int          offset;             /* offset of reference   */
  int          length;             /* length of reference   */
  unsigned int row;                /* row number for arrays */
  unsigned int row_high;           /* row number for arrays */
};
typedef struct facref FACREF;


void      deletec    (char* name);
FUNCTION* locatec    (char* name);
FUNCTION* loadc      (char* name);
FUNCTION* ldbindc    (char* name);
FUNCTION* ldbindc2   (char* name);
FUNCTION* loadc2     (char* name);
FUNCTION* xloadc     (char* name, char* path);
FUNCTION* xloadc2    (char* name, char* path);
int       bindc      (char* name1, char* name2);
int       simbind    (char* name1, char* name2);
FILE*     fileopen   (char* env, char* name, char* mode);
int       filesize   (FILE*);
unsigned  int filedate(FILE*);
char*     filename   (char* env, char* name);

int       SUBCMD     (char* cmd, int len, ...);

void      clrfac     (FACIDX);
FACIDX    efsrtsym   (char *name);
FACIDX    symbol     (char *name);
int       getint     (FACREF*);
int       getintx    (FACREF*);
void      putint     (FACREF*, int);
void      dotint     (FACREF*, int);
void      stickint   (FACREF*, int);
int       getintx    (FACREF*);
void      alterc     (FACREF*, char* value, char mode);
```

```
void      stickc     (FACREF*, char* value, char mode);
void      dotc       (FACREF*, char* value, char mode);
void      displayc   (FACREF*, char mode);
void      unstick    (FACREF*);
int       setvarc    (FACREF*, char* value, char mode);
void      clockc     (int);
int       symrows    (FACIDX);
int       symlen     (FACIDX);
char*     symname    (FACIDX);
char*     symname2   (FACIDX);
FACIDX    firsts     (void);
FACIDX    nexts      (FACIDX);
FACIDX    firstaets  (void);   /* not supported */
FACIDX    nextaets   (FACIDX); /* not supported */
void      texmsg     (int rc, char* msg);
void      va_texmsg  (int rc, char* msg, ...);
void      qstickc    (FACREF*, char* value, char mode);
void      qalterc    (FACREF*, char* value, char mode);
void      qputint    (FACREF*, int);
void      qstkint    (FACREF*, int);
BUSIDX    firstx     (void);   /* not supported */
BUSIDX    nextx      (BUSIDX); /* not supported */
FACIDX    xfac       (BUSIDX);
int       assertrc   (void);   /* not supported */
int       assertct   (void);   /* not supported */
int       busmsgct   (void);   /* not supported */
int       curcycle   (void);
int       cursig     (void);
int       curl1sig   (void);
int       curl2sig   (void);
double    switchcnt  (void);   /* not supported */
double    onecnt     (void);   /* not supported */
void      errorrtn   (void (*rtn)());
void      msgrtn     (void (*rtn)(int rc, char *msg));
void      txsave     (void (*rtn)());
void      txrestore  (void (*rtn)());
void      globalvg   (char* group, char* name, char *value);
void      globlvsp   (char* group, char* name, char *value);
void      globalvs   (char* group, char* name, char *value);
ROWIDX    firstrow   (FACIDX);
ROWIDX    nextrow    (ROWIDX);
unsigned int rowhigh (ROWIDX);
unsigned int rowlow  (ROWIDX);
int       symsparse  (FACIDX);
void      resetaa    (FACIDX);
int      *swcount    (FACREF*); /* not supported */
int      *onecount   (FACREF*); /* not supported */
int       hascount   (FACREF*); /* not supported */
int       simconfig  (char *);
void      setqtype   (int);
```

```
char      *modelfn    (void);
void       clockset   (int);
int        isbus      (FACIDX);
int        symtype    (FACIDX);
void       cyclec     (void);
SIMID      txid       (void);                 /* not supported */
void       txsetid    (SIMID);                /* not supported */
int        txsim      (int argc, char** argv); /* not supported */
void       txst       (void);                 /* not supported */
jmp_buf    severror;                          /* not supported */
int        get_count_func (int);

struct tokparms
{
  int itemlen;       /* length of the token pointed to by item */
  int it;                              /* item (token) type */
  char *item;                       /* pointer to the token */
  char token[MAX_TOKEN];         /* actual upper case token */
};
typedef struct tokparms TOKPARMS;
void      tokparse(char* start, char* end);
void      token(TOKPARMS*);
void      token2(TOKPARMS*);
void      tokfndc(TOKPARMS*, char);
void      tokfnds(TOKPARMS*, char*);
int       tokline(void);
int       tokos(void);
void      tokpop(void);
void      tokpush(void);
void      tokreset(void);
void      tokclear(void);
char*     tokstart(void);
void      tokalpha(char);
void      toknumer(char);
void      tokspec(char);
```

Die Funktionen `txid()`, `txsetid()`, `txsim()`, `txst()` sowie die Variable `severror` sind nicht Bestandteil des eigentlichen API, sondern nur im Objekt-Code `texsim.o` enthalten, der für die Nutzung des Simulators unter der Kontrolle eines anderen Programmes statisch angebunden werden kann. Diese Möglichkeit besteht aber bei *parallel*TEXSIM noch nicht. Die Funktionen `switchcnt()`, `onecnt()`, `swcount()`, `onecount()` und `hascount()` dienen der Protokollierung von Wertveränderungen der Facilities. Weil diese bei der parallelen Simulation zerteilt werden können, ist es wenig sinnvoll, solche Funktionen zu unterstützen. Gleiches gilt für `assertrc()`, `assertct()` und `busmsgct()`, die Assertion Violations bzw. Bus Violations zählen. Für die typischen Einsatzszenarien von *parallel*TEXSIM werden sie auch nicht gebraucht. Die restlichen nicht unterstützten Funktionen werden bei AETs bzw. bei der Busabfrage verwendet.

B Die Text-Listenfiles

Die Cross-Referenz-Liste Der strukturelle, zeilenorientierte Aufbau der Liste ist in Abbildung B.1 vom Prinzip her dargestellt. Die Bedeutung spezieller, darin verwendeter Symbole wird in Tabelle B.1 erklärt. Insbesondere ist NoB (number of blocks) ein Platzhalter (Variable) für die Anzahl von Blöcken, in denen die jeweilige Facility enthalten ist. Aus der sich durch die Benennung der Modellblöcke ergebenden Reihenfolge der Blöcke (vgl. Abbildung 16) bestimmen sich die Blocknummern (block identifiers). Die wesentliche zusätzliche Festlegung besteht darin, daß Bestandteile eines zerteilten Vektors (dieses sind Einzelnetze) unmittelbar aufeinander folgen müssen. Die Ursache dafür ist die so ermöglichte beschleunigte Auswertung der Liste.

```
@T ⟨hour⟩ ⟨minute⟩ ⟨second⟩
@D ⟨month⟩ ⟨day⟩ ⟨year⟩
@Z ⟨number of blocks⟩
@X ⟨number of nets and vectors⟩
@N ⟨net name⟩[(⟨index⟩)] ⟨NoB⟩ {⟨block id⟩}
⋮
@N ⟨net name⟩[(⟨index⟩)] ⟨NoB⟩ {⟨block id⟩}
@V ⟨vector name⟩(⟨index⟩..⟨index⟩) ⟨NoB⟩ {⟨block id⟩}
⋮
@V ⟨vector name⟩(⟨index⟩..⟨index⟩) ⟨NoB⟩ {⟨block id⟩}
@Y ⟨number of arrays⟩
@A ⟨array name⟩ ⟨NoB⟩ {⟨block id⟩}
⋮
@A ⟨array name⟩ ⟨NoB⟩ {⟨block id⟩}
```

Abbildung B.1: Struktur der Cross-Referenz-Liste

Die Signalschnitt-Listen Der strukturelle Aufbau in Textzeilen gestaltet sich prinzipiell gemäß Abbildung B.2 mit der Symbolik aus Tabelle B.1, wobei NoB (number of connected blocks) hier als Platzhalter (Variable) für die Anzahl von Blöcken dient, mit denen die jeweilige Facility verbunden ist. Die Blocknummern (block identifiers) entsprechen der durch die Benennung der Modellblöcke festgelegten Reihenfolge der Blöcke (Abbildung 16).

Symbol	Bedeutung
{T}	T tritt NoB–mal auf
[T]	T kann einmal auftreten, muß es aber nicht
⟨T⟩	T ist durch seine semantische Bedeutung zu ersetzen
⋮	beliebige Anzahl des davor stehenden Zeilentyps

Tabelle B.1: Spezielle syntaktische Symbole

```
@T ⟨hour⟩ ⟨minute⟩ ⟨second⟩
@D ⟨month⟩ ⟨day⟩ ⟨year⟩
@X ⟨number of inputs⟩
@I ⟨input name⟩[(⟨net index⟩)] ⟨NoB⟩ {⟨block id⟩}
⋮
@I ⟨input name⟩[(⟨net index⟩)] ⟨NoB⟩ {⟨block id⟩}
@Y ⟨number of outputs⟩
@O ⟨output name⟩[(⟨net index⟩)] ⟨NoB⟩ {⟨block id⟩}
⋮
@O ⟨output name⟩[(⟨net index⟩)] ⟨NoB⟩ {⟨block id⟩}
```

Abbildung B.2: Struktur einer Signalschnitt-Liste

Keyword	Bedeutung
@T	Erzeugungszeit
@D	Erzeugungsdatum
@X	Anzahl der Vektoren und Netze bzw. der Inputs
@Y	Anzahl der Arrays bzw. der Outputs
@Z	Anzahl der Modellblöcke
@A	Array
@N	Netz
@V	Vektor
@I	Input
@O	Output

Tabelle B.2: Keywords der Listen-Dateien

C Das TEXSIM-Subcommand-Interface

Die Befehle des Subcommand-Interfaces können interaktiv über den Kommandoprozessor oder über die API-Funktion `SUBCMD()` verwendet werden.

AET ON | OFF | FLUSH | RESET

ALIAS *alias* = *facref* [‖ *facref*] ...

ALIAS *alias facref*

ALTER *facref value* [*mode*]

ASSERTS ON | OFF

BATCH *fn* [*parms*]

BIND *pgm1 pgm2*

CALL *pgm*

CHECKPOINT [*fn*]

CLEAR *facref*

CLIENT *pgm parms*

CLOCK [*n* [UNTIL (*facref* = *value*)] [SET (*var*)]]

CLOCKSET *n*

COMMAND *pgm*

CONTINUE

CYCLE

CYCLEXIT START *pgm* [*parms*]

CYCLEXIT STOP *pgm*

CYCLEXIT (*start,interval,stop*) *pgm* [*parms*]

DELETE *pgm*

DISPLAY *facref* [*mode*]

DOT *facref value* [*mode*]

ECHO ON | OFF

ENDEXIT START *pgm* [*parms*]

ENDERROR ENABLE | DISABLE

EXIT [*rc*]

EXPECT *facref value* [*mode*]

HELP

?

INITARY *fn*

INITEXIT START *pgm* [*parms*]

INITEXIT STOP *pgm*

IEEE ON | OFF

LOAD pgm

LOG ON | OFF

MAC *fn*

MODELNAME *name*

QALTER *facref value* [*mode*]

QSTICK *facref value* [*mode*]

QTYPE BEFORE | AFTER

QUERY *var request*

QUIETEXP *facref value* [*mode*]

QUIT [*rc*]

RESETEXIT BEFORE AFTER | START *pgm* [*parms*]

RESETEXIT BEFORE AFTER | STOP *pgm*

RESTART [*fn*]

RETURN

SETVAR *facref var* [*mode*]

SHAPE *facname* [*rows width type*]

SIMLOG *msg*

SIMRESET

STICK *facref value* [*mode*]

STOP [IN] *cycle*

STOP *+n*

STOPEXIT START *pgm parms*

STOPEXIT STOP *pgm*

SUBCOMMAND

SWITCHING ON | OFF | RESET

SYSTEM *cmd*

TOUCH

TRACE ON | OFF

UNSTICK *facref*

XBATCH *fn* [*parms*]

XLOAD *fn* [*parms*]

Es bedeuten *rc* Return Code, *fn* File Name, *facref* Facility Reference, *pgm* Program, *parms* Parameters und *var* Variable. Durch *parallel*TEXSIM werden bis auf SWITCHING (siehe dazu Anhang A) alle Kommandos voll unterstützt.

D Performance-Messungen

D.1 Allgemeine Betrachtungen

Zur Einschätzung der Bedeutung der Performance-Messungen ist es zunächst erforderlich, einige allgemeinere Betrachtungen durchzuführen.

Hierzu sei T_{seq} die Gesamtlaufzeit von TEXSIM und T_{par} die von *parallel*TEXSIM. Vernachlässigt man gewisse Unterschiede beim Start-Up der beiden Programme, so setzen sich diese wie folgt zusammen:

$$T_{seq} = T_{user} + T_{model}^{seq},$$

wobei T_{user} die Zeit ist, die ein User-Programm benötigt, und T_{model}^{seq} die Dauer der eigentlichen Modellsimulation ist;

$$T_{par} = T_{user} + T_{model}^{par},$$

wobei T_{model}^{par} die für den parallelen Clock-Cycle-Algorithmus erforderliche Zeit ist, und der Einfachheit halber angenommen wird, daß die API-Zugriffe (ALTER und RETRIEVE) im parallelen Fall genauso schnell sind. Damit ergibt sich für den Speed-Up folgende Beziehung:

$$\frac{T_{seq}}{T_{par}} = \frac{T_{user} + T_{model}^{seq}}{T_{user} + T_{model}^{par}}.$$

Eine obere Grenze ergibt sich offensichtlich für $T_{model}^{par} \rightarrow 0$. Also gilt

$$\begin{aligned}\frac{T_{seq}}{T_{par}} \leq \lim_{T_{model}^{par} \rightarrow 0} \frac{T_{seq}}{T_{par}} &= \lim_{T_{model}^{par} \rightarrow 0} \left(\frac{T_{user} + T_{model}^{seq}}{T_{user} + T_{model}^{par}} \right) \\ &= \frac{T_{user} + T_{model}^{seq}}{T_{user} + 0} = 1 + \frac{T_{model}^{seq}}{T_{user}}.\end{aligned}$$

In der Praxis gilt natürlich eine echte Kleiner-Relation. Die erhaltene obere Grenze soll an einem Beispiel verdeutlicht werden.

Das beste für den Xmon bislang aufgetretene Verhältnis ist

$$\begin{aligned}T_{user} &= \frac{1}{10} T_{seq}, \\ T_{model}^{seq} &= \frac{9}{10} T_{seq}.\end{aligned}$$

Dies bedeutet für den maximal erreichbaren Speed-Up

$$\begin{aligned}\frac{T_{seq}}{T_{par}} &< 1 + \frac{\frac{9}{10}T_{seq}}{\frac{1}{10}T_{seq}} \\ &< 10.\end{aligned}$$

Um ein von den eigentlichen User-Programmen unabhängiges Maß für die qualitative Gütebewertung der Modellpartitionierung und von *parallel*TEXSIM zu haben, basieren die Performance-Messungen auf der Anwendung des sequentiellen und des parallelen Clock-Cycle-Algorithmus auf unitialisierten Prozessormodellen, d.h., es ist $T_{user} = 0$. Für den Speed-Up ergibt sich daher keine Beschränkung nach oben:

$$\begin{aligned}\lim_{T_{user} \to 0} \frac{T_{seq}}{T_{par}} &\leq \lim_{T_{user} \to 0} \left(1 + \frac{T^{seq}_{model}}{T_{user}}\right) \\ &= \infty.\end{aligned}$$

Damit ist jedoch das Problem verbunden, daß TEXSIM über eine sehr intelligente Implementierung des Simulationsalgorithmus verfügt. Dieser evaluiert pro Zyklus nicht das gesamte Modell, sondern er unterscheidet L1- und L2-Latch-Gruppen. Darüber hinaus können über die Sending-Unit-Boxen (vgl. S. 61) Multi-Clock-Domains behandelt werden. Bei der Simulation eines unitialisierten Modells werden daher nur die in jedem Zyklus zu evaluierenden Modellbestandteile abgearbeitet. Hinzu kommt, daß diese Feinheiten bei der Partitionierung noch nicht berücksichtigt werden.

Es stellt sich nun die Frage, inwieweit Performance-Messungen auf die genannte Weise (mit ermittelten Werten T_{seq} und T_{par}) qualitativ aussagekräftig sind. Sei daher x die im sequentiellen und y die im parallelen Fall benötigte Zeit für die Evaluierung der nicht in jedem Zyklus zu behandelnden Modellbestandteile. Gilt dann $\frac{T_{seq}}{T_{par}} \leq \frac{x}{y}$, so ergibt sich

$$\begin{aligned}\frac{T_{seq}}{T_{par}} &\leq \frac{x}{y} \\ y \cdot T_{seq} &\leq x \cdot T_{par} \\ T_{seq} \cdot T_{par} + y \cdot T_{seq} &\leq T_{seq} \cdot T_{par} + x \cdot T_{par} \\ (T_{par} + y) \cdot T_{seq} &\leq (T_{seq} + x) \cdot T_{par} \\ \frac{T_{seq}}{T_{par}} &\leq \frac{T_{seq} + x}{T_{par} + y}.\end{aligned}$$

Bei $\frac{T_{seq}}{T_{par}} > \frac{x}{y}$ dagegen erhält auf die gleiche wie oben

$$\frac{T_{seq}}{T_{par}} > \frac{T_{seq}+x}{T_{par}+y}.$$

Damit ergibt sich, da $y \leq x$ klar ist, im schlechtesten Fall $x = y$ bei einem tatsächlichen Speed-Up für das unitialisierte Modell $\frac{T_{seq}}{T_{par}} > 1$, daß der gemessene Wert eine Obergrenze des Speed-Ups für das gleiche initialisierte Modell ist. Hierin zeigt sich, wie wichtig die Berücksichtigung dieser Feinheiten bei der Partitionierung ist.

D.2 Ergebnisse

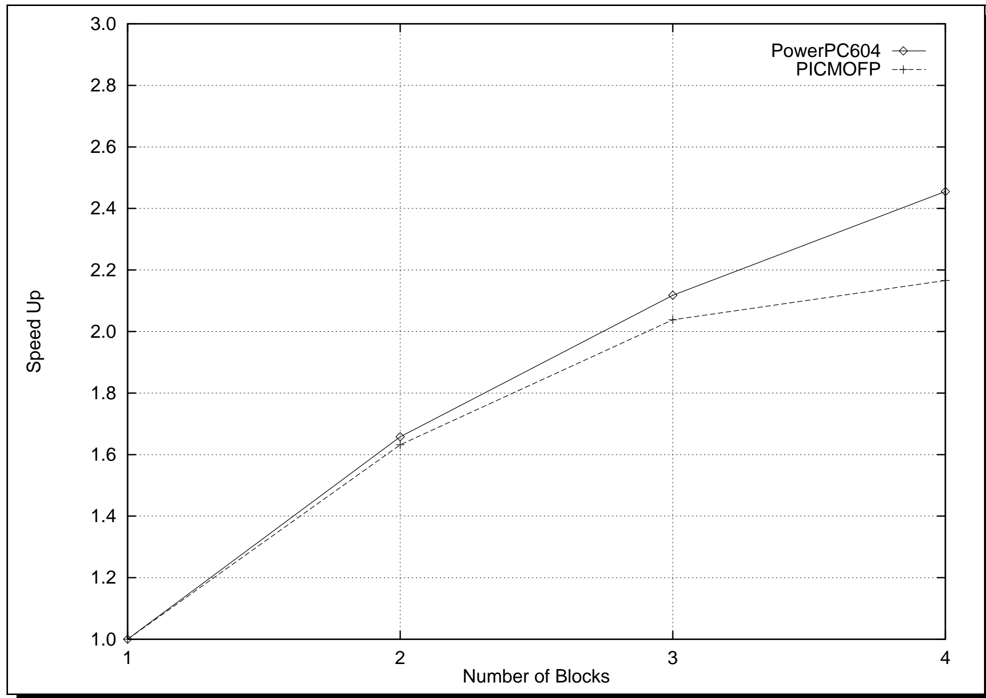
Die Performance-Untersuchungen wurden im IBM Entwicklungslabor Böblingen auf einem unbelasteten Parallelrechner IBM RS/6000 SP (SP2) mit 6 unter AIX 3.2.5 mit Parallel Environment (in der Version 2 Release 1) laufenden Knoten mit 66.7 MHz getakteten POWER2-Prozessoren, die jeweils über 2 GByte RAM und 2 GByte Swap verfügten, durchgeführt.

Die ersten Untersuchungen mit unterschiedlichen Partitionierungsalgorithmen und -strategien lassen vermuten, daß die Schritte CLOCK, GET und PUT der Implementierung des parallelen Clock-Cycle-Algorithmus in etwa gleichwertig bzgl. der parallelen Simulationskosten sind, während man die Kosten für TRANSFER als deutlich geringer und für alle Partitionierungen mit gleicher Blockanzahl eines Simulationsmodells als weitgehend konstant ansehen kann.

Die folgende Tabelle enthält für Blockzahlen von 2 bis 4 (Blockzahl 1 steht für den sequentiellen Fall) die bislang besten Laufzeiten (gerundet in Sekunden), die für 10000 Zyklen bei zwei vollkommen verschiedenen Modellen gemessen wurden — inklusive der entsprechenden Zeiten für die einzelnen Modellblöcke. Es handelt sich dabei um Partitionierungen zweier tatsächlich produzierter Mikroprozessoren: PowerPC 604 vom IBM, Motorola und Apple sowie der erste S/390-Prozessor in CMOS-Technologie PICMOFP (Codename PICASSO). Die fehlenden Werte für einzelne Teilblöcke konnten aufgrund der Kürze der zur Verfügung stehenden Zeit nicht ermittelt werden. Eine separate Abbildung gibt die sich ergebenden Speed-Up-Werte an.

Zusätzlich wird für einen gegenüber PICASSO ca. zehnmal größeren, bereits produzierten CMOS S/390-Prozessor ML100M0S (Codename MONET) die parallele Laufzeit für die bislang einzige Partitionierung (in 4 Blöcke) angegeben. Erste Läufe mit realen Test Cases für MONET mittels Xmon zeigten zudem Speed-Up-Werte über 3. Damit scheint sich die These zu bestätigen, daß der erreichbare Speed-Up mit der Modellgröße wächst.

Modell	Blöcke	Block 1	Block 2	Block 3	Block 4	Parallel
PowerPC 604	1	371	—	—	—	371
	2	174	212	—	—	232
	3	?	?	?	—	170
	4	126	110	118	91	148
PICMOFP	1	188	—	—	—	188
	2	88	106	—	—	115
	3	72	64	71	—	92
	4	56	56	43	?	87
ML100M0S	1	1722	—	—	—	1722
	4	410	381	400	418	509



Erklärung

Ich versichere, daß ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Leipzig, den

(DENIS DÖHLER)