Abschlußbericht zum DFG-Projekt

Leistungssteigerung ereignisgesteuerter Simulation durch Multi–Mikro–Rechnersysteme

im Rahmen des Schwerpunktprogramms
"Multi-Mikro-Rechner" (Juli 1989 - Juli 1992)

F. Mattern, J. Richter

Fachbereich Informatik
Universität des Saarlandes
Postfach 1150
66041 Saarbrücken

H. Mehl

Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
67653 Kaiserslautern

Inhaltsverzeichnis

1 Vorbemerkungen

1.1 Projektziele

Die Hauptaufgabe des Projektes "Leistungssteigerung ereignisgesteuerter Simulation durch Multi–Mikro–Rechnersysteme" welches von August 1989 bis Juli 1992 von der Deutschen Forschungsgemeinschaft im Rahmen des Schwerpunktprogramms "Multi-Mikro-Rechnersysteme" gefördert wurde, bestand in der Durchführung von grundlegenden Untersuchungen zur parallelen Ausführung von ereignisgesteuerten Simulationsmodellen auf Mehrrechnersystemen mit dem Ziel der Leistungssteigerung. Neben einer theoretischen Fundierung und Weiterentwicklung von Basisprizipien der sogenannten parallelen und verteilten Simulation ging es vor allem darum, die verschiedenen Methoden auf Mehrrechnersystemen prototypisch zu realisieren und praktisch zu evaluieren.

1.2 Hintergründe und Motivation

Die Beschleunigung des bekannten und oft verwendeten ereignisgesteuerten Simulationsprinzips durch Parallelisierung stellt eine besondere Herausforderung dar: Einerseits sind darauf beruhende Simulationsanwendungen typische "Langläufer", andererseits kann die ereignisgesteuerte Simulation in einem gewissen Sinn als inhärent sequentiell bezeichnet werden, da letztendlich prinzipiell jedes eingeplante Ereignis, wenn es zur Ausführung kommt, den globalen Zustand so ändern kann, daß andere, weiter in der Zukunft eingeplante Ereignisse, davon betroffen werden. Würde man nun zwei derartig voneinander abhängige Ereignisse gleichzeit ausführen, mit dem Ziel, den Gesamtablauf zu beschleunigen, so könnte das Ergebnis verfälscht werden. Ohne weitere (semantische) Informationen über die Abhängigkeit der Ereignisse voneinander kann ein Simulator also die Ereignisse stets nur sequentiell, entsprechend ihrem eingeplanten Zeitpunkt, ausführen.

Bei der zuletzt genannten Schwierigkeit setzt die Theorie der verteilten Simulation an. Die grundsätzlich Idee beruht darauf, die Ereignisse in Klassen einzuteilen und diese Klassen weitestgehend voneinander zu entkoppeln, so daß zumindest Ereignisse verschiedener Klassen gleichzeitig ausgeführt werden können. Die Klassenbildung erfolgt in der Weise modellbezogen, daß Einheiten mit disjunktem Zustandsraum gebildet werden, bei denen jeweils Ereignisse stattfinden. Jeder Einheit wird ein eigener Prozessor zugeordnet, der die "lokalen" Ereignisse (streng sequentiell) ausführt. Zwar existiert kein globaler Speicher, über den die lokalen Ereignisse wechselwirken können, jedoch muß prinzipiell die Möglichkeit bestehen, daß auch Ereignisse verschiedener Einheiten aufeinander einwirken können, da sonst nicht ein einziges System, sondern mehrere völlig unabhängige Systeme simuliert würden. Dies geschieht, indem ein Ereignis auch bei anderen Einheiten Ereignisse – in einer gewissen kontrollierten Weise – einplanen kann. Realisiert wird dies im zugrundliegenden Rechensystem typischerweise durch Ereignisnachrichten, die zwischen den Prozessoren ausgetauscht werden.

Seit Ende der 70er Jahre wird das Problem der Parallelisierung ereignisgesteuerter Simulationen in der Literatur untersucht. Allerdings lieferten zunächst durchgeführte rein algorithmische und analytische Untersuchungen keine klare Aussage dazu, unter welchen Bedingungen in der Praxis tatsächlich mit einer wesentlichen Beschleunigung zu rechnen ist. Mit der Verfügbarkeit einfach nutzbarer Mehrrechnersysteme Ende der 80er Jahre wurde vor allem an amerikanischen Universitäten mit praktisch orientierten Forschungen begonnen. Die ersten Ergebnisse waren teilweise ernüchternd, zeigten andererseits jedoch interessante Aspekte auf, die vorher unbeachtet geblieben sind und eine genauere konzeptionelle und empirische Untersuchung rechtfertigen.

Diese Situation stellte – zusammen mit dem Erfahrungshintergrund der Antragsteller auf dem Gebiet der Betriebsorganisation verteilter Rechnersysteme und der verteilten Algorithmen – die Motivation für das Projekt dar, dessen Hauptergebnisse in diesem Bericht vorgestellt werden. Da diese Ergebnisse ihren Niederschlag in mehreren Veröffentlichungen in Fachzeitschriften, Tagungsbänden und internen Berichten fanden [MAM89a, MAM89b, MAT93a, MEH91a, MEH91c, MEH92a, MEH92c, MEH93b, MEH93c, MMR90a, MMS91a, RIC90a, RIC91a], beschränken wir uns hier allerdings auf eine gekürzte Darstellung der wesentlichen Resultate und verweisen ansonsten auf die an den entsprechenden Stellen zitierten Veröffentlichungen.

1.3 Projektergebnisse: Übersicht

Der größte Teil der im Projektantrag genannten Ziele konnte erreicht werden oder im Rahmen des Projektes so vorbereitet werden, daß die zugehörigen Arbeitspunkte in zwei noch laufenden Dissertationsvorhaben abschließend bearbeitet werden können.

Die in Kapitel 3 skizzierte mathematische Beschreibung eines ereignisgesteuerten Simulators ist gleichermaßen ein Modell für die klassische sequentielle Simulation wie für die verteilte Simulation. Ein derartiges mathematisches Modell ist für eine präzise Festlegung der Korrektheit einer verteilten Ausführung einer Simulationsanwendung von Bedeutung.

Kapitel 4 gibt einen Überblick über die beiden im Rahmen des Projektes erstellten Anwendungssysteme DSL und DISQUE. Mit diesen umfangreichen Systemen können Simulationsmodelle mit unterschiedlichen Simulationsstrategien und auf unterschiedlichen Hardwareplattformen ausgeführt werden. Sie stellen die Voraussetzung für eine systematische experimentelle Analyse von Simulationsstrategien sowie der Analyse des möglichen Parallelisierungsgrades und weiterer wesentlicher Kenngrößen dar.

In Kapitel 5 werden neu entwickelte Simulationsstrategien oder Varianten bekannter Verfahren vorgestellt. Diese sind insofern interessant, als es sich dabei um Hybridformen der klassischen sogenannten konservativen und optimistischen Simulationen handelt.

Kapitel 6.1 skizziert Lösungsmöglichkeiten für das in der Literatur weitgehend vernachläs-

sigte Problem der Reproduzierbarkeit verteilter Simulationen. Es ergibt sich daraus, daß im Gegensatz zur klassischen sequentiellen Simulation ein verteilter Simulator bei Ereignissen mit gleichen Zeitstempeln a priori nicht deterministisch arbeitet.

In Kapitel 6.2 wird untersucht, wie innerhalb eines verteilten Simulationssystems globale Variablen realisierte werden können, die von mehreren der miteinander nur durch Nachrichten kooperierenden Submodellsimulatoren gemeinsam benutzt werden.

Kapitel 6.3 behandelt das algorithmische Problem der GVT-Approximation. Hierbei geht es im wesentlichen darum, das Minimum der asynchron fortgeschalteten logischen Uhren aller beteiligten Simulatoren möglichst gut zu approximieren. Bei optimistischen Simulationsverfahren ist dies einerseits für eine effiziente Implementierung von wesentlicher Bedeutung, andererseits jedoch nicht trivial, da einzelne Uhren bei einem sogenannten Rollback zurückgesetzt werden.

1.4 Projektverlauf

Das Projekt wurde an der Universität Kaiserslautern von J. Nehmer und F. Mattern initiiert und Mitte 1989 mit zunächst einem Mitarbeiter (J. Richter) begonnen. Mitte 1990 wurde von der DFG eine zweite Mitarbeiterstelle bewilligt, die von H. Mehl eingenommen wurde. Nach dem Wechsel von F. Mattern an die Universität des Saarlandes wurde das Projekt von Saarbrücken aus weitergeführt.

Während der gesamten Projektlaufzeit standen Mittel für eine studentische Hilfskraft zur Verfügung. Diese wurden für vielfältige subsidäre Programmieraufgaben eingesetzt. Ein Großteil der anspruchsvollen und umfangreichen Implementierungsaufgaben konnte im Rahmen von Projekt- und Diplomarbeiten durchgeführt werden [APE93a, ARM93a, BOC93a, ELS93a, HAM92a, KLI91a, KLU92a, LEO93a, MEH89a, SIM93a, STE93a, VIE92a, MEI91a, MEI92a, STU93a, SMI91a]. Das Projekt hat insofern auch einer Vielzahl von Studenten Gelegenheit zur Mitarbeit gegeben und leistete damit auch einen Beitrag zur forschungsorientierten Ausbildung.

Das Transputer-Parallelrechensystem, bestehend aus 16 Prozessoren, einer Graphikstation sowie entsprechender Software im Wert von knapp über 100.000 DM, welches von der DFG für das Projekt beschafft wurde, konnte wenige Monate nach Projektbeginn installiert werden. Leider wurden jedoch die Erwartungen, die in das System gesetzt wurden, nicht ganz erfüllt. Dies lag zum einen an der von uns als mangelhaft empfundenen Unterstützung des Herstellers, zum anderen an der Unzuverlässigkeit der Systemsoftware HELIOS, insbesondere bei Anwendungen mit hoher Dynamik, wie es beispielsweise das damit realisierte DISQUE-System darstellt. Häufige Systemzusammenbrüche (u.a. auch wegen unerkannter Speicherverletzungen), fehlende Dokumentation, langsame Kommunikation und fehlende Testmöglichkeiten erschweren eine Systementwicklung außerordentlich und kosten sehr viel Zeit.

Dennoch konnte das Transputersystem genutzt werden; im Rahmen der Parallel Computing Action der Europäischen Gemeinschaft (Esprit-Projekt 4123) wurden sogar Mittel zur Aufstockung des Systems bereitgestellt, mit dem so auch andere Aufgaben bearbeitet werden konnten. Die eigentlichen Entwicklungsarbeiten wurden im Verlauf des Projektes allerdings zunehmend auf vernetzte UNIX–Workstations verlagert, die aus Grundausstattunsmitteln beschafft wurden. Da an der Universität des Saarlandes ein iPSC860-Parallelrechner zur Verfügung stand, dessen Software stabiler und leistungsfähiger als die des Transputersystems ist, wurden schließlich große Teile der im Rahmen des Projektes realisierten Softwaresysteme unter Zuhilfenahme des MMK-Systems [BBL90a] auf diesen Rechner portiert und dieser Rechner für Beschleunigungsmessungen verwendet.

Erfreulicherweise konnte, wie dieser Bericht zeigt, einer Reihe von Fragestellungen nachgegangen werden, die im Projektantrag nicht enthalten waren, und dabei interessante Ergebnisse erzielt werden, die sich auch in Veröffentlichungen niederschlugen. Andererseits hat sich herausgestellt, daß eine wesentlich breitere Experimentierbasis als ursprünglich geplant geschaffen werden muß, um aussagekräftige Ergebnisse im Bezug auf erzielbare Beschleunigungswerte zu erhalten. Zusammen mit den oben angesprochenen Schwierigkeiten bei der Nutzung des Transputersystems ist dies die Ursache dafür, daß hierzu zwar gegenwärtig Experimente zur praktischen Evaluierung durchgeführt werden, aber noch keine Ergebnisse veröffentlicht wurden. Die Untersuchungen sind Gegenstand zweier laufender Dissertationsvorhaben, denen hier nicht vorgegriffen werden soll.

Nach Ende des Projektes werden die erzielten Ergebnisse nicht nur im Rahmen von Dissertationen verwendet und entsprechende Arbeiten weitergeführt, sondern das erworbene Know-How wurde auch in ein neues Teilprojekt des Sonderforschungsbereichs 124 "VLSI-Entwurf und Parallelität" zur parallelen VLSI-Simulation eingebracht, womit sich interessante Perspektiven zur Weiterbearbeitung der Thematik mit anderer Schwerpunktbildung ergeben.

Der DFG und den Gutachtern sei für die Unterstützung herzlich gedankt.

2 Verteilte und parallele Simulation - ein Überblick

Simulation ist ein wirkungsvolles, in weiten Bereichen der Technik sowie der Natur- und Wirtschaftswissenschaften eingesetztes Hilfsmittel. Insbesondere in der Informatik und der Operations-Research wird das Prinzip der zeitdiskreten ereignisgesteuerten Simulation verwendet, zu dem auch die prozeßorientierte Simulation (mit SIMULA67 als typischem Vertreter einer Simulationssprache) und die transaktionsorientierte Simulation (Bsp.: GPSS) gehören. Simulationen sind jedoch meist zeitaufwendig, sie gehören oft zu den Rechneranwendungen mit dem größten Rechenzeitbedarf. Ein Hauptgrund dafür ist, daß die Simulationsmodelle oft notgedrungen komplex sind, wenn sie die Realität genau genug widerspiegeln sollen, und der Rechenaufwand meistens probleminhärent in nichtlinearer Weise mit der Modellgröße zunimmt. Eine Beschleunigung der Simulation ist daher von großer praktischer Bedeutung

[MAM89a].

Das vorherrschende Paradigma zur Leistungssteigerung ereignisgesteuerter Simulationen stellt die sogenannte *parallele* oder *verteilte Simulation* dar [FUJ90a, RIW89a]. Dabei wird versucht, durch konkurrente Ausführung mehrerer Ereignisse eine Beschleunigung rechenintensiver Simulationsexperimente herbeizuführen. Zu diesem Zweck wird das zu simulierende Modell in möglichst unabhängige Teilmodelle partitioniert, wobei jedes Teilmodell durch einen in der Literatur üblicherweise *logischen Prozeß (LP)* genannten sequentiellen Simulator ausgeführt wird. Beispielsweise könnte ein LP ein Stadtviertel oder auch nur eine einzelne Kreuzung bei der Simulation des Straßenverkehrs einer Großstadt simulieren.

Als klassischer, sequentieller ereignisgesteuerter Simulator besteht ein LP im wesentlichen aus einem Zustand, einer Ereignisliste und einer logischen Uhr, die die jeweils aktuelle Simulationszeit des LP repräsentiert. Die Ereignisliste enthält stets diejenigen Ereignisse, von denen zum Zeitpunkt der logischen Uhr bekannt ist, daß sie von diesem LP zukünftig noch auszuführen sind, und zwar zu einem logischen Zeitpunkt, der dem jeweiligen Ereignis in Form eines "Zeitstempels" anhaftet. Zyklisch entfernt ein LP jeweils sein *nächstes* Ereignis e (d.h das Ereignis mit der kleinsten Eintrittszeit, falls vorhanden) aus seiner Ereignisliste, setzt die lokale Uhrzeit auf die Eintrittszeit von e und führt das Ereignis e (d.h. genauer, die mit e assoziierte Ereignisroutine) aus. Durch Ausführen von e wird i. allg. der lokale Zustand verändert und es können neue Ereignisse (mit zugehörigen Eintrittszeitpunkten, die nicht kleiner als der momentane Wert der logischen Uhr sind) in die lokale Ereignisliste oder die Ereignislisten anderer LPs eingeplant werden (etwa wenn in einer Straßenverskehrsimulation ein Auto von einem Stadtviertel in ein anderes fährt).

Das Einplanen eines Ereignisses bei einem anderen LP geschieht bei der verteilten Simulation durch das Versenden einer sogenannten Ereignisnachricht an diesen LP, da die einzelnen sequentiellen Simulatoren nur durch Kommunikation miteinander kooperieren können. Bei einer derartigen verteilten Ausführung muß der Effekt der konkurrenten Ereignisausführung durch mehrere LPs der gleiche sein wie bei chronologischer Ausführung aller Ereignisse durch einen einzigen Simulator. Das Hauptproblem besteht in der geeigneten Synchronisation der Simulationszeit der einzelnen Simulatoren, denn offenbar darf ein Simulator nicht einem anderen Simulator ein Ereignis mit einer Ausführungszeit einplanen, welcher zwar bezüglich des ersten Simulators in der Zukunft liegt, sich aber bezüglich der aktuellen Simulationszeit des zweiten Simulators bereits in der Vergangenheit befindet. Andererseits kann die Lösung des Problems auch nicht darin bestehen, daß alle Simulatoren vollkommen synchron bzgl. der Simulationszeit gehalten werden, da dann höchstens in dem seltenen Fall, daß zwei Ereignisse in zwei verschiedenen Simulatoren exakt die gleiche Eintrittszeit haben, zwei Ereignisse parallel ausgeführt werden können. Grundsätzlich lassen sich zur Lösung dieses Problems die existierenden *verteilten Simulationsalgorithmen* in zwei Klassen einteilen: *konservative* und *optimistische* Verfahren.

Bei einem konservativen Algorithmus führt ein logischer Prozeß LP_i das nächste Ereignis e seiner lokalen Ereignisliste erst dann aus, wenn er sicher ist, daß kein anderer LP ihm ein

Ereignis einplanen wird, welches vor e auszuführen ist. Da aus diesem Vorgehen folgt, daß ein LP u. U. warten muß, bevor er die Simulation fortsetzen kann, besteht prinzipiell die Gefahr eines Deadlocks. Konservative Verfahren lassen Deadlocks entweder zu und erkennen und beheben sie, oder sie vermeiden Deadlocks durch Austausch von Kontrollinformationen. Als Deadlockerkennungs- und Behebungsverfahren wurden in der Literatur sowohl Verfahren vorgestellt, die lokale [GRT87a, GRT88a, GRT91a] als auch solche die globale [CHM81a] Deadlocks erkennen und anschließend beheben. Eine Möglichkeit der globalen Deadlockbehebung besteht beispielsweise darin, das Ereignis mit dem global kleinsten Zeitstempel zu ermitteln und mit diesem die Simulation fortzusetzen. Nachteil dieses Verfahrens ist jedoch, daß alle am Deadlock beteiligten LPs bis nach der Deadlockbehebung nichts simulieren können. Deadlockvermeidende Verfahren tauschen *Garantien* über in Zukunft *nicht* mehr eingeplante Ereignisse aus. Sendet LP_i dem Simulator LP_j die Garantie G, so kann LP_j sicher sein, von LP_i kein weiteres Ereignis mit einem kleineren Zeitstempel als G eingeplant zu bekommen. Hat beispielsweise das nächste Ereignis e von LP_i den Zeitstempel t, so wird LP_i solange e nicht ausführen, wie er nicht von *allen* seinen *Nachbarn* (d.h. von all denjenigen LPs, die LP_i potentiell ein Ereignis einplanen können) eine Garantie $G \geq t$ erhalten hat.

Um Deadlocks zu vermeiden, müssen in ausreichendem Maße Garantien ausgetauscht werden. Garantien können dabei versendet werden sobald sie bekannt sind [MIS86a] oder erst wenn ein LP blockiert [FUJ88b]. Sie können jedoch auch explizit von Nachbarn angefordert werden [BAS88a, FUJ88b, SUS89a]. Da jedes Ereignis nur Ereignisse in der Gegenwart und der Zukunft erzeugen kann, haben alle von einem LP erzeugten Ereignisse immer einen Zeitstempel größer oder gleich der lokalen Uhrzeit. Ein LP kann daher prinzipiell immer seine lokale Uhrzeit als Garantie an andere LPs senden. Bessere Garantien können ggf. aus dem sogenannten *Lookahead* abgeleitet werden. Ein LP mit Uhrzeit t hat einen Lookahead L, wenn er alle Ereignisse kennt, die er bis zum Zeitpunkt $t + L$ erzeugen wird [CHM79b]. Die Lookahead–Information muß in der Regel vom Modellentwickler spezifiziert werden. Es zeigt sich, daß die Performanz konservativer Verfahren i. allg. sogar entscheidend von der verfügbaren Lookahead–Information abhängt [FUJ88b, FUJ88c]. Daher muß ein verteiltes Modell, welches mit einem konservativen Verfahren ausgeführt werden soll, in der Regel von vorneherein so entwickelt werden, daß überhaupt ausreichende Lookahead–Information spezifizierbar ist (siehe beispielsweise [NIR90a]). Läßt sich für eine Menge von LPs, die sich zyklisch Ereignisse einplanen können, kein Lookahead spezifizieren, so läßt sich dieses Modell von vielen deadlockvermeidenden Verfahren auch nicht verteilt simulieren; auch mit deadlockerkennenden Verfahren läßt sich bei solchen Modellen nur in Ausnahmefällen eine signifikante Beschleunigung erreichen. Ein Beispiel für eine solche "schwierige" Modellklasse stellen zyklische Warteschlagennetze mit exponentialverteilter Bedienstrategie dar.

Im Gegensatz zu konservativen Verfahren, die Kausalitätsverletzungen durch Warten auf hinreichend gute Garantien *a priori* vermeiden, lassen optimistische Verfahren solche Verletzungen grundsätzlich zu. Wird eine Kausalitätsverletzung erkannt, so wird sie *a posteriori* korrigiert. Prinzipiell nimmt ein optimistischer LP immer an, daß er sein gegenwärtig nächstes Ereignis ausführen darf. Dadurch ist es möglich, daß nach Ausführung eines Ereignisses e in

einem LP diesem weitere Ereignisse in seine Ereignisliste eingeplant werden, die er vor e hätte ausführen müssen. In diesem Fall liegt eine Kausalitätsverletzung vor, und alle Effekte verfrüht ausgeführter Ereignisse müssen rückgängig gemacht werden (*Rollback*). Beim Time Warp, dem bekanntesten Ansatz in dieser Klasse, wird ein Rollback dadurch durchgeführt, daß der lokale Zustand geeignet zurückgesetzt wird und verfrüht erzeugte Ereignisse durch das Versenden sogenannter *Anti–Nachrichten* annulliert werden [JES82a, JEF85a]. Solche Rollbacks sind jedoch nicht beliebig weit in die simulierte Vergangenheit möglich. Vielmehr läßt sich zu jedem Zeitpunkt der Simulationsausführung ein globales Minimum virtueller Zeit angeben, die sogenannte *Global Virtual Time* (GVT), derart, daß keine Rücksetzung mehr auf einen früheren virtuellen Zeitpunkt möglich ist. Die GVT ist eine monoton nicht fallende Funktion der Ausführungszeit des Simulationsmodells. Da kein Rollback auf eine Zeit vor der GVT möglich ist, ist die Berechnung einer möglichst engen unteren Schranke der GVT für Time Warp, aber auch für die meisten anderen optimistischen Verfahren, sehr wichtig.

Optimistische Verfahren unterscheiden sich im wesentlichen in der Art und Weise des Erkennens von Kausalitätsverletzungen [JEF85a, WES88a], in der Verwaltung von Rücksetzinformationen [BAS92a, PML92a], in der Durchführung von Rücksetzungen [FUJ89a, MWM88a, PRS92a, RFB90a], in der Berechnung von GVT–Approximationen [LIL90d, MAT93a, SAM85a, STE92a] und in dem Grad an Optimismus, mit dem sie bereit sind, potentiell noch rücksetzbare Ereignisse auszuführen [LSW89a, MHF92a, RWJ89a, SSH89a]. Eine effiziente Implementierung ist nicht einfach, jedoch scheint das zugrundeliegende Prinzip der spekulativen Ausführung die einzige Möglichkeit zu sein, für Modellklassen mit zyklischer Topologie, für die sich kein Lookahead angeben läßt, eine signifikante Beschleunigung zu erreichen.

3 Eine mathematische Beschreibung ereignisgesteuerter Simulation

Trotz der Vielzahl von Veröffentlichungen auf dem Gebiet der verteilten ereignisgesteuerten Simulation wurde das einer solchen Simulation zugrundeliegende Modell bisher meist gar nicht oder nur in unbefriedigender Weise erläutert. Dies erwies sich unter anderem als Problem bei der Bewertung und beim Vergleich der verschiedenen vorgeschlagenen Verfahren, da eine einheitlichen Beschreibungsmethode für die verwendeten Simulationsmodelle fehlte. Oft wurden auch gewisse Voraussetzungen an das Simulationsmodell gestellt, die jedoch nicht explizit ausgeführt wurden. Um eine exakte Begriffsbildung auf dem Gebiet der verteilten Simulation zu ermöglichen, wurde im Rahmen dieses Projektes der Begriff des ereignisgesteuerten Simulationsmodells mathematisch sowohl für sequentielle als auch für verteilte Simulationen exakt definiert. Diese Modelle können über den eigentlichen Zweck der Begriffsbildung hinaus sowohl als Grundlage für eine einheitliche Darstellung der Methoden der verteilten Simulation als auch zur Definition der Semantik verteilter Simulationssprachen dienen.

3.1 Grundlegende Ideen

Um zu verstehen, wie man zu den weiter unten beschriebenen mathematischen Modellen kommt, betrachte man die im vorherigen Kapitel bereits kurz skizzierte Arbeitsweise eines auf einer sequentiellen Maschine laufenden *Simulators für ereignisgesteuerte Simulation* (Abb. 1). Ein derartiges operationales Simulationsmodell besteht aus einer Menge von Prozeduren, den sogenannten *Ereignisroutinen*, und einer Menge von globalen Datenstrukturen, deren Werte den Zustand des zugrundeliegenden realen (d. h. des zu simulierenden) Systems beschreiben. Der Aufruf einer solchen Ereignisroutine entspricht dem Eintreten eines Ereignisses. Innerhalb der Ereignisroutine werden dann die globalen Datenstrukturen des Simulators manipuliert, um so den Effekt des im realen System stattfindenden Ereignisses im Simulationsmodell nachzubilden. Darüberhinaus werden im allgemeinen in einer solchen Ereignisroutine weitere Aufrufe von Ereignisroutinen veranlaßt. Diese Aufrufe sind mit *Zeitstempeln* versehen, die die Eintrittszeitpunkte der Ereignisse im realen System darstellen. Dies entspricht der Tatsache, daß in einer ereignisorientierten Sichtweise der Realität ein Ereignis weitere Ereignisse in der Zukunft zur Folge haben kann. (So kann beispielsweise in einer Straßenverkehrssimulation das Ereignis "Auto verläßt Kreuzung" das Ereignis "Auto kommt an Kreuzung an" für die Ankunftszeit des Autos an der nächsten Kreuzung veranlassen.) Alle auf diese Art erzeugten Aufrufe werden in einer zentralen Datenstruktur, der *Ereignisliste* zwischengespeichert. Der Simulator arbeitet dann nach folgendem Verfahren:

```
while Ereignisliste nicht leer
do
 hole Aufruf mit kleinstem Zeitstempel aus der Ereignisliste;
 führe die entsprechenden Ereignisroutine aus;
 füge die dabei erzeugten Aufrufe in die Ereignisliste ein;
od
```

Ein *verteilter Simulator* besteht im Prinzip aus einer Menge von sequentiellen Simulatoren, bei denen der Aufruf einer Ereignisroutine durch einen Simulator Ereignisse für beliebige Simulatoren erzeugen kann (also nicht nur für den Simulator, der die Ereignisroutine enthält, vgl. Abb. 2). Simuliert man z. B. den Straßenverkehr einer Stadt verteilt, wobei jeder einzelne Simulator für ein Stadtviertel zuständig ist, so kann es passieren, daß die Bearbeitung eines Ereignisses "Auto verläßt Kreuzung" ein Ereignis "Auto kommt an Kreuzung an" für den Simulator eines Nachbarviertels erzeugt, falls das entsprechende Auto von einem Stadtviertel ins nächste fährt. Dies führt zu den im vorherigen Kapitel angesprochenen Synchronisationsproblemen bei der verteilten Simulation, da ein Simulator nun erst dann den Aufruf mit dem kleinsten Zeitstempel seiner Ereignisliste durchführen kann, wenn er sicher ist, daß kein anderer Simulator noch Aufrufe mit kleineren Zeitstempeln für ihn erzeugt.

Diese operationale Sicht der ereignisgesteuerten Simulation spiegelt sich in den folgenden

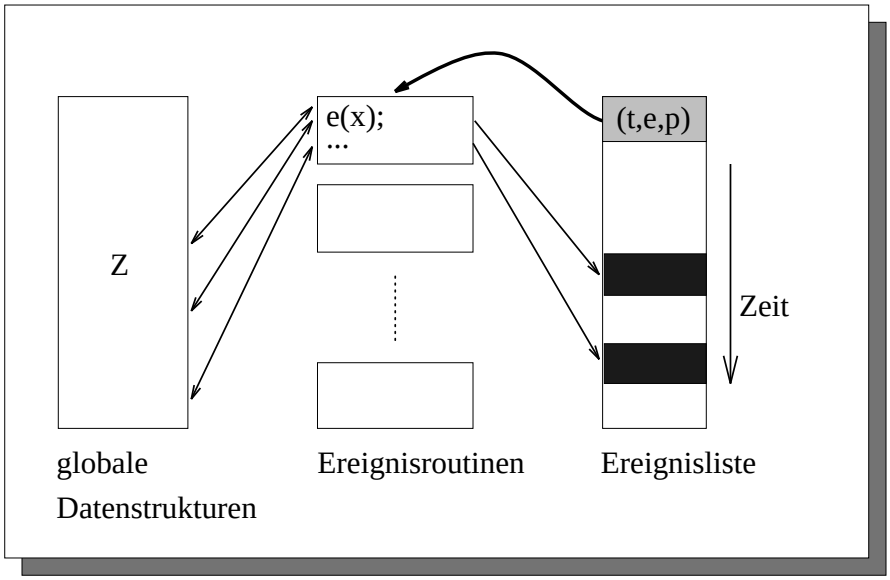

Abbildung 1: Sequentieller Simulator

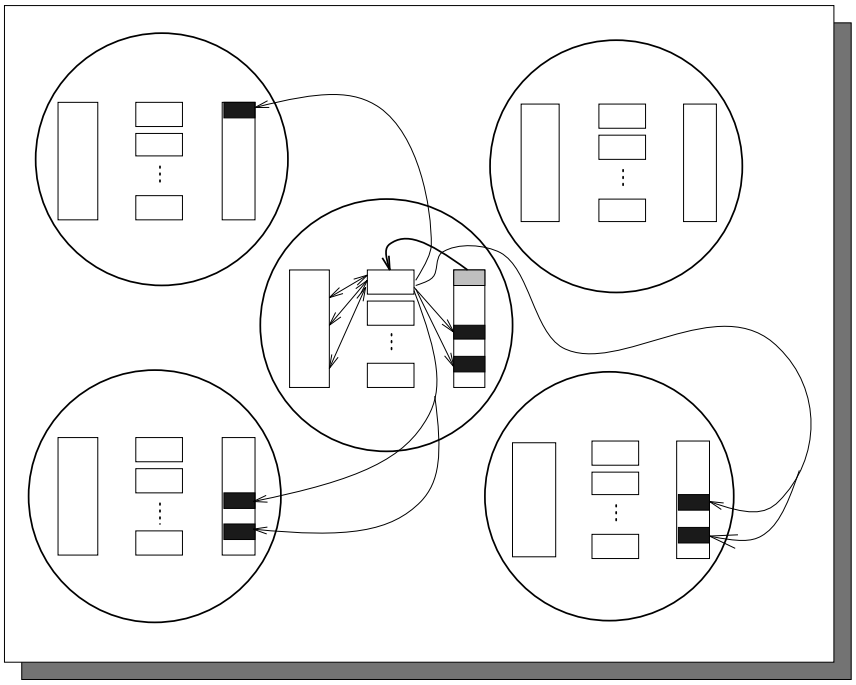

Abbildung 2: Verteilter Simulator

mathematischen Modellen wider.

3.2 Sequentielles Simulationsmodell

Ein **sequentielles Simulationsmodell** $\mathcal{S}$ ist ein 8-Tupel

$$(Z, E, P, z_s, I, \varphi, \tau, \pi)$$

wobei die einzelnen Komponenten folgende Bedeutungen haben:

- Z ist die Menge der möglichen Werte der Datenstrukturen des Simulators, $z_s \in Z$ ist der Wert dieser Datenstrukturen beim Start des Simulationslaufs, E die Menge der Bezeichner für Ereignisroutinen und P die Menge der Parameterwerte für diese Routinen.
- Ein Ereignis, d. h. ein Tripel $(t, e, p) \in \mathbb{R} \times E \times P$, stellt einen Eintrag in der Ereignisliste des Simulators dar, und zwar den mit dem Zeitstempel t versehenen Aufruf der Ereignisroutine e, bei dem der Parameter p an e übergeben wird. Aus Gründen der einfacheren Darstellung wurde für alle Ereignisroutinen nur ein einziger Parameter und eine gemeinsame Menge von Parameterwerten angenommen, was keine Einschränkung darstellt, da p beispielsweise ein Tupel von Parametern für e sein kann.
- Die Multimenge[1] $I \subset \mathbb{R} \times E \times P$ ist der Inhalt der Ereignisliste vor dem Start des Simulationslaufs. Da in Ereignislisten grundsätzlich dasselbe Ereignis mehrfach auftreten kann und die Anzahl der identischen Ereignisse angibt, wie oft die entsprechende Ereignisroutine aufgerufen werden muß, hat I statt einer Mengen- eine Multimengen-Struktur.
- Die Funktionen

  $$\varphi : \mathbb{R} \times E \times P \times Z \to Z$$

  und

  $$\tau : \mathbb{R} \times E \times P \times Z \to \mathcal{P}(\mathbb{R} \times E \times P)$$

  beschreiben das Verhalten der Ereignisroutinen. Arbeitet der Simulator im Zustand $z \in Z$ das Ereignis $(t, e, p) \in \mathbb{R} \times E \times P$ ab, so ist sein interner Zustand anschließend $\varphi(t, e, p, z)$. Insbesondere hat also die Ereignisroutine e außer auf den Parameter p und auf den internen Zustand z noch auf den Zeitstempel t Zugriff. Die Multimenge der von der Ereignisroutine e in diesem Fall neu erzeugten Ereignisse ist $\tau(t, e, p, z)$. Da hier die Möglichkeit bestehen sollte, ein Ereignis mehrfach zu erzeugen, wurde wiederum für $\tau(t, e, p, z)$ die Multimengen-Struktur gewählt.

[1] Anschaulich formuliert ist eine Multimenge eine Menge, in der Elemente mehrfach vorkommen dürfen. Für eine Menge M bezeichne $\mathcal{P}(M)$ die Menge aller Multimengen von M.

- $\pi : \mathbb{R} \times \mathcal{P}(E \times P) \rightarrow E \times P$ wird dazu benutzt, aus einer Multimenge von Ereignissen mit gleichem Zeitstempel ein Ereignis auszuwählen. Der gemeinsame Zeitstempel ist dann der erste Parameter von π, der zweite Parameter ist die Multimenge der Ereignisse ohne den Zeitstempel. Ein einfaches Beispiel für π ist z. B. die Vergabe von Prioritäten für die Ereignisroutinen.

Um unsinnige Simulationsmodelle auszuschließen, muß man noch einige Zusatzbedingungen an $\mathcal{S}$ stellen, die man [RIC90a] entnehmen kann. Für ein gegebenes Simulationsmodell $\mathcal{S}$ kann man dann den **logischen Ablauf** von $\mathcal{S}$ definieren, der aus einer Folge von Ereignissen besteht, die ein sequentieller Simulator erzeugen soll. Ein **korrekter Simulator** ist dann jedes Programm, das den logischen Ablauf von $\mathcal{S}$ berechnet.

3.3 Verteiltes Simulationsmodell

Für $n \in \mathbb{N}$ ist ein ***n*-fach verteiltes Simulationsmodell** $\mathcal{S}_n$ ein 8-Tupel

$$((Z_1, \ldots, Z_n), (E_1, \ldots, E_n), P, (z_s^1, \ldots, z_s^n), I, (\varphi_1, \ldots, \varphi_n), (\tau_1, \ldots, \tau_n), \pi)$$

wobei die 8-Tupel $(Z_i, E_i, P, z_s^i, \{(t, e, p) \in I | e \in E_i\}, \varphi_i, \tau_i, \pi)$ sequentielle Simulationsmodelle darstellen, die außer sich selbst auch anderen Modellen Ereignisse einplanen können. (Die Werte, die τ_i annehmen kann, sind Multimengen über der Menge *aller* Ereignisse.) Die Ereignisroutinen der einzelnen Teilmodelle können dabei nur auf ihren eigenen Zustand zugreifen, nicht auf die Zustände anderer Teilmodelle, wie man am Definitionsbereich der Funktionen τ_i und φ_i erkennt. Auch hier muß man noch einige Zusatzbedingungen an $\mathcal{S}_n$ stellen, um unsinnige Simulationsmodelle auszuschließen (vgl. wiederum [RIC90a]).

[RIC90a] enthält außerdem die Definition des logischen Ablaufs und des korrekten Simulators für $\mathcal{S}_n$. Dies ist von grundlegender Bedeutung für den Äquivalenzbegriff von sequentieller und verteilter Simulation. Weiterhin wird dort auch darauf eingegangen, wie man die "Granularität" eines Simulationsmodells vergröbert, indem man mehrere Teilsimulatoren zu einem einzigen Simulator verschmilzt und anhand eines weit verbreiteten Typs von Simulationsmodellen (Warteschlangennetze) gezeigt, wie sich ein solches Simulationsmodell mit den oben erläuterten Begriffen spezifizieren läßt.

4 Realisierung verteilter Simulation

Das Ziel des DFG-Projektes bestand neben theoretischen Untersuchungen einschließlich der Aufarbeitung und Bewertung veröffentlichter Verfahren vor allem darin, durch experimentelle Untersuchungen (Leistungsmessungen verschiedener Verfahren unter unterschiedlichen Bedingungen wie Zahl der Prozessoren, Hardwaretopologie etc.) die prinzipiellen Grenzen -

aber auch Chancen und Perspektiven - existierender und neuer paralleler und verteilter Simulationsverfahren zu ermitteln.

Eine empirische Untersuchung (Leistungsmessungen an Prototyprealisierungen) scheint ein unverzichtbarer Bestandteil einer umfassenden Untersuchung und Bewertung verteilter Simulationsverfahren zu sein; analytische Berechnungen sind nur für sehr einfache, nicht realistische Simulationsmodelle möglich [MIM84a], da die Verfahren - wie z. B. Fujimoto, Reed und Wagner in unabhängigen Experimenten gezeigt haben - teilweise empfindlich und in unerwarteter Weise auf die Änderungen in der Topologie, der Nachrichtenlaufzeit und anderer Parameter reagieren. Andererseits zeigen sich überraschende Effekte (z. B. den in [FUJ88b], [FUJ88c] beschriebenen Avalanche-Effekt, bei dem sich ab einer gewissen Nachrichtendichte die Zahl der beim Chandy-Misra-Bryant-Verfahren zu behebenden Blockierungen drastisch vermindert), die unter gewissen - analytisch nicht vorhersehbaren - Bedingungen eine deutliche Effizienzsteigerung hervorrufen.

Um die experimentellen Untersuchungen in realistischer Weise und in der notwendigen Breite durchführen zu können, wurden zwei verschiedene Anwendungssysteme konzipiert. Mit dem DSL-System können vor allem in flexibler Weise universelle Simulationssysteme realisiert werden und unterschiedliche Simulationsstrategien relativ zueinander verglichen werden. Dagegen wurde das DISQUE-System von vornherein auf eine spezifische Anwendungsklasse konzipiert. Der Vorteil dieses kleineren und flexibleren Systems besteht darin, daß es einfach auf unterschiedliche Hardwareplattformen portiert werden kann und es sich leicht instrumentieren und um Analysemöglichkeiten (Kritische-Pfad-Analyse, statistische Auswertung, Visualisierung) erweitern läßt. Beide Systeme, deren Anwendung zum Zweck der Analyse verteilter Simulationsverfahren z. Zt. noch Gegenstand laufender Dissertationsverfahren ist, werden im folgenden vorgestellt.

4.1 Das DSL–System

Zur experimentellen Untersuchung verteilter Simulationsverfahren wurde in der Zeit von 1989–1993 das *DSL–System* entwickelt. Zentraler Bestandteil des Systems ist die neue verteilte Simulationssprache *DSL* (*Distributed Simulation Language*) mit der beliebige ereignisgesteuerte verteilte Simulationsmodelle spezifiziert werden können. Das System enthält einen Compiler, welcher in DSL spezifizierte Modelle auf die verteilte Programmiersprache *CSSA*[2] abbildet und dabei wahlweise eines von mehreren verteilten Simulationsverfahren beibindet. Das so erhaltene CSSA-Programm stellt einen verteilten Simulator dar, der auf einem homogenen verteilten System von UNIX–Rechnern ausgeführt werden kann.

Das DSL–System ist so konzipiert, daß bekannte (aber auch neue) Simulationsverfahren anwendungs*un*abhängig programmiert werden können. Durch diese Trennung eignet sich das

[2]CSSA ist eine an der Universität Kaiserslautern entwickelte und implementierte verteilte Programmiersprache, die auf dem Aktorkonzept von Hewitt aufbaut [HEW77a, MAT88c].

DSL–System als *Plattform*, um verschiedene Simulationsverfahren zu analysieren: Ein in DSL geschriebenes Modell kann durch alle Simulationsverfahren ausgeführt werden, *ohne* Änderungen an der Modellbeschreibung vornehmen zu müssen. Auch der Compiler muß i.a. nicht geändert werden, um neue Simulationsverfahren zu realisieren. Im folgenden werden die Komponenten des DSL–Systems kurz vorgestellt. Eine ausführliche Beschreibung der Sprache DSL kann [MEH93b] entnommen werden.

4.1.1 Der DSL–Compiler

Der DSL–Compiler ist in der Programmiersprache C unter Verwendung der Scanner- und Parsergeneratoren FLEX und BISON geschrieben [KLU92a, VIE92a]. Aus einem DSL–Programm wird intern ein attributierter Programmbaum aufgebaut und daraus entsprechender CSSA-Code erzeugt. Das zur Übersetzungszeit jeweils ausgewählte Simulationsverfahren liegt in Form von CSSA–Codefragmenten vor und wird dabei lediglich an geeigneten Stellen textuell in den erzeugten Code eingestreut. Durch diese Technik können Compiler–Änderungen bei neuen Simulationsverfahren weitestgehend vermieden werden.

Die Abbildung von DSL auf CSSA wurde vorgenommen, um das DSL–System schneller realisieren zu können. CSSA bietet als verteilte Sprache bereits flexible Konstrukte zur Erzeugung von Prozessen (*Agenten* in CSSA) und zum synchronen oder asynchronen Versenden von Nachrichten an. Auch das *Facettierungskonzept* und sogenannte *Assertions*, die ähnlich *Guarded Commands* zur Laufzeit erlauben, die Verarbeitung bereits empfangener Nachrichten zurückzustellen bis ein boole'sches Prädikat über dem lokalen Prozeßzustand und dem Nachrichteninhalt erfüllt ist, wurden vielfach bei der Implementierung der Simulationsverfahren in CSSA ausgenutzt.

Einige Konzepte, die zur Realisierung der Simulationsverfahren notwendig waren, ließen sich in CSSA jedoch nicht von vorneherein in effizienter Weise realisieren. Hierzu gehört beispielsweise das für die spekulative Simulation (vgl. Kap. 5.2) benötigte Erkennen von Untätigkeitsphasen von Prozessoren. Da jedoch das Laufzeitsystem von CSSA an der Universität Kaiserslautern entwickelt wurde [ELS93a, MEH89a, WOL88a], waren die notwendigen Kenntnisse vorhanden, um die Sprache schnell um die gewünschten Eigenschaften zu erweitern.

4.1.2 Die Sprache DSL

Präambel. Ein DSL–Programm besteht i.w. aus zwei Teilen: Einer *Präambel* und einer Menge von *Modelltypbeschreibungen*. Die Präambel erlaubt die Beschreibung globaler Aspekte eines Simulationsexperiments. Dazu gehören die Erzeugung von Instanzen eines Modelltyps (LPs), deren Plazierung auf Prozessoren (Mapping) und die Beschreibung der erlaubten Kommunikationsbeziehungen durch Ereignisnachrichten (Topologie) oder durch gemeinsame Variablen. Ferner kann optional der Start– und die Endezeit der Simulation angegeben werden.

Das folgende Beispiel zeigt die Präambel eines künstlichen Simulationsmodells, welches von 16 LPs ausgeführt wird. Die Topologie des verteilten Simulators besteht aus einem 4x4 Torus, d.h. ein LP kann jeweils vier benachbarten LPs Ereignisse einplanen ('$' leitet eine Kommentarzeile ein; Schlüsselwörter der Sprache sind klein geschrieben):

```
$=========================================================================$
$ Praeambel eines kuenstlichen Simulationsmodells (4x4 Torus)             $
$=========================================================================$

$-------------------------------------------------------------------------$
$ Namen aller Modelltypen von denen LPs Instanzen sind                    $
$                                                                         $
$ (In diesem Beispiel gibt es nur einen einzigen Modelltyp)               $
$-------------------------------------------------------------------------$
models: LP_TYP;

$-------------------------------------------------------------------------$
$ Lokale Deklarationen                                                    $
$-------------------------------------------------------------------------$
var int: I,J;
const int: G := 4;  $ Torusgroesse

$-------------------------------------------------------------------------$
$ Erzeugung der LPs                                                       $
$                                                                         $
$ ('LP' ist ein zweidimensionales Feld von LPs vom Typ 'LP_TYP')          $
$-------------------------------------------------------------------------$
simulator LP_TYP: LP[1..G,1..G];

$-------------------------------------------------------------------------$
$ Beschreibung erlaubter Kommunikationsbeziehungen und Mapping            $
$                                                                         $
$ ("mapw X on i" bewirkt die Zuordnung von LP X auf den Rechner, dessen   $
$  Namen in einer Konfigurationsdatei in der i-ten Zeile angeben ist;     $
$  "addlink(X,Y)" erlaubt LP X potentiell Ereignisse in LP Y einzuplanen)$
$-------------------------------------------------------------------------$
loop for I in 1..G do
  loop for J in 1..G do
      mapw    LP[I,J] on ((J-1) * G + I);   $ Mapping
      addlink (LP [I,J]          , LP [MOD(I+1,G), J]);
      addlink (LP [I,J]          , LP [I, MOD(J+1,G)]);
      addlink (LP [MOD(I+1,G), J], LP [I,J]);
      addlink (LP [I, MOD(J+1,G)], LP [I,J]);
```

```
  endloop;
endloop;

$-------------------------------------------------------------------------$
$ Sonstiges                                                               $
$-------------------------------------------------------------------------$
simulation starts at 10.0 ends at 1000.0;
```

Modelltypbeschreibung. Ein verteiltes Simulationsmodell besteht in der Regel aus Instanzen mehrerer verschiedener Modelltypbeschreibungen. Um die Übersetzungszeit größerer Simulationsmodelle zu reduzieren, wurde auf die *getrennte Übersetzbarkeit* von Modelltypbeschreibungen geachtet. Eine *Modelltypbeschreibung* eines LPs besteht aus einem *Definitionsmodul* und einem *Implementierungsmodul*.

Das Definitionsmodul enthält die Namen und Parameterlisten von Ereignissen, die im Implementierungsmodul definiert sind. Durch diese Schnittstellenbeschreibung kann der DSL–Compiler beispielsweise überprüfen, ob die zu erzeugenden Ereignisse und deren Parameterlisten eine Typstruktur haben, die im empfangenden LP bekannt ist.

Das Implementierungsmodul enthält i.w. die Beschreibung des Zustandsraums eines LPs und eine Menge von Ereignisroutinen. Zwei spezielle Ereignisroutinen `start` und `finish` werden automatisch vor Beginn bzw. nach dem Ende der Simulation durch einen LP ausgeführt, sofern diese Routinen definiert wurden. Dadurch können LPs gezielt initialisiert und Statistiken am Ende der Simulation ausgegeben werden.

Eine Ereignisroutine kann (mit wenigen Ausnahmen) den vollen Sprachumfang von CSSA nutzen. Insbesondere stehen Befehle zur Durchführung von Arithmetik, Stringverarbeitung, Mengenverarbeitung, Pointerzugriffen, Prozeduraufrufen, Funktionsaufrufen sowie verschiedene flexible Kontrollstrukturen zur Verfügung. Zur Ein-/Ausgabe werden eigene Befehle angeboten, da diese bei manchen Simulationsverfahren (wie etwa bei Time Warp) besonders behandelt werden müssen. Daneben existiert eine Schnittstelle zu der Programmiersprache C, so daß in Ereignisroutinen auch extern definierte C-Funktionen und Prozeduren aufgerufen werden können.

Neben diesen allgemeinen Sprachkonstrukten sind besonders die simulationsspezifischen Sprachkonstrukte hervorzuheben. Beispielsweise kann mittels des `schedule`–Befehls ein Ereignis zu einer bestimmten Simulationszeit in einem LP eingeplant werden. Durch `reschedule` kann die Eintrittszeit nachträglich geändert werden, und durch `cancel` kann ein bereits eingeplantes Ereignis wieder annulliert werden (in konservativen Methoden allerdings nur dann, wenn es noch nicht ausgeführt wurde, da es konzeptionell dort keine Möglichkeit gibt, Effekte bereits ausgeführter Ereignisse rückgängig zu machen).

Ferner enthält DSL eine Reihe neuartiger Befehle, die es dem Modellierer erlauben, dem Simulator Wissen über das Modell mitzuteilen. Hierdurch kann in Experimentserien untersucht werden, inwieweit solches Wissen zur Leistungssteigerung beiträgt. Beispielsweise kann in DSL auf vielfältige Weise Lookahead–Information spezifiziert werden, von der für konservative Verfahren bereits bekannt ist, daß sie entscheidenden Einfluß auf die Performanz hat [FUJ88b, FUJ88c]. Aber auch optimistische Verfahren können Lookahead–Information ausnutzen. So kann bei dem `schedule`–Befehl anstelle eines Zeitpunkts ein virtuelles *Eintrittsintervall* angegeben werden. Damit drückt der Modellierer aus, daß die Ausführung eines solchen Ereignisses irgendwann *innerhalb des Zeitintervalls* stattfinden kann. Durch die Abschwächung der genauen Eintrittszeit wird beispielsweise bei Time Warp eine Einsparung von Rollbacks erwartet.

Schließlich gibt es in DSL auch eine Reihe von Zufallszahlengeneratoren für Gleich-, Exponential-, Erlang- und Normalverteilung nach in der Literatur beschriebenen Standardverfahren [NEE87a, PAG91a, PAM88a].

4.1.3 Verfahrens- und Modellbibliotheken

Das DSL–System läuft mit mehreren konservativen, hybriden und optimistischen Simulationsverfahren. Alle Simulationsverfahren verwenden das in Kapitel 6.1 skizzierte deterministische Tie–Breaking Verfahren, so daß Simulationen unter DSL reproduzierbar ablaufen. Die folgende Liste gibt einen Überblick über die Verfahren, die verfügbar sind bzw. kurz vor ihrer Fertigstellung stehen; desgleichen werden die z. Zt. vorhandenen Simulationsmodelle aufgelistet.

1. Sequentielle Simulationsverfahren

   Um die Effizienz verteilter Simulationsverfahren beurteilen zu können, wurde ein sequentieller Simulator realisiert. Wie die weiter unten aufgeführten verteilten Simulationsverfahren kann dieser alle in DSL spezifizierten Simulationsmodelle ausführen, ohne Änderungen am Modell vornehmen zu müssen. Zur Ereignislistenverwaltung kann wahlweise die gleiche Methode wie bei den verteilten Verfahren verwendet werden oder eines der z. Zt. besten Verfahren zur Ereignislistenverwaltung bei sequentiellen Simulationen, die Calendar Queue von Brown [BRO88a].

2. Verteilte Simulationsverfahren

   - konservative Verfahren
     - Versenden von Garantien, sobald sie bekannt sind [MIS86a]
     - Versenden von Garantien, wenn ein LP blockiert [FUJ88c]
     - Versenden von Garantien, wenn ein LP blockiert, wobei gemeinsame Variablen erlaubt sind ([SIM93a] nach Algorithmen in [MEH92e, MEH93c])

  - Versenden von Garantien, wenn ein Prozessor blockiert
  - Versenden von Garantien, sobald ein Deadlock eintritt [CHM81a]
  - Versenden von Garantien nach einer Variante des TNE-Verfahrens [GRT91a, APE93a]
  - hybride konservative Verfahren, die Deadlock-Erkennungs- und Behebungsverfahren mit „sporadischem“ Versenden von Garantien oder Nachfragen von Garantien kombinieren
- optimistische Verfahren
  - Time Warp mit „Aggressive Cancellation“ [JEF85a]
  - Time Warp mit „Aggressive Cancellation“, wobei gemeinsame Variablen erlaubt sind ([SIM93a] nach Algorithmen in [MEH92e, MEH93c])
  - in Vorbereitung: Time Warp mit „Lazy Cancellation“ [BOC93a]
- hybride Verfahren
  - spekulative Simulation in allen weiter unten beschriebenen Varianten [MEH91a, STE93a]

3. Simulationsmodelle
   - künstliche Modelle (skalierbare geschlossene Warteschlangennetze, beispielsweise mit Torus- oder zyklisch verketteter Baumstruktur) [APE93a]
   - realistische, generische Straßenverkehrssimulation [LEO93a]
   - Simulation einer hypothetischen Fabrik [ARM93a]

Mit Messungen zum Zweck des Vergleichs der verschiedenen Simulationsverfahren wird derzeit begonnen. Diese sollen Aufschluß über die Eignung der unterschiedlichen Simulationsverfahren für bestimmte Anwendungsklassen und Modellstrukturen geben sowie Hinweise zur erzielbaren Beschleunigung liefern.

4.2 Das DISQUE-System

Warteschlangennetze sind eine weit verbreitete Klasse von Simulationsmodellen, die ihre Hauptanwendungsgebiete im Bereich der Leistungsanalyse von Betriebssystemen, Rechnernetzen und Materialflußsystemen haben. Im Rahmen des DFG-Projektes wurde ein Testbett namens DISQUE (=**Di**stributed **S**imulator for **Que**ueing Networks) entwickelt [RIC91a], um verschiedene Strategien zur Parallelisierung ereignisgesteuerter Simulationen im Hinblick auf ihre praktische Verwendbarkeit für die Simulation von Warteschlangennetzen zu untersuchen. Ziel war, Möglichkeiten und Grenzen der erzielbaren Beschleunigung zu erkunden sowie verschiedene Werkzeuge zu entwickeln, mit denen sich Parallelisierungsgrad sowie inhärente Sequentialität der Modelle ermitteln lassen.

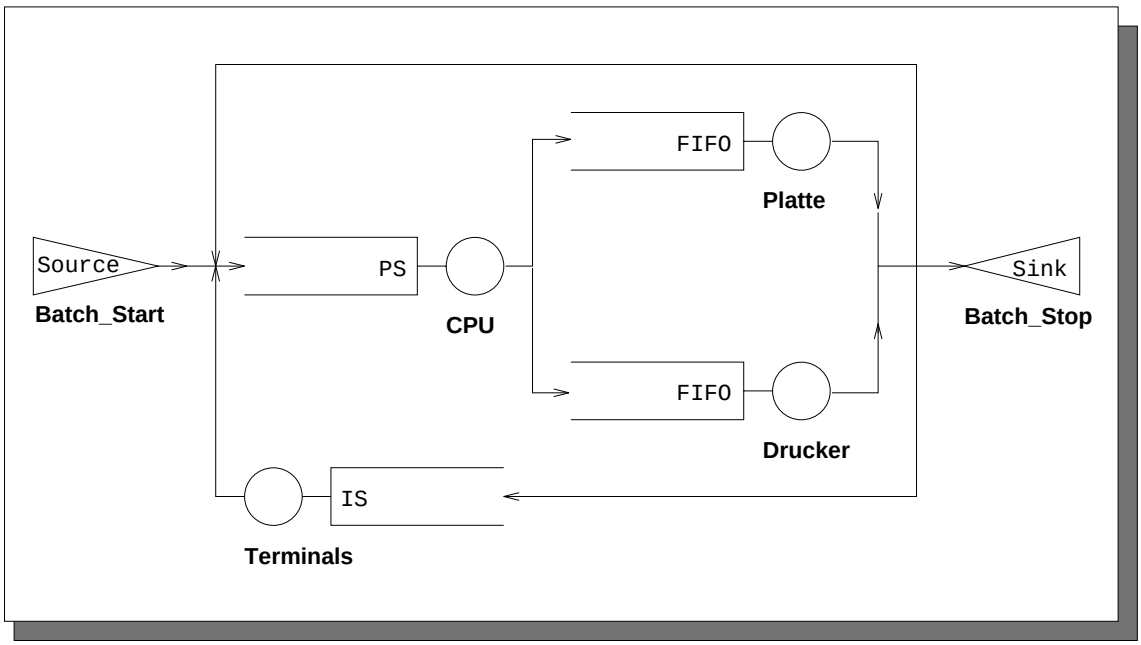

Abbildung 3: Computersystem als Warteschlangennetz

4.2.1 Warteschlangennetze

Ein Warteschlangennetz besteht aus einer Menge von Warteschlangen, die durch Kanäle miteinander verbunden sind. In diesem Netz kreisen Kunden. Jede Warteschlange besteht aus einem oder mehreren Warteräumen für Kunden, sowie einer Bedienstation. Das Verhalten der Bedienstation ist dabei durch die Bedienstrategie und die statistische Verteilung der Bedienzeiten der Kunden gegeben. Die Bedienstrategie legt fest, nach welchen Regeln die Kunden aus den Warteräumen geholt werden (FIFO, prioritätsgesteuert etc.). Die Bedienzeit der Kunden, d. h. die Zeit, die ein Kunde in der Bedienstation verbringt, wird in DISQUE durch eine statistische Verteilung spezifiziert. In der Simulation werden dann für die Bedienzeit der einzelnen Kunden entsprechend verteilte Zufallszahlen erzeugt. Insgesamt wird in Warteschlangennetzen also nur spezifiziert, wann und wie lange, aber nicht wie ein Kunde bedient wird. Ein Kunde, dessen Bedienung an einer Wartschlange beendet ist, verläßt diese augenblicklich und begibt sich (in Nullzeit) zu einer anderen Warteschlange, die er u. U. aus mehreren Möglichkeiten gemäß vorgegebener Wahrscheinlichkeiten zufällig auswählt, oder verschwindet in einer Senke. Eine Anfangsbelegung von Warteschlangen mit Kunden zu Beginn der Simulation ist in DISQUE ebenso spezifizierbar wie Kunden-Quellen, die Kunden während der Simulation neu erzeugen, wobei die Zeit zwischen dem Erzeugen zweier Kunden wiederum durch eine statistische Verteilung spezifiziert wird.

Abb. 3 zeigt ein kleines Beispiel, bei dem ein Computersystem mit Batchverarbeitung und Dialogbetrieb modelliert wurde. Das Ergebnis der Simulation eines Warteschlangennetzes ist

eine Statistik, die für jede Warteschlange Auslastung und Durchsatz der Bedienstation, mittlere und maximale Länge des bzw. der Warteräume sowie die maximale Wartezeit eines Kunden enthält. Falls gewünscht, werden an gewissen Warteschlangen nur ein Teil dieser Größen bestimmt. Mögliche Fragestellungen, die ein Simulationslauf für das Beispiel in Abb. 3 beantworten könnte, sind z. B. die Frage, ob die CPU leistungsstark genug ist (Auslastung der CPU) oder wie groß der Puffer für den Drucker sein muß (maximale Länge der Warteschlange des Druckers). Abb. 4 zeigt die Beschreibung des Beispiels in der Sprache LAVENDER (*La*nguage for *V*arious Qu*e*ueing *N*etwork *De*scr*i*ptions), die in Anlehnung an die Sprache RESQ2 von IBM als Eingabesprache für DISQUE entwickelt wurde.

4.2.2 Die Komponenten von DISQUE

Ziel bei der Implementierung von DISQUE war nicht, einen praktisch einsetzbaren Warteschlangennetz-Simulator zu schaffen, sonder ein Testbett zur Verfügung zu haben, mit dem sich verschiedene Strategien der Parallelisierung ereignisgesteuerter Simulationen unter realistischen Bedingungen auf ihre Tauglichkeit hin testen lassen. Daher wurde auf die sonst oft übliche graphische Eingabe verzichtet und auch die Modellwelt bewußt einfach und überschaubar gehalten. Statt dessen wurden zusätzlich zu der parallelen Version des Simulators noch mehrere sequentielle Simulatoren entwickelt, mit denen sich relevante Kenngrößen ermitteln lassen.

Sequentielle Simulatoren. Um verschiedene Vergleichsmaßstäbe für die Messungen am verteilten Simulator zu haben, wurden drei verschiedene sequentielle Simulatoren implementiert:

- Sequentieller Simulator
- Optimierter sequentieller Simulator
- Kritischer-Pfad-Analysator

Der **sequentielle Simulator** benutzt dieselben Ereignisse wie der weiter unten skizzierte verteilte Simulator.

Der **optimierte sequentielle Simulator** nutzt die Tatsache aus, daß sequentielle Simulatoren auf einem globalen Zustandsraum arbeiten, indem er den Abgang eines Kunden an einer Warteschlange und die Ankunft desselben an der nächsten Warteschlange als ein einziges Ereignis modelliert, was z. B. die Zahl der Zugriffe auf die Ereignisliste reduziert. Die Differenz zwischen den Laufzeiten des sequentiellen und des optimierten sequentiellen Simulators liefert damit ein Maß für den zeitlichen Mehraufwand, der in den sequentiellen Simulator investiert werden muß, um den Code "parallelisierbar" zu machen. Außerdem dient die Laufzeit des optimierten sequentiellen Simulators als Basis für "realistische" Beschleunigungsmessungen.

```
model Computer_System

queues
        queue Terminals
                type active
                discipline is
                classes class iTerminals distribution negexp 3000.0
        queue CPU
                type active
                discipline ps
                classes class iCPU distribution negexp 3.1
                        class bCPU distribution negexp 8.2
        queue Drucker
                type active
                discipline fifo
                classes class iDrucker distribution normal 200.1 100.6
                        class bDrucker distribution normal 300.6 100.3
        queue Platte
                type active
                discipline fifo
                classes class iPlatte distribution constant 7.0
                        class bPlatte distribution constant 7.0

topology
        from iTerminals to iCPU
        from iCPU to iDrucker iPlatte probabilities 0.1 0.9
        from iPlatte to iCPU iTerminals probabilities 0.7 0.3
        from iDrucker to iCPU iTerminals probabilities 0.7 0.3

        from source to bCPU
        from bCPU to bDrucker bPlatte probabilities 0.3 0.7
        from bDrucker to sink bCPU probabilities 0.8 0.2
        from bPlatte to sink bCPU probabilities 0.8 0.2

population
        at iTerminals 10
        source of bCPU with distribution constant 40000.0

statistics
        at CPU record utilisation
        at Drucker record maxql

terminate
        clock limited to 1000000.0

end
```

Abbildung 4: LAVENDER-Beschreibung des Computersystems

Beim **Kritischen-Pfad-Analysator** handelt es sich um eine instrumentierte Version des sequentiellen Simulators, der die CPU-Zeit, die für die Ausführung von Ereignissen benötigt wird, mißt und daraus berechnet, wie schnell der Simulationslauf auf einem Parallelrechner hätte ausgeführt werden können. Er tut dies unter folgenden Annahmen:

1. Jeder Simulator besitzt eine einzige Warteschlange und hat seinen eigenen Prozessor.
2. Die Zeiten für Nachrichtenübertragungen und Verwaltung von Ereignislisten werden ignoriert.
3. Mit der Ausführung eines Ereignisses wird zum frühest möglichen Zeitpunkt begonnen. (Das heißt, es wird beispielsweise nicht auf Garantien gewartet.)

Diese Annahmen sind natürlich für einen realen Parallelrechner hypothetisch, so daß das Ergebnis dieser Berechnungen immer eine obere Schranke für tatsächliche erreichbare Beschleunigungen sein wird. Der Wert dieser Analyse liegt vielmehr darin, daß man mit ihr Simulationsmodelle ermitteln kann, die "inhärent sequentiell" sind, d. h. deren verteilte Simulation im wesentlichen einer sequentiellen Simulation entspricht, die abwechselnd auf unterschiedlichen Rechnerknoten stattfindet. Ein einfaches Beispiel dafür ist die Simulation eines beliebigen Warteschlangennetzes, in dem nur ein einziger Kunde zirkuliert.

Verteilter Simulator. Die Probleme und Lösungsstrategien bei der verteilten ereignisgesteuerten Simulation wurden bereits in den vorangegangenen Kapiteln angesprochen. Die Idee bei der Konzeption von DISQUE war, diese recht unterschiedlichen Strategien an denselben Simulationsmodellen anwenden zu können, um so vergleichende Messungen zu ermöglichen. Daher wurde der Simulator von vornherein so ausgelegt, daß die Teile des Code, die bei der jeweiligen Simulationsstrategie unterschiedlich implementiert werden mußten, streng gekapselt sind. Die Idee bei dieser Art der Kapselung stammt aus [DRE89]. Die Idee dabei ist, in den normalen sequentiellen Simulationsalgorithmus an bestimmten Stellen Funktionsaufrufe (sogenannte Filter) einzubauen, in denen die eigentliche Simulationsstrategie codiert ist. Die drei wichtigsten Funktionen sind die folgenden:

- Dem *In-Filter* werden alle ankommenden Nachrichten übergeben. Seine Aufgabe ist es, die Ereignisnachrichten als Ereignisse in die Ereignisliste des Simulators einzufügen und die eingehenden Kontrollnachrichten (z. B. Garantien anderer Simulatoren, Anti-Nachrichten, etc.) entsprechend zu verarbeiten.
- Dem *Out-Filter* werden alle vom Simulator erzeugten Ereignis-Nachrichten übergeben, die dieser dann an die anderen Simulatoren verschickt und gegebenenfalls weitere Nachrichten (z. B. Garantien) verschickt.
- Der *Sim-Filter* wird vom Simulator vor der Ausführung jedes Ereignisses aufgerufen. Er blockiert den Simulator so lange, bis dieser das früheste Ereignis seiner Ereignisliste abarbeiten darf (z. B. weil ausreichende Garantien vorliegen).

Bislang sind Filter für die Simulationsstrategien Deadlock-Avoidance, Deadlock-Detection, SRADS [SMI91a] und Timewarp [MEI92a] entwickelt worden. Damit stehen mehrere verteilte Simulatoren zur Verfügung, deren Leistungsfähigkeit für unterschiedliche Modellstrukturen miteinander verglichen werden kann. Ferner wurden Analysewerkzeuge entwickelt, mit denen Einblicke in die internen Vorgänge innerhalb der einzelnen Simulatoren während einer Simulation (Verlauf von Garantien, Häufigkeiten von Rollbacks, etc.) gewonnen werden können [STU93a]. Systematische Auswertungen werden gegenwärtig durchgeführt, erste vorläufige Ergebnisse zeigen jedoch, daß es bedingt durch den starken Einfluß des zu simulierenden Modells sowie von Systemgrößen wie Nachrichtenlaufzeiten und Kosten für hardwareabhängige Operationen (z. B. Interrupts, Cache-Kohärenz) schwierig ist, zu allgemeingültigen Aussagen zu kommen. Die Veröffentlichung dieser Ergebnisse ist zu einem späteren Zeitpunkt geplant.

4.2.3 Implementierung

Sämtliche Teile von DISQUE sind in C programmiert. Begonnen wurde mit der Entwicklung des sequentiellen Simulators auf einer SUN–Workstation unter SUNOS 4.03 (UNIX). Anschließend wurde der sequentielle Simulator auf ein Transputersystem unter HELIOS 1.19 portiert [MEI91a].

Auf der Basis dieses Simulators entstanden auf dem Transputer-Parallelrechner der optimierte sequentielle Simulator, der Kritische-Pfad-Analysator sowie der verteilte Simulator. Alle diese Varianten wurden anschließend zu Debugging-Zwecken zusätzlich auf eine SUN–Workstation portiert, wobei dort der verteilte Simulator nur "pseudoverteilt" als System von kommunizierenden UNIX–Prozessen läuft.

Mittlerweile wurde das gesamte System im Rahmen eines Projektes des Sonderforschungsbereichs 124 auch auf den Intel iPSC860–Parallelrechner sowie auf ein Netzwerk von SUN–Workstations portiert. Dabei wurde das an der TU München im Rahmen des TOPSYS–Projektes entwickelte MMK-System eingesetzt [BBL90a].

5 Neue Simulationsverfahren

Die Entwicklung und Erprobung von neuen verteilten Simulationsverfahren sowie von Varianten und Kombinationen bekannter Verfahren stellte von vornherein eines der engeren Ziele des Projektes dar. Einige dieser Verfahren werden nachfolgend kurz vorgestellt; eine eingehende Diskussion findet sich in der erwähnten Literatur. Da derartige Verfahren nur empirisch evaluiert werden können, mit dementsprechenden Experimenten jedoch soeben erst begonnen wurde, ist es für eine Wertung allerdings noch zu früh.

5.1 Hybride konservative Verfahren

Konservative Simulationsverfahren vermeiden entweder Deadlocks durch Austausch von Garantien oder sie lassen Deadlocks zu und erkennen und beheben sie. Die erstgenannte Klasse von Verfahren hat den Nachteil, hinreichend viele Garantien austauschen zu *müssen*, um Deadlocks zu vermeiden. Die zweite Klasse hat den Nachteil, daß am Deadlock beteiligte LPs mindestens bis nach der Deadlockbehebung nichts simulieren. Beide Nachteile können durch Kombination von Verfahren aus beiden Klassen abgeschwächt werden. Eine solche hybride konservative Methode enthält zum einen ein Verfahren, mit dem Deadlocks erkannt und behoben werden können. Zusätzlich werden jedoch Garantien ausgetauscht und so die Deadlock–Wahrscheinlichkeit deutlich reduziert. Da Deadlocks prinzipiell auftreten dürfen, muß jedoch nicht mehr ein Mindestmaß an Garantien ausgetauscht werden, um Deadlocks zu vermeiden. Vielmehr können die Garantie–Austauschschemata deadlockvermeidender Simulationsverfahren adaptiv beliebig abgeschwächt und kombiniert werden; die ausgetauschten Garantien dienen lediglich dazu, den Grad an potentiell erreichbarer Parallelität zu erhöhen. Bei einem Garantieanfrageschema könnte somit beispielsweise die Nachfrage nach Garantien und damit der Gesamtaufwand für Garantieanfragen auf einige wenige "Hops" beschränkt werden (Abb. 5).

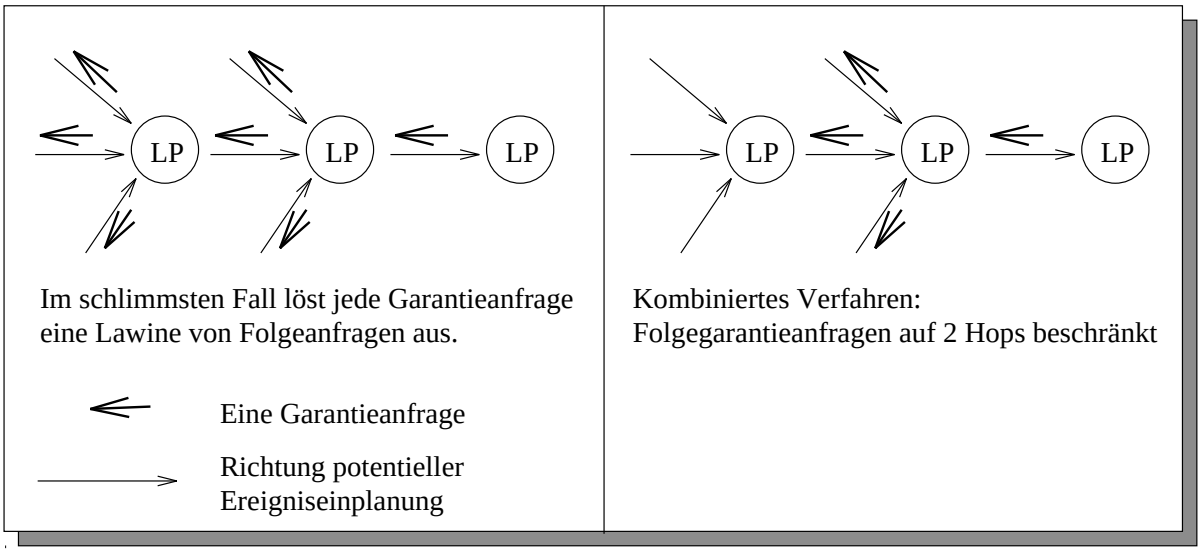

Abbildung 5: Ein hybrid–adaptives konservatives Verfahren.

5.2 Hybride konservativ–optimistische Verfahren (spekulative Simulation)

Wie bereits in Kapitel 2 skizziert, lassen sich verteilte Simulationsmethoden im wesentlichen als konservativ oder optimistisch einstufen. Konservative Verfahren vermeiden Kausalitätsverletzungen durch hinreichend langes Warten auf Garantien. Optimistische Methoden hingegen

führen Ereignisse auch dann aus, wenn ein konservatives Verfahren noch warten würde. War der Optimismus ungerechtfertigt, müssen jedoch die Effekte verfrüht ausgeführter Ereignisse und der von ihnen erzeugten Folgeereignisse rückgängig gemacht werden. Gerade die Rücksetzung von Folgeereignissen kann zu einer langen Kette von Rollbacks führen (sogenannten *Rollbackkaskaden*), die den zeitlichen Gewinn optimistischer Ereignisausführungen wieder reduziert. Leistungsmessungen belegen, daß weder rein konservative noch rein optimistische Verfahren optimal für alle Modelle sind. So gibt es Modelle, die wesentlich schneller mit konservativen Methoden simuliert werden können als mit optimistischen und solche, wo das Umgekehrte zutrifft [LIM90a, PRE90a]. Dies begründet die Suche nach hybriden konservativ–optimistischen Verfahren, die unter Beibehaltung der Vorteile die Defizite zu kompensieren versuchen. Exemplarisch für eine solche Methode wurde die sogenannte *spekulative* Simulationsmethode entwickelt [MEH91a].

Diese Methode geht von einem frei wählbaren konservativen Verfahren aus. Anstatt jedoch während des Wartens auf Garantien (oder auf eine Deadlockbehebung) zu blockieren und den Prozessor untätig sein zu lassen, werden wie in optimistischen Verfahren bereits eingeplante Ereignisse auf einer privaten Kopie des Zustands spekulativ ausgeführt. Die durch eine spekulative Ereignisausführung erzeugten Ereignisse werden lokal gepuffert. Eine spekulative Berechnung verändert daher weder den eigentlichen lokalen Zustand noch die Ereignisliste. Stellt sich später heraus, daß die Spekulation korrekt war, können die private Kopie und die gepufferten Ereignisse für eine schnelle Aktualisierung des Zustands und der Ereignislisten herangezogen werden.

Da bei diesem Verfahren ausschließlich korrekte Informationen übernommen werden, sind keine Rollbacks wie in optimistischen Verfahren nötig. Daher können auch *keine Rollbackkaskaden* auftreten, so daß spekulative Simulation prinzipiell sogar schneller sein kann als optimistische. Mit einer hohen Wahrscheinlichkeit ist spekulative Simulation jedoch auch schneller als die (beliebig gewählte) zugrundeliegende konservative Methode, da spekulative Berechnungen ausschließlich dann gemacht werden, wenn der entsprechende Prozessor bei der konservativen Methode wartet, d.h. *nichts* tut. Der einzige zusätzliche Zeitaufwand gegenüber der zugrundeliegenden konservativen Methode tritt dann auf, wenn schließlich getestet wird, ob die Spekulation korrekt war oder nicht. Für ein spekulativ ausgeführtes Ereignis e wird dieser Test durchgeführt, sobald (vermittelt durch das konservative Verfahren) die Erlaubnis vorliegt, e auf dem aktuellen Zustand auszuführen. Der Test, ob die spekulative Ausführung von e korrekt war, wird also während einer Zeit durchgeführt, während der das zugrundeliegende konservative Verfahren damit beginnen würde, e auszuführen. Bei dem Test wird überprüft, ob die während der spekulativen Ausführung von e gelesenen Zustandsvariablen noch mit dem aktuellen Zustand übereinstimmen. Trifft dies zu, würde eine erneute Ausführung von e auf dem aktuellen Zustand die gleichen Effekte hervorrufen, wie die bereits durchgeführte spekulative Ausführung. In diesem Fall war die Spekulation korrekt und die Ergebnisse werden übernommen; andernfalls werden die Ergebnisse der spekulativen Berechnung gelöscht und e auf dem aktuellen Zustand ausgeführt.

Folgende Varianten dieser Methode wurden in dem DSL–System implementiert.

Variante A

Wie weiter oben angesprochen, entsteht der wesentliche Zusatzaufwand gegenüber der zugrundeliegenden konservativen Methode, wenn der Test durchgeführt wird. Ein Test für ein Ereignis e kann jedoch ganz entfallen, wenn ein LP seit Beginn einer spekulativen Ausführung von e bis zu dem Realzeitpunkt τ, zu dem der Test durchgeführt werden darf, kein Ereignis mit kleinerem Zeitstempel als e eingeplant bekommt hat: In diesem Fall kann sich der lokale Zustand des LPs seit Beginn der spekulativen Berechnung bis zur Zeit τ nicht verändert haben; die Spekulation ist daher korrekt.

Variante B

Ist ein Test nötig, bevor Ergebnisse spekulativer Ereignisausführungen übernommen werden können, so könnte für einige Ereignisse e der zugehörige Test viel länger dauern als eine erneute Ausführung von e. In diesen Fällen lohnt es sicher nicht, den Test durchzuführen. Anstelle dessen ist eine direkte erneute Ausführung von e ohne Berücksichtigung der spekulativen Ereignisausführung sinnvoller. Glücklicherweise kann jedoch der Zeitbedarf für die Ausführung und den Test von e während der Wartezeit des Prozessors (und damit quasi „umsonst“) abgeschätzt werden: Die Dauer des Tests ist proportional zur Anzahl der während der spekulativen Berechnung von e potentiell gelesenen oder aktualisierten Zustandsvariablen; die Dauer der Aktualisierung des Zustandes mit den Ergebnissen einer korrekten Spekulation ist proportional der Anzahl potentiell aktualisierter Variablen und die Dauer der Ausführung von e ist gleich der spekulativen Ausführung von e, falls sich die Spekulation später als korrekt herausstellt. Andernfalls stellt diese Zeit nur eine Näherung dar. Eine gute Heuristik, um zu entscheiden, ob der Test sinnvoll ist, läßt sich damit durch folgende Formel angeben:

$$\alpha\ (Anzahl_{read}\ +\ 2\ \ Anzahl_{write}) < Anzahl_e \Rightarrow Test\ ist\ sinnvoll$$

Dabei bedeuten

- $Anzahl_{read}$: Die Anzahl potentiell gelesener Zustandsvariablen.
- $Anzahl_{write}$: Die Anzahl potentiell aktualisierter Zustandsvariablen.
- $Anzahl_e$: Die Anzahl der Anweisungen, die bei der spekulativen Ausführung von e durchgeführt wurden. Falls die Spekulation korrekt ist, ist diese Zahl proportional der Ausführungszeit von e.
- α: Ein Proportionalitätsfaktor.

Je höher α eingestellt wird, desto weniger spekulative Ereignisse werden die Bedingung erfüllen, die durch die Formel spezifiziert ist. Für diejenigen Spekulationen, die dies jedoch tun, kann sich der Aufwand für den Test besonders lohnen. Daher

wirkt diese Heuristik wie ein Filter, der aus allen spekulativen Berechnungen lohnenswerte herausfiltert. Zusammen mit dieser Heuristik lassen sich auch die beiden folgenden Varianten kombinieren.

Variante C

Anstatt *alle* während einer Spekulation erzeugten Ereignisse zu puffern, könnten zumindest die im gleichen LP einzuplanenden Ereignisse sofort in die lokale Ereignisliste eingefügt werden. Dadurch könnten auch solche „vorläufigen" Ereignisse spekulativ ausgeführt werden.

Variante D

Neben den lokalen Ereignissen könnten auch die bei einer spekulativen Ereignisausführung erzeugten Ereignisse für andere LPs dort als vorläufig eingeplant werden. Diese Time Warp schon sehr ähnliche Variante hat allerdings den Nachteil, daß vorläufige Ereignisse explizit durch Nachrichten bestätigt oder annulliert werden müssen, wodurch sich der Nachrichtenaufwand deutlich erhöht.

Mit empirischen Untersuchungen und einem Vergleich der Varianten an konkreten Modellen ist soeben erst begonnen worden. Gegenwärtig wird vermutet, daß sich spekulative Simulation durch eine Kombination der Varianten A, B und C insbesondere dann als vorteilhaft erweist, wenn der Benutzer hinreichend viel Lookahead–Informationen spezifizieren kann (und daher die zugrundeliegende konservative Methode bereits sehr schnell ist), dennoch längere Wartezeiten auftreten, und Ereignisausführungen i.allg. sehr lange dauern. Mindestens läßt sich jedoch festhalten, daß die Variante A spekulativer Simulation beliebige konservative Verfahren verbessert und damit alleine bereits ein interessantes Ergebnis darstellt.

6 Spezielle Aspekte verteilter Simulation

6.1 Reproduzierbarkeit verteilter Simulationsexperimente

Idealerweise ist ein ereignisgesteuertes Simulationsmodell derart spezifiziert, daß die Reihenfolge, in der Ereignisse stattfinden sollen, eindeutig durch die Eintrittszeiten der Ereignisse definiert ist: Wenn alle Zeitstempel der Ereignisse, die im gleichen LP eintreten sollen, verschieden sind, braucht ein LP seine Ereignisse lediglich chronologisch auszuführen, um Reproduzierbarkeit der Ergebnisse sicherzustellen. In der Praxis sind jedoch die Modelle oft nicht in der Weise eindeutig spezifiziert [AGT91a]. Hinzu kommt, daß durch Verwendung von Pseudo–Zufallszahlengeneratoren bei der Berechnung von Zeitstempeln der Benutzer in der Regel nicht garantieren kann, daß es keine zwei Ereignisse mit dem gleichen Zeitstempel für den gleichen LP geben wird. Daher muß ein LP ein *Tie–Breaking–Schema* enthalten, welches bei Ereignissen mit gleichem Zeitstempel die Reihenfolge ihrer Ausführung bestimmt. Dieses Schema sollte jedoch *deterministisch* in dem Sinne sein, daß alle benutzersichtbaren

Ereignisausführungen bei jeder Wiederholung desselben Simulationsexperiments in der gleichen Reihenfolge stattfinden. Dadurch ist aus Benutzersicht nicht nur das Simulationsergebnis (etwa eine Statistik über das Verhalten des simulierten Systems), sondern auch der Simulationsverlauf reproduzierbar.

In einem sequentiellen Simulator läßt sich ein deterministisches Tie–Breaking–Schema leicht angeben: Von zwei gleichzeitigen Ereignissen wird dasjenige zuerst ausgeführt, welches in Realzeit zuerst generiert wurde. Determinismus ist garantiert, da ein sequentieller Simulator nur durch einen einzigen Prozessor ausgeführt wird. Wie Abb. 6 zeigt, ist dieses Schema jedoch nicht mehr deterministisch, wenn es bei verteilter Simulation auf mehreren Prozessoren eingesetzt wird:

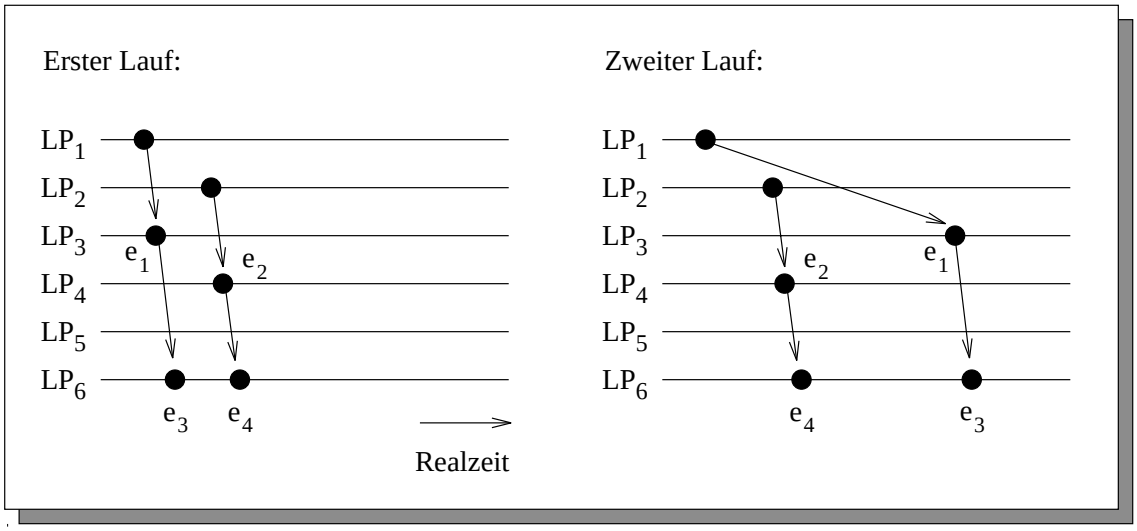

Abbildung 6: Unterschiedliche Nachrichtenlaufzeiten können bei verteilter Simulation der Grund für nicht reproduzierbare Simulationsergebnisse sein.

Sei beispielsweise angenommen, daß e_3 und e_4 in Abb. 6 den gleichen Zeitstempel haben. Aufgrund unterschiedlicher Latenzzeiten bei der Nachrichtenübertragung kann die relative Ausführungsreihenfolge der Ereignisse e_1 und e_2 in Realzeit in zwei aufeinanderfolgenden Simulationsläufen verschieden sein. Da e_3 und e_4 jeweils von e_1 bzw. e_2 erzeugt werden, würde obiges Tie–Breaking–Schema dafür sorgen, daß e_3 und e_4 in beiden Läufen in verschiedener Reihenfolge stattfinden. Durch unterschiedliche Reihenfolgen der Ereignisausführungen kann jedoch auch ein unterschiedliches Simulationsergebnis resultieren.

In der Literatur über verteilte Simulation wurde dieses für die Praxis wichtige Problem bisher kaum behandelt. Von den wenigen vorgestellten Verfahren stellt keines die deterministische Reproduzierbarkeit in allen Fällen sicher (siehe [MEH91c]). Insbesondere garantieren die vorgeschlagenen Tie–Breaking–Schemata meist nicht die Einhaltung der *transitiven Erzeugungsreihenfolge* von Ereignissen. Zur Einhaltung dieser Reihenfolge muß sichergestellt

werden, daß alle direkt oder indirekt von einem Ereignis e_{Vater} erzeugten Ereignisse e_{Sohn}, die im gleichen LP wie e_{Vater} ausgeführt werden sollen, *nach* e_{Vater} auszuführen sind. Diese in sequentieller Simulation trivialerweise erfüllte Forderung (e_{Sohn} existiert erst, nachdem e_{Vater} ausgeführt wurde, und Ereignisausführungen werden nicht wiederholt) ist bei optimistischer verteilter Simulation nicht mehr per se erfüllt. Wie in [MEH91c] gezeigt wird, ist es aufgrund der Verletzung dieser Bedingung in Time Warp möglich, in eine systeminterne Endlosschleife zu gelangen, so daß die Simulationszeit nicht mehr voranschreitet. Daher wurde ein Verfahren entwickelt, welches die Reproduzierbarkeit in gleicher Weise für konservative und optimistische Verfahren sicherstellt [MEH91a].

Die Idee dieses Verfahrens besteht im wesentlichen darin, den Zeitstempel t in einer für den Benutzer transparenten Weise intern zu einem Viertupel (t, a, S, i) zu erweitern und die Reihenfolge der Ereignisausführungen durch einen lexikographischen Vergleich der Viertupel festzulegen. Die erste Komponente des Viertupels entspricht dem vom Benutzer vorgegebenen Zeitstempel. Aufgrund des lexikographischen Vergleichs werden die folgenden drei Komponenten des Viertupels nur dann berücksichtigt, wenn zwei Ereignisse den gleichen Zeitstempel t besitzen. Die zweite Komponente ist das *Alter* a eines Ereignisses. Erzeugt ein Ereignis e_{Vater} mit Zeitstempel t und Alter a ein Ereignis e_{Sohn} mit dem gleichen Zeitstempel t, so erhält das Sohnereignis das Alter $a + 1$ zugeordnet, andernfalls das Alter 1. Zusammen mit dem lexikographischen Vergleich ist so auf einfache Weise sichergestellt, daß die transitive Erzeugungsreihenfolge eingehalten wird. Die zwei weiteren Komponenten S und i stellen in sich eine global eindeutige Kennung dar, so daß sichergestellt ist, daß letztendlich *alle* Ereignisse linear angeordnet werden können. Dabei ist S die Kennung des LPs. Von ihr wird vorausgesetzt, daß sie bei jeder Wiederholung des Simulationslaufs für den gleichen LP identisch ist. Dies ist leicht erfüllbar, wenn die Anzahl der LPs sich während der Simulation nicht mehr verändert, was bei der von uns betrachteten Klasse von verteilten Simulatoren der Fall ist. Die vierte Komponente i stellt schließlich einen Folgezähler dar, der angibt, als wievieltes Ereignis das erzeugte Ereignis von LP S generiert wurde. Das Schema ist so gewählt, daß es eine deterministische Reproduzierbarkeit sicherstellt und damit dieses Problem für die Praxis auf einfache Weise löst. Bei Time Warp mit „Lazy Cancellation“ sind einige Feinheiten zu beachten, die ausführlich in [MEH91c] beschrieben sind.

6.2 Modellierungsunterstützung für inhärent globale Daten

Verteilte Simulation wird auf einem Mehrrechnersystem ohne gemeinsamen Speicher ausgeführt. Daher muß ein verteiltes Simulationsmodell in Teilmodelle mit *disjunkten Zustandsräumen* partitioniert werden, welche jeweils von einem LP ausgeführt werden. Aus Modellierersicht ist diese strikte Disjunktheit jedoch oft unerwünscht. Zwar läßt sich im Prinzip jedes Modell derart partionieren, aber eine Partitionierung inhärent globaler Daten führt dabei üblicherweise zu einer Replikation der Daten und vielen benutzerspezifizierten Ereignissen, um diese Replikate konsistent zu halten [WIJ89a, WHB90a]. Dies ist jedoch nicht nur fehleranfällig, sondern führt auch zu Simulationsmodellen, die nur schwer zu verstehen, zu

validieren und zu warten sind. Aus Anwendungssicht würde man oft gerne gemeinsame Daten zwischen mehreren LPs zur Verfügung haben; in [GHF91a, CCU90a, LUB90a] werden beispielsweise Simulationsanwendungen beschrieben, die sich mit gemeinsamen Variablen räumlich benachbarter Sektoren einer Modellwelt einfach realisieren lassen. Diese Beobachtungen motivierten uns, mögliche Ansätze zu untersuchen, logisch gemeinsamen Speicher konsistent in verteilte Simulation zu integrieren.

Da Lesen und Schreiben gemeinsamer Variablen Aktionen sind, die zu einem bestimmten virtuellen Simulationszeitpunkt auszuführen sind, scheint ein naheliegender Ansatz darin zu bestehen, Lese- und Schreiboperationen als primitive Ereignisse zu betrachten. Bei einer Implementierung von globalen Variablen mittels Replikation könnten „Aktualisierungs–Ereignisse“ allen LPs eingeplant werden, die eine Kopie der gemeinsamen Daten besitzen. Dadurch würden diese Ereignisse in der Ereignisliste eines LPs gepuffert und chronologisch ausgeführt. „Lese-Ereignisse“ würden jedoch Probleme bereiten: Das Ergebnis eines Lese-Ereignisses müßte am Ende der Leseoperation zurückgeliefert werden, also noch *während* der Ausführung des Ereignisses, das die Leseoperationen enthält. Dies widerspricht allerdings der Atomizitätsvoraussetzung ereignisgesteuerter Simulation. Prinzipiell könnte ein Ereignis e in mehrere Ereignisses $e_1, \ldots, e_m$ aufgeteilt werden, so daß Leseoperationen, wenn überhaupt, nur am Ende eines solchen Ereignisses e_i ausgeführt würden. Jedes Ereignis e_i könnte sein Folgeereignis e_{i+1} so einplanen, daß wenn e_i ein Lese-Ereignis generiert, dieses zwischen e_i und e_{i+1} eingeplant werden kann. Ereignisse jedoch derart aufzusplitten ist für einen Benutzer umständlich und führt zu schwer wartbaren Programmen. Eine Unterstützung durch den Compiler beim Zerteilen von Ereignissen ist zwar prinzipiell möglich, allerdings führt dies zu gewissen technischen Problemen.

Ein anderer Ansatz besteht darin, die verteilte Simulation auf einer Emulation eines physisch gemeinsamen Speicher (*Distributed Shared Memory*, kurz *DSM*) ablaufen zu lassen. Algorithmen, die logisch gemeinsamen Speicher auf Mehrrechnersystemen ohne physisch gemeinsamen Speicher realisieren, werden im folgenden *DSM–Algorithmen* genannt. Dieser Ansatz führt jedoch auch auf Probleme, da verteilte Simulation — im Gegensatz zu vielen anderen verteilten Anwendungen — genau vorschreibt, in welcher Reihenfolge Lese- und Schreiboperationen auszuführen sind. Beispielsweise könnte LP_1 eine Variable zur Simulationszeit 100 lesen wollen, welche LP_2 zur Simulationszeit 80 aktualisiert. Durch die asynchrone Ausführung von LPs kann es jedoch sein, daß LP_1 zwar die Simulationszeit 100 erreicht hat und daher die Variable lesen möchte, aber LP_2 erst bei Simulationszeit 50 ist. Der zu lesende Wert ist daher noch gar nicht geschrieben (er wird erst geschrieben, sobald LP_2 die Simulationszeit 80 erreicht).

Unsere Lösung des Problems beruht auf der Beobachtung, daß die Synchronisation, die nötig ist, um logisch gemeinsame Variablen mit virtueller Zeit zu synchronisieren, ähnlich zu der Synchronisation von Ereignissen in verteilter Simulation ist. Das Grundproblem besteht darin zu entscheiden, wann sich ein LP "traut", einen Wert einer gemeinsamen Variable tatsächlich zu lesen oder zu schreiben (sein nächstes Ereignis auszuführen), obwohl andere LPs möglicherweise weitere Aktualisierungen dieser gemeinsamen Variablen (weitere Ereignisse) einplanen,

die noch vorher auszuführen sind. Dies führt auf die Idee, verteilte Simulationsalgorithmen mit DSM–Algorithmen zu kombinieren.

Zu diesem Zweck wurde zunächst eine Studie über bekannte Techniken zur Realisierung von DSM–Algorithmen erstellt [MEH92c]. Es zeigt sich, daß DSM–Algorithmen prinzipiell auf drei verschiedenen Ansätzen beruhen: Entweder werden keine Daten repliziert, oder sie werden nur zum Lesen repliziert, oder Daten werden vollständig bei allen Prozessen repliziert. Desweiteren sind die ersten beiden Ansätze jeweils mit Migration kombinierbar. Aufgrund des fehlenden Lokalitätsverhaltens bei verteilter Simulation scheint Migration jedoch nicht für die Realiserung gemeinsamer Variablen in verteilter Simulation geeignet. Es bleiben also drei Basisalgorithmen zur Realiserung von DSM. Da es auf der anderen Seite i.w. zwei Grundklassen verteilter Simulationsalgorithmen gibt — konservative und optimistische — lassen sich durch Kombination potentiell sechs Grundalgorithmen erhalten, mit denen konsistente gemeinsame Variablen für verteilte Simulation realisiert werden können. Es zeigte sich, daß eine dieser Kombinationen (konservative Simulation und vollständige Replikation) nicht sinnvoll ist. Die verbleibenden fünf Verfahren stellen jedoch vielversprechende neue Algorithmen dar, die das Problem z.T. besser lösen als die in der Literatur bekannten Ansätze. Eine ausführliche Diskussion der Algorithmen, ihrer Eigenschaften und einen Vergleich mit den in der Literatur bekannten Verfahren kann [MEH92e] entnommen werden. Durch die Bereitstellung von gemeinsamen Variablen in verteilter Simulation hat der Modellierer damit einen Mechanismus zur Verfügung, um verteilte Simulationsanwendungen trotz einiger inhärent globaler Daten einfach und effizient zu realisieren.

6.3 GVT – Approximation

Als Global Virtual Time (GVT) bezeichnet man den Wert des Minimums aller logischen Uhren der LPs eines verteilten Simulators sowie aller Zeitstempel von Ereignisnachrichten, die noch nicht verarbeitet wurden. Da die logischen Uhren aller Simulatoren i. allg. verschieden weit fortgeschritten sind, spielt die GVT im verteilten Fall die Rolle der eigentlichen Simulationszeit. Sie wächst im Verlauf der Simulation monoton mit der Ausführungszeit.

Von besonderer Bedeutung ist die GVT bei optimistischen Simulationsverfahren wie TimeWarp. Durch einen Rollback, der durch den Empfang einer Antinachricht oder einer Ereignisnachricht mit einem kleineren Zeitstempel als der Wert der lokalen Uhr ausgelöst wird, kann zwar die logische Uhr des Empfängers zurückgesetzt werden, jedoch niemals auf einen Wert, der kleiner als der momentane Wert der GVT ist. (Lediglich aufgrund technischer Ursachen kann dies dann erforderlich sein, wenn aus Optimierungsgründen nicht genügend Sicherungspunkte vorhanden sind. Dies ist allerdings für die grundsätzlichen Überlegungen ohne Bedeutung.) Daher können alle lokalen Sicherungspunkte bis auf den jüngsten, die älter als die GVT sind, gelöscht werden. Neben dieser Anwendung im Rahmen der Speicherplatzverwaltung ist eine Kenntnis der GVT auch für die Durchführung von Ausgabeoperationen, die nicht mittels Rollback rückgängig gemacht werden können, für das Abbrechen der Simulation bei Erreichen

eines Endezeitpunktes sowie ggf. für Lastausgleichszwecke (z. B. Verlagerung von Last vom "schnellsten" zum "langsamsten" LP) notwendig.

Falls die verteilte Simulation nicht zeitweise angehalten wird, was generell unerwünscht ist, kann der momentane Wert der GVT i. allg. nicht exakt, sondern nur approximativ ermittelt werden. Wünschenswert ist die Berechnung einer möglichst guten unteren Schranke. Hierzu wurden zwei Ansätze verfolgt, die beide einfache, effiziente und (im Sinne der Approximation) gute Algorithmen ergeben. Diese wurden in unsere weiter oben beschriebenen Simulationssysteme implementiert.

Der erste Ansatz beruht darauf, daß eine Lösung für das Problem der Entdeckung der verteilten Terminierung, welches schon vielfach untersucht wurde (vgl. etwa [MAT87a] für eine Diskussion dieses Problems sowie für weitere Literaturhinweise), zu einer Lösung für das GVT-Approximationsproblem erweitert wird. Zwar war schon länger bekannt, daß das Terminierungsproblem als ein GVT-Approximationsproblem mit binärer logischer Zeit aufgefaßt werden kann [JEF85a], interessanterweise läßt sich jedoch auch umgekehrt aus einem synchron ausgeführten Bündel von konzeptuell unendlich vielen Algorithmen zur Lösung des Terminierungsproblems ein GVT-Approximationsverfahren herleiten. Dies wird in [MMS91a] ausführlich beschrieben.

Im zweiten Ansatz wird gewissermaßen umgekehrt vorgegangen, indem Lösungen eines allgemeineren Problems (des sogenannten Schnappschußproblems) zu Lösungen des GVT-Approximationsproblems spezialisiert werden. In [MAT93a] wird dazu zunächst ein einfaches Verfahren vorgestellt, mit dem eine kausal konsistente globale Sicht eines verteilten Ablaufs ermittelt werden kann. Die Idee beruht darauf, Prozesse und Nachrichten mit zwei "Farben" zu kennzeichnen, so daß Nachrichten, die die Kausalität verletzen würden, aufgrund ihrer "Farbe" als solche erkannt werden und entsprechend behandelt werden können. Ferner wird gezeigt, wie ebenfalls durch die Verwendung dieser Farben diejenigen Nachrichten erkannt werden können, die während des ermittelten Schnappschusses unterwegs sind. Da sich die GVT–Approximation – wie in [MAT93a] gezeigt – auch auf kausal inkonsistenten Schnappschüssen durchführen läßt, ergibt sich für die Lösung des GVT–Problems noch eine wesentliche Vereinfachung. Ein Vergleich mit bisher bekannten GVT–Approximationsverfahren zeigt, daß der neue Algorithmus sowohl bzgl. des Speicherplatzbedarfs als auch bzgl. des Nachrichtenaufkommens effizienter ist.

Literatur

[ABL90a] ABRAMS, M. and LOMOW, G. (1990) *Issues in Languages for Parallel Simulation (Panel).* Proceedings of the Distributed Simulation Conference, (San Diego, California, Jan. 17–19), SCS, pp. 227–288

[AGT91a] AGRE, J. and TINKER, P. (1991) *Useful Extensions to a Time Warp Simulation System.* Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation, (Anaheim, California, January 23-25), SCS, pp. 78–85

[APE93a] APEL, M. (1993) *Verteilte Simulation mit konservativen Strategien für DSL.* Projektarbeit, Fachbereich Informatik, Universität Kaiserslautern

[ARM93a] ARMBRECHT, T. (1993) *Entwicklung eines verteilten Simulationsmodells mit DSL.* In Vorbereitung: Projektarbeit, Fachbereich Informatik, Universität Kaiserslautern

[BAS88a] BAIN, W. and SCOTT, D. (1988) *An Algorithm for Time Synchronization in Distributed Discrete Event Simulation.* Proceedings of the Multiconference on Distributed Simulation, (San Diego, CA, February 3–5), SCS, pp. 30–33

[BAS92a] BAUER, H. and SPORRER, C. (1992) *Distributed Logic Simulation and an Approach to Asynchronous GVT–Calculation.* Proceedings of the 6th Workshop on Parallel and Distributed Simulation (PADS92), (Newport Beach, California, January 20-22), SCS, pp. 205-208

[BBL90a] BEMMERL, T., BODE, A., LUDWIG, T. and TRITSCHER, S. (1990) *MMK - Multiprocessor Multitasking Kernel.* SFB-Bericht 342/26/90 A, TU München

[BOC93a] BOCK, S. (1993) *Realisierung von Erweiterungen und Varianten des Time Warp Systems für DSL.* In Vorbereitung: Projektarbeit, Fachbereich Informatik, Universität Kaiserslautern

[BRO88a] BROWN, R. (1988) *Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem.* Communications of the ACM 31:10, pp. 1220-1227

[CCU90a] CONKLIN, D., CLEARY, J., and UNGER, B. (1990) *The Sharks World—A Study in Distributed Simulation Design.* Proceedings of the SCS Multiconference on Distributed Simulation, (San Diego, January 17–19), pp. 157–160

[CHB83a] CHANDAK, A. and BROWNE, J.C. (1983) *Vectorization of Discrete Event Simulation.* Proceedings of the International Conference on Parallel Processing, pp. 359-361

[CHM79b] CHANDY, K. and MISRA, J. (1979) *Distributed Simulation: A Case Study in Design and Verification of Distributed Programs.* IEEE Transactions on Software Engineering SE-5:5, pp. 440-452

[CHM81a] CHANDY, K. and MISRA, J. (1981) *Asynchronous Distributed Simulation Via a Sequence of Parallel Computations.* CACM 24:4, pp. 198-205

[DRE89] DICKENS, P. M. and REYNOLDS, P. F. (1989) *SPECTRUM: A Parallel Simulation Testbed.* Proceedings of the Hypercube Conference

[ELS93a] ELSBERND, K. (1993) *Portierung von CSSA auf ANSI–C und Entwicklung einer Graphikoberfläche für CSSA und DSL.* In Vorbereitung: Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern

[FUJ88b] FUJIMOTO, R. (1988) *Performance Measurements of Distributed Simulation Strategies.* Proceedings of the Distributed Simulation Conference, (San Diego, California, February 3–5), pp. 14-20

[FUJ88c] FUJIMOTO, R. (1988) *Lookahead in Parallel Discrete Event Simulation.* Proceedings of the International Conference on Parallel Processing, pp. 34-41

[FUJ89a] FUJIMOTO, R. (1989) *The Virtual Time Machine.* University of Utah, Salt Lake City, Technical Report UUCS-88-019

[FUJ90a] FUJIMOTO, R. (1990) *Parallel Discrete Event Simulation.* Communications of the ACM 33:10, pp. 30-53

[GHF91a] GHOSH, K. and FUJIMOTO, R. (1991) *Parallel Discrete Event Simulation Using Space–Time Memory.* International Conference on Parallel Processing, pp. III:201–208

[GRT87a] GROSELJ, B. and TROPPER, C. (1987) *Pseudosimulation: An Algorithm for Distributed Simulation with Limited Memory.* International Journal of Parallel Programming 15:5, pp. 413-456

[GRT88a] GROSELJ, B. and TROPPER, C. (1988) *The Time–of–Next–Event Algorithm.* Proceedings of the Multiconference on Distributed Simulation, (San Diego, California, February 3–5), SCS, pp. 25-29

[GRT91a] GROSELJ, B. and TROPPER, C. (1991) *The Distributed Simulation of Clustered Processes.* Distributed Computing, 4, pp. 111-121

[HAM92a] HAMMES, S. (1992) *Distributed Shared Memory und seine Einsatzmöglichkeiten in der verteilten Simulation.* Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern

[HEW77a] HEWITT, C. (1977) *Viewing Control Structures as Patterns of Passing Messages.* Artif. Intell. 8, pp. 323-364

[JEF85a] JEFFERSON, D. (1985) *Virtual Time.* ACM Transactions on Programming Languages and Systems 7:3, pp. 404-425

[JES82a] JEFFERSON, D. and SOWIZRAL, H. (1982) *Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control.* RAND Note N–1906AF, Rand Corp., Santa Monica, CA

[KLI91a] KLINGEL, H. (1991) *Portierung des Laufzeitsystems einer verteilten Programmiersprache auf Transputer.* Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern

[KLU92a] KLUGER, G. (1992) *Konzeption und Implementierung eines verteilten Simulationssystems.* Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern

[LEO93a] LEOPOLD, T. (1993) *Verteilte Straßenverkehrssimulation mit DSL.* In Vorbereitung: Projektarbeit, Fachbereich Informatik, Universität Kaiserslautern

[LIL90d] LIN, Y. and LAZOWSKA, D. (1990) *Determining the Global Virtual Time in a Distributed Simulation.* Technical Report 90-01-02, Department of Computer Science, University of Washington, Seattle

[LIM90a] LIPTON, R. and MIZELL, D. (1990) *Time Warp vs. Chandy–Misra: A worst case Comparison.* Proceedings of the Distributed Simulation Conference, (San Diego, California, Jan. 17–19), SCS, pp. 137–143

[LSW89a] LUBACHEVSKY, B., SHWARTZ, A., and WEISS, A. (1989) *Rollback Sometimes Works ... If Filtered.* Proceedings of the Winter Simulation Conference, pp. 630-639

[LUB90a] LUBACHEVSKY, B. (1990) *Simulating Colliding Rigid Disks in Parallel Using Bounded Lag Without Time Warp.* Proceedings of the SCS Multiconference on Distributed Simulation, (San Diego, California, January 17–19), pp. 194–202

[MAM89a] MATTERN, F. and MEHL, H. (1989) *Diskrete Simulation — Prinzipien und Probleme der Effizienzsteigerung durch Parallelisierung.* Informatik-Spektrum 12:4, pp. 198-210

[MAM89b] MATTERN, F. and MEHL, H. (1989) *Kooperierende Simulatoren und verteilte Simulation.* Fuss H., Thome R. (Eds.): 5. GI–Workshop Simulationsmethoden für verteilte, parallele Prozesse, (Würzburg, Deutschland, 10.–11. April), ASIM

[MAT87a] MATTERN, F. (1987) *Algorithms for Distributed Termination Detection.* Distributed Computing 2, pp. 161-175

[MAT88c] MATTERN, F. (1988) *CSSA – Sprache und Systembenutzung.* Interner Bericht SFB124-24/88, Fachbereich Informatik, Universität Kaiserslautern

[MAT93a] MATTERN, F. (1993) *Efficient Algorithms for Distributed Snapshots and Global Virtual Time.* Journal of Parallel and Distributed Computing 18, pp. 423-434

[MEH89a] MEHL, H. (1989) *Konzeption und Implementierung eines verteilten Simulationssystems und einer Laufzeitumgebung für eine verteilte Programmiersprache.* Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern

[MEH91a] MEHL, H. (1991) *Speed–Up of Conservative Distributed Discrete Event Simulation Methods by Speculative Computing.* Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation, January 23-25, Anaheim, California, Simulation Series 23:1, Eds.: V. Madisetti, D. Nicol, R. Fujimoto, pp. 163-166

[MEH91c] MEHL, H. (1991) *Breaking Ties Deterministically in Distributed Simulation Schemes.* Technical Report 217/91, Department of Computer Science, University of Kaiserslautern, Fed. Rep. of Germany (December)

[MEH92a] MEHL, H. (1992) *A Deterministic Tie–Breaking Scheme for Sequential and Distributed Simulation.* Proceedings of the 6th Workshop on Parallel and Distributed Simulation (PADS92), (Newport Beach, California, January 20-22), SCS, pp. 199-200

[MEH92c] MEHL, H. (1992) *Distributed Shared Memory: A Survey.* Technical Report SFB124–33/92, Department of Computer Science, University of Kaiserslautern, Fed. Rep. of Germany

[MEH92e] MEHL, H. and HAMMES, S. (1992) *How to Integrate Shared Variables in Distributed Simulation.* Technical Report SFB124–32/92, Department of Computer Science, University of Kaiserslautern, Fed. Rep. of Germany

[MEH93b] MEHL, H. (1993) *Die verteilte Simulationssprache DSL (Sprachbeschreibung).* In Vorbereitung: Technischer Bericht, Fachbereich Informatik, Universität Kaiserslautern

[MEH93c] MEHL, H. and HAMMES, S. (1993) *Shared Variables in Distributed Simulation (Invited Paper).* Proceedings of the 7th Workshop on Parallel and Distributed Simulation, (San Diego, CA, May 16–19), SCS

[MEI91a] MEISTER, G. (1991) *Verteilte Simulation von Warteschlangennetzen auf Transputern.* Projektarbeit, Fachbereich Informatik, Universität Kaiserslautern

[MEI92a] MEISTER, G. (1992) *Implementierung der Time-Warp-Strategie für den verteilten Warteschlangennetz-Simulator DISQUE.* Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern

[MER90a] MEISTER, G. and RICHTER, J. (1990) *Distributed Simulation of Queueing Networks on a Network of Transputers.* Proceedings of the PCA Workshop, Bonn

[MHF92a] MADISETTI, V., HARDAKER, D., and FUJIMOTO, R. (1992) *The MIMDIX Operating System for Parallel Simulation.* Proceedings of the 6th Workshop

on Parallel and Distributed Simulation (PADS92), (Newport Beach, California, January 20-22), SCS, pp. 65–74

[MIM84a] MITRA, D. and MITRANI, I.(1986) *Analysis and Optimum Performance of Two Message-Passing Parallel Processors Synchronized by Rollback.* Proceedings of the 10th International Symposium on Computer Performance, pp. 35-50

[MIS86a] MISRA, J. (1986) *Distributed Discrete-Event Simulation.* Computing Surveys 18:1, pp. 39-65

[MMR90a] MEHL, H., MATTERN, F., and RICHTER, J. (1990) *Distributed and Parallel Simulation on a Network of Transputers.* Workshop of the 2nd ESPRIT–Parallel Computing Action, (Ispra, Italy, Dec. 4–7)

[MMS91a] MATTERN, F., MEHL, H., SCHOONE, A., and TEL, G. (1991) *Global Virtual Time Approximation with Distributed Termination Detection Algorithms.* Technical Report RUU–CS–91-32, University of Utrecht

[MWM88a] MADISETTI, V., WALRAND, J., and MESSERSCHMITT, D. (1988) *WOLF: A Rollback Algorithm for Optimistic Distributed Simulation Systems.* Proceedings of the Winter Simulation Conference, pp. 296-305

[NEE87a] NEELAMKAVIL, F. (1987) *Computer Simulation and Modelling.* John Wiley & Sons

[NIR90a] NICOL, D. and RIFFE, S. (1990) *A "Conservative" Approach to Parallelizing the Sharks World Simulation.* Proceedings of the Winter Simulation Conference, (New Orleans, Louisiana, December 9–12), pp. 186–190

[PAG91a] PAGE, B. *Diskrete Simulation—Eine Einführung mit Modula-2.* Springer–Verlag 1991

[PAM88a] PARK, S. and MILLER, K. (1988) *Random Number Generators: Good Ones are Hard to Find.* Communications of the ACM 31:10, pp. 1192-1201

[PML92a] PREISS, B., MACINTYRE, D., and LOUCKS, W. (1992) *On the Trade–off Between Time and Space in Optimistic Parallel Discrete–Event Simulation.* Proceedings of the 6th Workshop on Parallel and Distributed Simulation (PADS92), (Newport Beach, California, January 20-22), SCS, pp. 33–42

[PRE90a] PREISS, B. (1990) *Performance of Discrete Event Simulation on a Multiprocessor Using Optimistic and Conservative Synchronization.* International Conference on Parallel Processing (August 13–17), pp. 218-222

[PRS92a] PRAKASH, A. and SUBRAMANIAN, R. (1992) *An Efficient Optimistic Distributed Simulation Scheme Based on Conditional Knowledge.* Proceedings of the 6th Workshop on Parallel and Distributed Simulation (PADS92), (Newport Beach, California, January 20–22), SCS, pp. 85–94

[RFB90a] REIHER, P., FUJIMOTO, R., BELLENOT, S., and JEFFERSON, D. (1990) *Cancellation Strategies in Optimistic Execution Systems.* Proceedings of the SCS Multiconference on Distributed Simulation 22:1, pp. 112-121

[RIC90a] RICHTER, J. (1990) *Ein mathematisches Modell für ereignisgesteuerte Simulation.* Interner Bericht Nr. 206/90, Fachbereich Informatik, Universität Kaiserslautern

[RIC91a] RICHTER, J. (1991) *DISQUE: Ein verteilter Simulator für Warteschlangennetze auf Transputern.* Reinhard Grebe, Martin Baumann (Ed.): Parallele Datenverarbeitung mit dem Transputer. 3. Transputer-Anwender-Treffen TAT '91, Springer (Informatik aktuell)

[RIW89a] RIGHTER, R. and WALRAND, J. (1989) *Distributed Simulation of Discrete Event Systems.* Proceedings of the IEEE, 77:1, pp. 99-113

[RMM88a] REED, D.A. and MALONY, A. and McCREDIE, B.D. (1989) *Parallel Discrete Event Simulation Using Shared Memory.* IEEE Transactions on Software Engineering 14:4, pp. 541-553

[RWJ89a] REIHER, P., WIELAND, F., and JEFFERSON, D. (1989) *Limitation of Optimism in the Time Warp Operating System.* Proceedings of the Winter Simulation Conference, pp. 765-770

[SAM85a] SAMADI, B. (1985) *Distributed Simulation: Performance and Analysis.* Ph.D. Dissertation, Dept. of Computer Science, University of California at Los Angeles

[SIM93a] SIMON, S. (1993) *Realisierung von gemeinsamen Variablen für die verteilte Simulationssprache DSL.* In Vorbereitung: Projektarbeit, Fachbereich Informatik, Universität Kaiserslautern

[SMI91a] SCHMITZ, W. (1991) *Verteilte Simulation von Warteschlangennetzen.* Projektarbeit, Fachbereich Informatik, Universität Kaiserslautern

[SSH89a] SOKOL, L., STUCKY, B., and HWANG, V. (1989) *MTW: A Control Mechanism for Parallel Discrete Simulation.* International Conference on Parallel Processing, pp. 250-254

[STE92a] STEINMAN, J. (1992) *SPEEDES: A Unified Approach to Parallel Simulation.* Proceedings of the 6th Workshop on Parallel and Distributed Simulation (PADS92), (Newport Beach, California, January 20-22), SCS, pp. 75–84

[STE93a] STEINACKER, S. (1993) *Realisierung von spekulativer verteilter Simulation in mehreren Varianten für DSL.* In Vorbereitung: Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern

[STU93a] STURM, B. (1993) *YES: Ein Werkzeug zur Analyse verteilter Simulationsalgorithmen.* Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern

[SUS89a] SU, W.-K. and SEITZ, C. (1989) *Variants of the Chandy–Misra–Bryant Distributed Discrete-Event Simulation Algorithm.* Proceedings of the Multiconference on Distributed Simulation, (Tampa, Florida, March 28–31), SCS, pp. 38-43

[VIE92a] VIETZ, R. (1992) *Entwicklung eines Compilers für eine verteilte Simulationssprache.* Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern

[WES88a] WEST, D. (1988) *Optimizing Time Warp: Lazy Rollback and Lazy Re–Evaluation.* M.S. thesis, University of Calgary

[WHB90a] WIELAND, F., HAWLEY, L., and BLUME, L. (1990) *An Empirical Study of Data Partitioning and Replication in Parallel Simulation.* Proceedings of the Distributed Memory Computing Conference

[WIJ89a] WIELAND, F. and JEFFERSON, D. (1989) *Case Studies in Serial and Parallel Simulation.* International Conference on Parallel Processing, pp. 255-258

[WOL88a] WOLF, T. (1988) *Realisierung von CSSA in einer verteilten UNIX-Umgebung.* Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern