Abschlußbericht des Projekts DNS (Distributed Net Simulation)

OFFIS
Escherweg 2
D-26121 Oldenburg

Dipl.-Inform. Stefan Schöf
Dipl.-Inform. Ralf Wieting
Prof. Dr. Michael Sonnenschein

Juni 1997

Inhaltsverzeichnis

Kapitel 1

Ziele und Ergebnisse des Projekts DNS

1.1 Ziele

Das Projekt DNS ist im weiteren Sinne in den Bereich „Sprachen und Werkzeuge zur Modellbildung und Simulation“ einzuordnen. Durch rechnerinterpretierbare Modelle realer Systeme wird die Möglichkeit gegeben, Hypothesen über das Verhalten des Systems zu überprüfen und Prognosen über seine Entwicklung unter modifizierten Bedingungen abzugeben.

Die wesentlichen Ziele des Projekts DNS (Distributed Net Simulation) können wie folgt zusammengefaßt werden:

- Entwicklung einer anwendungsfeld-unabhängigen Sprache auf der Grundlage höherer Petrinetze zur konkreten, diskreten Modellierung komplexer Systeme,
- Entwurf und prototypische Realisierung eines optimierten, verteilten Simulators für Modelle der entwickelten Sprache,
- Implementierung einer Modellierungsumgebung mit einem optimierten, sequentiellen Simulator für die entwickelte Sprache,
- Entwicklung und prototypische Implementierung einer hybriden Modellierungssprache auf der Grundlage höherer Petrinetze.

Im Vordergrund stand dabei einerseits die Fragestellung, inwiefern eine komfortable, „real world“ Modellierungssprache auf der Grundlage höhere Petrinetz durch den Ansatz der verteilten Simulation effizient realisiert werden kann. Andererseits sollte die Integrationsmöglichkeit kontinuierlicher Modellierungsmethoden in höhere Petrinetze untersucht werden. DNS ist somit ein Forschungsprojekt der praktischen Informatik mit hohem Entwicklungsanteil.

1.2 Ergebnisse

Im Rahmen des Projektes DNS wurde zunächst mit der Sprache der THORNs (Timed Hierarchical Object Related Nets) eine umfangreiche und komfortable Sprache für diskrete Petrinetzmodelle entworfen. Diese Sprache, die unter anderem zwei unterschiedliche Zeit- und Hierarchiekonzepte integriert, wurde vollständig durch eine grafische Entwicklungsumgebung und einen sequentiellen Simulator auf einer UNIX-Plattform und später auch unter Windows-NT realisiert. Die Entwicklungsumgebung wurde im Rahmen von Lizenzabkommen an derzeit 11 Anwender an anderen Universitäten und Forschungsinstituten weitergegeben. So entstanden neben eigenen Fallstudien zur Anwendung der Sprache bereits umfangreiche Anwendungen etwa in den Bereichen Produktionsplanung und Verkehrswesen.

Mit der verteilten Implementierung von THOR-Netzen entstand ein verteilter Simulator für eine der umfangreichsten Sprachen, die bisher durch diesen Ansatz realisiert wurden. Durch die Implementierung und Optimierung dieses sehr komplexen Softwareprodukts konnten wichtige neue Erkenntnisse über den Ansatz verteilter, ereignisdiskreter Simulation gewonnen werden.

In der im Rahmen des Projekts entwickelten Sprache der HYNETZE können erstmals auch komplexe Systeme mit diskreten und kontinuierlichen Vorgängen in höheren Petrinetzen dargestellt werden, was diesem Modellierungsansatz weitere Anwendungsmöglichkeiten eröffnet. Die Sprache der HYNETZE wurde prototypisch auf einer UNIX-Plattform durch einen Interpreter mit graphischer Oberfläche implementiert und mit Fallstudien aus verschiedenen Anwendungsgebieten validiert.

Mit dem Projekt DNS entstand so in OFFIS wichtiges Know-how auf dem Gebiet diskreter und hybrider Modellierungssprachen und deren Implementierung. Diese Erfahrungen trugen u. a. zur erfolgreichen Akquisition des Drittmittelprojekts WESP bei, in dem eine Modellierungs-, Experiment- und Auswertungsumgebung für individuen-orientierte Metapopulationsmodelle in Zusammenarbeit mit dem Umweltforschungszentrum Leipzig-Halle entwickelt wird.

Insgesamt kann DNS als ein erfolgreich abgeschlossenes Vorlauf-Projekt angesehen werden, in dem – wie es dem Profil von OFFIS entspricht – auf der einen Seite Beiträge zur Forschung geleistet wurden. Dies ist dokumentiert durch 17 Veröffentlichungen, acht technische Berichte, die Durchführung von zwei Workshops und voraussichtlich zwei Dissertationen. Auf der anderen Seite entstanden komplexe Informatik-Werkzeuge im Umfang von ca. 200.000 LOC, die mit Partnern exemplarisch angewandt wurden.

1.3 Projektdauer und Förderung

Das Projekt DNS wurde 1992 gemeinsam von den Professoren Claus und Sonnenschein beantragt. Nach einer zunächst genehmigten Laufzeit von zwei Jahren und einer zwischenzeitlichen Berufung von Herrn Claus an die Universität Stuttgart wurde 1994 der Verlängerungsantrag durch Herrn Sonnenschein alleine gestellt. Die Gesamtlaufzeit des Projekts betrug vier Jahre: 1.1.1993 bis 31.12.1996.

DNS war bei OFFIS zunächst mit einer Mitarbeiterstelle, ab dem 1.7.1994 mit einer zweiten Mitarbeiterstelle, ferner Mitteln für studentische Hilfskräfte, Sachmitteln und Investitionsmitteln für die erforderliche Rechnerausstattung ausgestattet. Weitere Arbeiten zum Projekt wurden an der Universität Oldenburg im Rahmen der Arbeitsgruppe Informatik-Systeme durchgeführt, aus deren Mitteln ebenfalls eine Förderung des Projekts DNS erfolgte.

1.4 Personelle Ausstattung

Leitung:

- Prof. Dr. M. Sonnenschein, Prof. Dr. V. Claus (bis 31.12.1994)

Wissenschaftliche Mitarbeiter:

- Dipl.-Inform. Karin Rössig (1.4.1993 - 31.7.1993)
- Dipl.-Inform. Stefan Schöf (1.5.1993 - 31.12.1996)
- Dipl.-Inform. Ralf Wieting (1.7.1994 - 31.12.1996)

Studentische Hilfskräfte (mit unterschiedlichen Vertragslaufzeiten):

- Holger Bohle
- Frank Köster
- Roland Majchszak
- Stephan Maurer
- Björn Mielenhausen
- Roland Radtke
- Gerriet Reents
- Lutz Twele
- Werner Ziegler

Im Rahmen des Projekts DNS wurden ferner 10 Diplomarbeiten und sieben Studienarbeiten angefertigt sowie eine Projektgruppe durchgeführt, an der sich 10 Studierende der Universität Oldenburg über ein Jahr beteiligten.

Kapitel 2

Inhaltliche Schwerpunkte

2.1 Die Modellierungssprache der THOR-Netze

Höhere Petrinetze haben als Mittel zur Modellbildung von verteilten Systemen mit diskreter Ereignisstruktur, wie etwa Verkehrssystemen, Fertigungsanlagen, Steuerungssystemen oder Kommunikationsprotokollen, in den letzten Jahren stark wachsendes Interesse gefunden (siehe [JR91] für eine aktuelle Einführung mit Anwendungsbeispielen).

Im Rahmen des Projekts DNS wurde zur Modellierung solcher Systeme eine spezielle Klasse höherer Petrinetze entwickelt. Die sogenannten **T**imed **H**ierarchical **O**bject-**R**elated **N**ets – kurz THOR-Netze – sind höhere Petrinetze, in denen verschiedene bekannte Konzepte integriert wurden, die eine kompakte und effiziente Modellbildung komplexer Systeme erlauben:

- Statt attributloser Marken können komplexe Objekte im Sinne einer objektorientierten Programmiersprache verwendet werden. Da in dem Projekt DNS die Programmiersprache C++ verwendet wird, werden Objekttypen als C++-Klassen definiert.
- Stellen kann die Struktur *Multimenge, Stack, Queue* oder *Priority Queue* zugeordnet werden. Die Objekte werden dann entsprechend dem bekannten Zugriffsverhalten dieser Strukturen auf den jeweiligen Stellen verwaltet. Jede Stelle kann Objekte eines Typs beinhalten. Des weiteren kann sie durch eine Kapazität beschränkt werden.
- Neben Standardkanten gibt es zwischen Stellen und Transitionen *aktivierende, inhibitorische* und *konsumierende* Kanten. Diese Kanten beeinflussen die Aktivierung und das Schaltverhalten der inzidenten Transition.
- Transitionen erhalten eine *Schaltbedingung* und eine *Schaltaktion*, die in C++ formuliert werden. Ferner kann einer Transition eine *Schaltkapazität* zugeordnet werden, die angibt, wie oft die Transition parallel zu sich selbst schalten kann.
- Zur Beschreibung von zeitlichen Abläufen können für Transitionen zwei spezifische Funktionen zur Bestimmung einer *Verzögerungszeit* und einer *Schaltdauer* definiert werden. Die Verzögerungszeit gibt dabei an, wie lange eine Transition ununterbrochen aktiviert sein muß, um schalten zu dürfen. Durch die *Schaltdauer* wird fest-

gelegt, wie lange das Schalten einer Transition dauert. Beide Zeiten können sowohl deterministisch als auch stochastisch spezifiziert werden.

- Die Strukturierung der Netzmodelle kann wahlweise durch eine *Transitionenverfeinerung* im klassischen Sinne (Makroexpansion) oder durch den *Unternetzaufruf* einer Transition im Sinne eines Prozeduraufrufs einer imperativen Programmiersprache durchgeführt werden.

In Kapitel 2 werden die einzelnen Konzepte der Modellierungssprache informell beschrieben.

2.2 Werkzeuge für THOR-Netze

Zur Erstellung, Bearbeitung und Auswertung von THOR-Netz-Modellen wurden in dem Projekt DNS ein interaktives graphisches Netzeditorsystem, ein Netzcompiler und effiziente Simulatoren für eine sequentielle und verteilte Ausführung der Netze entwickelt und prototypisch implementiert. Alle Werkzeuge sind in C++ implementiert und können als autonome Prozesse auf demselben oder verschiedenen Rechnern ablaufen.

Das grundlegende Konzept für eine effiziente Ausführung der Netze ist deren Übersetzung in ausführbare Simulationsprogramme. Nachdem ein Netzmodell mit Hilfe des graphischen Netzeditors eingegeben wurde, kann durch Verwendung des Netzcompilers wahlweise Code für den sequentiellen oder den verteilten Simulator erzeugt werden. Der generierte C++-Code wird anschließend von einen C++-Compiler mit einem Simulatorkern zu einem ausführbaren Programm gebunden. Je nach Art der gewünschten Simulationsform entsteht so ein sequentieller oder verteilter Simulator.

Die Simulation eines Netzes wird direkt aus der Editorumgebung heraus gestartet. Dabei werden dem jeweiligen Simulator einige Simulationsoptionen (max. Schrittzahl, max. Simulationszeit, etc.) und der Anfangszustand (Markierung) übergeben. Nach der Ausführung eines Netzes wird der erreichte Endzustand wieder über eine Datei an den Editor zurückgegeben. Die Benutzerinteraktion findet während des gesamten Prozesses ausschließlich über den graphischen Editor statt.

Eine detaillierte Beschreibung der Systemarchitektur, der graphischen Oberfläche und des sequentiellen Simulators wird in Kapitel 4 angegeben. Weitere einführende Informationen zu den Systemteilen befinden sich in [SSW95a, SSW94a, SSW94b]. Eine ausführliche Darstellung der Konzepte zur sequentiellen und verteilten Simulation ist in [SWK+94] angegeben und die Bedienung der Werkzeuge wird im Handbuch zur THORN-Entwicklungsumgebung [Wie96a] beschrieben.

Ergänzend zum Netzeditor für X-Windows-basierte Systeme wurde ein graphischer Editor für Windows-NT entwickelt [Mie97], um das System auch für PC-Anwender zugänglich zu machen. Dieser Editor stellt insbesondere eine Verbindung über OLE zu anderen Softwarekomponenten des PC zur Verfügung.

2.3 Verteilte Simulation von THOR-Netzen

Bei den im Rahmen des Projektes DNS durchgeführten verteilten Simulationen von THOR-Netzen wurden vor allem optimistische Verfahren zur Synchronisation der einzelnen Subsimulatoren angewendet. Hierbei wurden auch allgemeine Untersuchungen zur verteilten Simulation, die nicht spezifisch für die THORN-Simulation sind, durchgeführt. Diese betrafen zum einen Untersuchungen zur Effizienz der Algorithmen zur Ermittlung der globalen Simulationszeit (GVT), die zum Eindämmen des sehr hohen Speicherverbrauches optimistischer Verfahren wichtig sind. Zum anderen wurden effiziente Datenstrukturen zur Verwaltung der Simulationsereignisse entwickelt und implementiert.

Zur Bewertung von GVT-Approximationsalgorithmen wurde in einer Diplomarbeit ein Testprogramm entwickelt, bei dem verschiedene Parameter der zugrundeliegenden Simulationsanwendung einstellbar sind. Mit Hilfe dieses Testprogramms wurden verschiedene Algorithmen untersucht, die sich aufgrund ihrer theoretischen Eigenschaften als besonders geeignet darstellten. Die hierdurch empirisch ermittelten Ergebnisse bildeten dann die Grundlage für die Auswahl des GVT-Approximationsalgorithmus, der in den optimistischen THORN-Simulatoren implementiert wurde.

Die Notwendigkeit zur Entwicklung einer effizienten Datenstruktur für die Ereignisliste ergab sich aus der Tatsache, daß bei der THORN-Simulation prinzipiell eine große Anzahl von Ereignissen anfallen kann und daß für optimistische Simulationsverfahren i. allg. nur doppelt verkettete lineare Listen verwendet werden, die für größere Ereigniszahlen ineffizient sind. Die neuartige Datenstruktur basiert auf einer ähnlichen Datenstruktur zur Verwaltung der Ereignisliste bei der sequentiellen Simulation. Sowohl theoretische Analysen als auch umfangreiche praktische Erprobungen ergaben ein sehr gutes Verhalten dieser Datenstruktur bei größeren Ereignislisten. Ihre Anwendung ist keineswegs auf die THORN-Simulation beschränkt, sondern für beliebige optimistische Simulationsanwendungen einsetzbar, bei der eine große Anzahl von Ereignissen anfällt.

Bei der verteilten THORN-Simulation wurden verschiedene Varianten optimistischer Synchronisationsverfahren sowie ein konservatives Verfahren angewendet. Es fanden für verschiedene Modelle, u. a. eine Logiksimulationsanwendung und ein Optimierungsproblem umfangreiche Laufzeitmessungen statt. Selbst auf einem Netzwerk von Arbeitsplatzrechnern, die über eine relativ langsame Ethernet-Verbindung kommunizierten, konnte eine deutliche Beschleunigung durch die Hinzunahme von Prozessoren beobachtet werden.

2.4 Beobachtung und Steuerung der verteilten Simulation

Um zu untersuchen, inwieweit es möglich ist, auch bei der verteilten Simulation von THORNs das Auftreten bestimmter Bedingungen zu überprüfen und zu protokollieren, wurde ein Beobachtungsalgorithmus für ein verteiltes Simulationssystem entwickelt und implementiert. Es können hier boolesche Funktionen formuliert werden, die mit Hilfe prädikaten- und temporallogischer Quantoren Bedingungen an die Zustandsvariablen

des Modells stellen. Die Implementierung zeigte, daß eine Beobachtung globaler Modellzustände auch bei der verteilten Simulation prinzipiell möglich ist.

Das Verfahren wurde in einem speziellen Simulator, d. h. nicht im verteilten THORN-Simulator implementiert, da ein direktes Integrieren der Beobachtungsverfahren in den THORN-Simulator wegen dessen Komplexität wesentlich aufwendiger geworden wäre. Im Rahmen einer Lehrveranstaltung wurde allerdings für THORNs eine Beobachtungssprache entwickelt, mit Hilfe derer bestimmte Bedingungen über das Netzmodellverhalten (Stellenbelegungen, Transitionsaktivierungen etc.) formuliert werden können und auf die dann in geeigneter, definierbarer Weise reagiert wird. Eine Integration dieser Beobachtungssprache und der entwickelten verteilten Beobachtungsalgorithmen fand aus Zeitgründen nicht mehr statt, stellt aber keine prinzipielle Schwierigkeit dar.

2.5 Hybride höhere Netze

THOR-Netze haben ihre Eignung zur Modellierung einer Vielzahl von realen Systemen, wie zum Beispiel von industriellen Anlagen, Kommunikationsprotokollen, Büros oder Rechnerarchitekturen in einer Reihe von Fallstudien gezeigt. Allerdings hat sich bei der Erstellung der Fallstudien auch herausgestellt, daß eine Modellbildung nur dann unproblematisch möglich ist, wenn das zu modellierende System in natürlicher Weise durch diskrete Zustandsübergänge beschrieben werden kann. Sollen in dem Modell auch kontinuierliche Zustandstransformationen berücksichtigt werden, so können die entsprechenden kontinuierlichen Variablen beispielsweise nur durch eine Diskretisierung in kleine Einzelteile repräsentiert werden. Dies führt dann allerdings zu einer großen Menge von Zwischenzuständen, von denen die Mehrzahl irrelevant ist. Außerdem werden dadurch an einigen Stellen „unnatürliche“ Hilfskonstruktionen erforderlich.

Wenn also Petrinetze auch zur Modellbildung von Sytemen verwendet werden sollen, in denen neben diskreten auch kontinuierliche Zustandsübergänge wesentlich sind, so müssen entsprechende Beschreibungsmittel integriert werden. Beispiele für solche „hybriden“ Systeme sind Anlagen der chemischen Industrie und ökologische und biologische Systeme.

Um die Modellbildung hybrider Systeme mit Petrinetzen zu verbessern, wurde in der zweiten Projektphase auf der Grundlage von THOR-Netzen die Modellierungssprache der hybriden höheren Netze – kurz HYNETZE – definiert. Dazu wurde die bestehende Netzklasse um kontinuierliche Beschreibungsmittel erweitert. Mit Hilfe dieser Erweiterungen können nun hybride Systeme wie beispielsweise Anlagen der chemischen Industrie oder biologische Systeme auf einfache und kompakte Art und Weise modelliert werden.

Zur Erstellung und Untersuchung der Netzmodelle wurden ein graphischer Editor und ein Simulator entwickelt und prototypisch implementiert.

Die neue hybride Netzklasse und die dafür entwickelten Werkzeuge werden in Kapitel 8 näher beschrieben. Eine ausführliche Beschreibung von HYNETZEN, ihrer Simulation und ihren Anwendungsmöglichkeiten wird in [Wie97] angegeben.

2.6 Erweiterte Simulationsumgebung

Im Rahmen einer Lehrveranstaltung (Projektgruppe) wurden komfortable Schnittstellen zu den im Projekt DNS entwickelten Werkzeugen entworfen und implementiert. Hierbei lagen die Schwerpunkte auf der Steuerung der Simulation und der Visualisierung von Simulationsläufen.

Die Steuerung der Simulation kann dabei in gewissem Maße interaktiv stattfinden; das Setzen von Haltebedingungen sowie die Einzelschrittsimulation sind möglich. Durch die Verknüpfung gewisser graphischer Befehle mit einzelnen Netzzuständen können einerseits visualisierte Statistiken über den Simulationslauf ausgegeben werden, andererseits kann der Simulationslauf auch regelrecht animiert werden.

Zur Visualisierung sind dabei drei verschiedene Diagrammtypen (Balken-, Kurven- und Ganttdiagramm) vorgesehen; diese können jeweils aktuelle oder auch über einen Zeitraum akkumulierte Informationen über das gesamte Netzmodell oder einzelne Teile wiedergeben. Bei der Animation können in einer einfachen Graphiksprache Zeichenbefehle mit bestimmten Vorgängen im Netz verknüpft werden; so kann etwa erreicht werden, daß bei einer Verkehrssimulation das im THORN modellierte Verhalten einer Ampelschaltung durch die Darstellung der entsprechenden Ampel mit ihren Zuständen verdeutlicht wird.

Weitere im Rahmen der Projektgruppe vorgenomme Verbesserungen betrafen die Bereitstellung eines Klasseneditors für die bei THORNs notwendigen Beschreibungen von Token und die Möglichkeit, den Zustand des Editorsystems, d. h. die Größe und Anzahl geöffneter Fenster mit den zugehörigen Unternetzen, abspeichern und wieder laden zu können.

Kapitel 3

Die Modellierungssprache der THOR-Netze

THOR-Netze wurden in erster Linie mit dem Ziel entwickelt, eine höhere Modellierungssprache zu schaffen, die es erlaubt, auch komplexe diskrete Systeme möglichst einfach, anschaulich und kompakt beschreiben zu können. Dazu wurden verschiedene bekannte Konzepte höherer Petrinetze [JR91] miteinander kombiniert. Die Auswahl und Kombination der verwendeten Konzepte wurde durch die Erstellung von Fallstudien motiviert. Insbesondere führten einige Schwierigkeiten bei der Modellierung der Fallbeispiele zu kleineren Erweiterungen und Verbesserungen der Netzdefinition (vgl. Abschnitt 3.5).

Die hier angegebene Beschreibung der Modellierungssprache beruht auf der aktuellen Definition von THOR-Netzen. Ähnliche informelle Beschreibungen sind in unterschiedlichem Umfang in [SSW95a, SSW95b, Wie96a] wiedergegeben. Eine ausführliche formale Definition der ursprünglichen Version der THORNs ist in [FLSW93b] zu finden, während die erste informelle Beschreibung zu THOR-Netzen in [FLSW93a] veröffentlicht wurde.

Im folgenden werden die wesentlichen Konzepte der THOR-Netze präsentiert und mit den in der Literatur bekannten Konzepte für höhere Petrinetze verglichen bzw. aus ihnen abgeleitet. Einige spezielle Details der THOR-Netz-Definition werden hier nicht beschrieben. Für eine vollständige Beschreibung der THOR-Netze wird auf das Handbuch [Wie96a] verwiesen.

3.1 High-level Konzepte

Unter dem Oberbegriff high-level Konzepte sind einige Eigenschaften von THOR-Netzen wie individuell definierbare Objekte, zusätzliche Netzelemente und die Beschriftung der Netzelemente mit verschiedenen Attributen zusammengefaßt.

Zur Definition der Objekttypen und zur Beschriftung der Netzelemente wird die allgemein bekannte objektorientierte Programmiersprache C++ [Str92] verwendet. Prinzipiell könnten auch andere (objektorientierte) Sprachen verwendet werden. Allerdings bot sich C++ an, da diese Sprache auch als Implementierungssprache für die Werkzeuge in dem Projekt DNS eingesetzt wurde.

3.1.1 Individuelle Objekte

Eine klare Verbesserung bzgl. einer höheren Ausdrucksmächtigkeit und einer kompakteren Darstellung gegenüber einfachen Petrinetzklassen [Rei86] wird bei höheren Petrinetzen dadurch erzielt, daß anstatt einfacher Marken individuelle Objekte bzw. Objekttypen mit verschiedenen Attributen definiert werden können.

Zur Spezifikation der Objekttypen werden dabei unterschiedliche Formalismen verwendet. Bei Coloured Petri Nets (CP-Netze) [Jen92] werden beispielsweise Objekttypen – Farben genannt – im Sinne einer funktionalen Programmiersprache definiert. Demgegenüber werden Objekttypen bei Prädikat/Transitionen-Netzen (PrT-Netze) [Gen87] oder algebraischen Netzen [Rei91a] als Sorten im Sinne einer algebraischen Spezifikation definiert. Ein zusammenhängender Überblick über einen Teil der hier zitierten Konzepte ist in [JR91] angegeben. Einige neuere Arbeiten verwenden zur Spezifikation von Objekttypen objektorientierte Konzepte in unterschiedlicher Form und Umfang. Häufig werden dabei Objekttypen als Klassen verstanden, die entweder durch eigene Subnetze repräsentiert werden (vgl. [Lak95, BDM96, SB94, Val96]), oder direkt als Klasse einer objektorientierten Programmiersprache [CT93] definiert werden. Letzteres führt im Gegensatz zu den vorher genannten Ansätzen zu kompakten und intuitiv verständlichen Modellen vorausgesetzt der Leser ist mit der Syntax und Semantik der objektorientierten Beschriftungssprache vertraut.

In THOR-Netzen werden Objekte bzw. die entsprechenden Objekttypen mit Hilfe der weit verbreiteten objektorientierten Programmiersprache C++ spezifiziert. *Objekttypen* werden hier in Form von C++-Klassen definiert und Marken sind Instanzen dieser Klassen. Sie werden entsprechend der objektorientierten Terminologie als *Objekte* bezeichnet.

Theoretisch können bei der Spezifikation der Objekttypen alle objektorientierten Konzepte wie Datenabstraktion, Kapselung, Vererbung, Polymorphismus und dynamisches Binden verwenden werden, die C++ bietet. Praktisch werden jedoch Vererbung und Polymorphismus nur eingeschränkt bzw. gar nicht unterstützt. Diese Entscheidung ist durch Schwierigkeiten in der Implementierbarkeit begründet, die unter anderem auf dem fehlenden Laufzeit-Typinformationssystem von C++ beruhen.

Aber auch ohne Polymorphismus und Vererbung kann ein Objekttyp sehr individuell mit frei wählbaren Attributen und beliebig komplexen Methoden versehen werden. Ein Objekttyp kann aber auch direkt einer vordefinierten Klasse entsprechen. Dazu gehören neben den elementaren C++-Typen `Int`, `Long`, `Float`, `Double` und `Char`, ein Typ für Zeichenketten `String`, ein Typ für boolesche Werte `Bool` und der Petrinetz-spezifische Typ `Token`. Objekttypen werden entsprechend dieser Unterteilung in vordefinierte *elementare* Typen und benutzerdefinierte *komplexe* Typen unterschieden.

Konkret erfolgt die Spezifikation von komplexen Objekttypen wie für C++ üblich durch eine Deklaration und eine Definition der entsprechenden Klassen. Vom Modellierer ist dabei zu beachten, daß für Objekte selbstdefinierter Typen eine externe Repräsentation nicht automatisch definiert ist. Zur Eingabe und Darstellung solcher Objekte im Netzeditor und zur Ein- und Ausgabe durch die Simulatoren muß der Modellierer deshalb für jeden komplexen Objekttyp eine Lade- und Speicherroutine definieren, durch die die externe Repräsentation eines Objekts festgelegt wird. Technisch wird dies dadurch realisiert, daß

jeder komplexe Objekttyp von der Klasse `IOABLE` abgeleitet und die virtuellen Methoden `Load` und `Store` redefiniert werden. Für die elementaren Objekttypen ist das nicht nötig. Weitere Einzelheiten zur Spezifikation von Objekten und Objekttypen befinden sich im Handbuch zur THORN-Entwicklungsumgebung [Wie96a].

3.1.2 Stellen

Stellen werden außer mit einem *Namen*, mit einem *Typ* und einer *Kapazität* beschriftet. Durch den Typ der Stelle wird festgelegt, welchen Typ Objekte haben dürfen, die auf einer Stelle abgelegt werden. Dies kann entweder ein elementarer oder ein selbstdefinierter komplexer Objekttyp sein. Durch die Kapazität der Stelle wird eine maximale Anzahl von Objekten definiert, die sich zu einem Zeitpunkt auf ihr befinden dürfen. Damit lassen sich beispielsweise beschränkte Lagerkapazitäten modellieren. Die Kapazität kann aber auch den unbeschränkten Wert `Omega` annehmen.

Zusätzlich zu diesen Beschriftungen muß eine Stelle mit einer *Struktur* versehen werden, durch die die Reihenfolge der Objekte auf einer Stelle definiert wird. THOR-Netze bieten vier verschiedene Stellenstrukturen an, die graphisch wie in Abbildung 3.1 dargestellt werden.

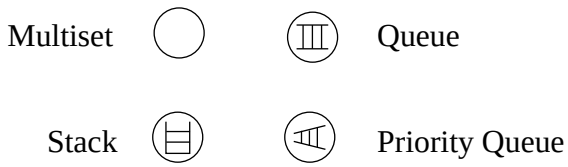


Abbildung 3.1: Darstellung von THOR-Netz-Stellen

- **Multiset-Stellen** haben keine besondere Struktur. Objekte können beliebig hinzugefügt und abgezogen werden. Im Gegensatz zu anderen höheren Petrinetzen wie z. B. CP-Netzen [Jen92] werden die Objekte jedoch nicht in Form von Multimengen verwaltet, sondern als Mengen. Auch Objekte mit gleichen Attributen werden dementsprechend nicht durch ihre Vielfachheit sondern jedes für sich repräsentiert, da sie durch Objektreferenzen eindeutig bestimmt sind.
- **Stack-Stellen** verwalten ihre Objekte entsprechend dem abstrakten Datentyp eines Kellers, d. h. nach dem LIFO-Prinzip. Es wird immer das zuletzt hinzugefügte Objekt abgezogen.
- **Queue-Stellen** verwalten ihre Objekte entsprechend dem abstrakten Datentyp einer Schlange (FIFO-Prinzip). Es wird immer das Objekt abgezogen, daß sich am längsten auf der Stelle befindet.
- **Priority-Queue-Stellen** ordnen die Objekte entsprechend einer vom Modellierer zu spezifizierenden Prioritätsfunktion an. Es wird immer das Objekt mit der höchsten Priorität abgezogen.

Erstmals wurden Stellenstrukturen in dieser Form von der Projektgruppe Nessi [CFL+89] eingeführt, die im Rahmen des MOBY-Projekts [FL93] an der Universität Oldenburg veranstaltet wurde. Insbesondere die Struktur der Queue-Stellen hat sich neben den klassischen Multiset-Stellen in diversen Fallstudien als sehr praktisch erwiesen.

3.1.3 Kanten

Kanten haben einen *Namen* und einen *Kantentyp*. Neben Standard-Kanten gibt es in THOR-Netzen drei spezielle Kantentypen, mit denen die Aktiviertheit und das Schaltverhalten der inzidenten Transition beeinflußt werden kann. Die speziellen Kanten sind nur von Stellen zu Transitionen zugelassen. Für einige Kantentypen kann zusätzlich ein *Kantengewicht* und ein *Variablenname* angegeben werden. Die graphische Repräsentation der verschiedenen Kanten ist in Abbildung 3.2 dargestellt.

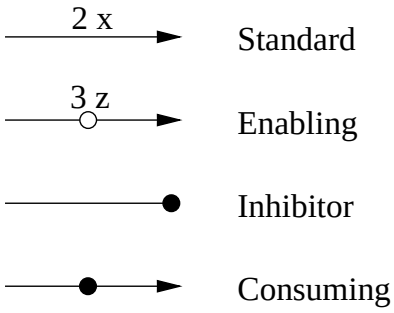


Abbildung 3.2: Darstellung von THOR-Netz-Kanten

- **Standard-Kanten** entsprechen den herkömmlichen Kanten in Petrinetzen. Sie werden mit einem *Gewicht* und einem *Variablennamen* beschriftet. Durch das Gewicht einer Standard-Kante von einer Stelle zu einer Transition wird definiert, wieviele Objekte auf der Stelle liegen müssen, um die Transition zu aktivieren bzw. wieviele Objekte von der Stelle abgezogen werden, wenn die Transition schaltet. Entsprechend wird durch das Gewicht einer Standard-Kante von einer Transition zu einer Stelle festgelegt, wieviel Platz auf der Stelle zur Verfügung stehen muß, um die Transition zu aktivieren bzw. wieviele Objekte auf der Stelle beim Schalten der Transition produziert werden. Mit Hilfe des Variablennamens können die Objekte einer Stelle in der Transition referenziert werden (s. u.). Aus historischen Gründen (vgl. [FLSW93b]) und in Ermangelung eines konkreten Bedarfs dürfen Kanten, die zu Stellen mit einer besonderen Struktur (Stacks, Queues und Priority Queues) inzident sind, nur das Gewicht eins haben.
- **Enabling-Kanten** (auch aktivierende Kanten genannt) werden genauso beschriftet wie Standard-Kanten. Sie haben denselben Einfluß auf die Aktiviertheit einer Transition wie Standard-Kanten, allerdings werden beim Schalten der Transition keine Objekte über diese Kanten konsumiert. Die Objekte dürfen beim Schalten der Transition nicht verändert werden. Manchmal wird dieser Kantentyp daher auch als Lese- (read arc) oder Testkante (test arc) bezeichnet.

- **Inhibitor-Kanten** (inhibitorische Kanten) deaktivieren eine Transition, wenn sich Objekte auf der inzidenten Stelle befinden. Das Schalten einer Transition hat keinen Einfluß auf die adjazente Stelle.
- **Consuming-Kanten** (konsumierende Kanten) haben keine Auswirkungen auf die Aktiviertheit einer Transition, beim Schalten der Transition werden jedoch sämtliche Objekte von der adjazenten Stelle gelöscht.

Inhibitorische und aktivierende Kantentypen sind in ähnlicher Form in der Literatur weitläufig bekannt. Für CP-Netze wurden sie bspw. in [CH93] eingeführt. Im Gegensatz dazu werden konsumierende Kanten wesentlich seltener erwähnt. Außer bei Nessi-Netzen [CFL+89] und den daraus abgeleiteten Netzklassen werden sie nur von Baumgarten [Bau90] unter der Bezeichnung Abräumkanten eingeführt.

3.1.4 Transitionen

Transitionen werden in Thor-Netzen dazu benutzt, durch ihr Schalten Objekte von Stellen zu konsumieren, zu modifizieren und zu produzieren. Sie werden dazu ähnlich wie bei PrT-Netzen [Ram93] mit einer *Schaltbedingung* und einer *Schaltaktion* beschriftet. Durch die Schaltbedingung können zusätzliche Bedingungen an die zu konsumierenden Objekte im Vorbereich gestellt werden und mit der Schaltaktion wird das Verhalten der Transition beim Schalten festgelegt. Zusätzlich können Transitionen mit einem *Namen* und einer *Schaltkapazität* versehen werden. Durch die Schaltkapazität wird festgelegt, wie oft eine Transition parallel zu sich selbst schalten kann. Mit Hilfe dieser Beschriftung lassen sich sehr elegant mehrere Transitionen mit identischem Verhalten zu einer Transition zusammenfassen. Besonders nützlich kann in manchen Fällen die ebenfalls erlaubte unbeschränkte Schaltkapazität `Omega` sein.

Die Spezifikation der Schaltbedingung und Schaltaktion erfolgt in Form von C++-Code. Der Bezug zu den Objekten auf den umliegenden Stellen wird durch die Variablennamen der inzidenten Kanten einer Transition hergestellt. Um dabei Namenskonflikte auszuschließen, müssen die Variablennamen in der Umgebung einer Transition eindeutig sein. Wenn das Gewicht einer Kante eins ist, kann der Variablenname direkt in der Transitionsbeschriftung verwendet werden. Andernfalls erfolgt eine Indizierung der Variablen entsprechend der Array-Notation von C++ (z. B. `x[0]` und `x[1]` bei einem Kantengewicht von zwei und der Kantenvariablen `x`). Die Typen der Variablen werden durch den Typ der jeweiligen Stelle bestimmt.

Die Schaltbedingung einer Transition wird als boolescher C++-Ausdruck über den Variablen *eingehender* Kanten formuliert. Sie darf allerdings keine Seiteneffekte auf die referenzierten Objekte haben, da sie unabhängig von einem tatsächlichen Schalten ausgewertet wird – Objektwerte dürfen also nur gelesen und nicht verändert werden. Die Schaltbedingung muß für eine Belegung der Variablen mit Objekten erfüllt sein, damit die Transition unter der Belegung aktiviert ist. Die Kombination einer Transition mit einer Variablenbelegung wird in Thor-Netzen auch als *Schaltereignis* bezeichnet.

Die Schaltaktion einer Transition wird in Form einer C++-Anweisungsfolge angegeben. In dieser Anweisungsfolge können sowohl Variablen *ein- und ausgehender* Kanten als auch

lokale Hilfsvariablen verwendet werden. Die Schaltaktion wird beim Schalten der Transition ausgeführt. Sie wird in der Regel zum Erzeugen und Modifizieren von Objekten verwendet. Dadurch daß an dieser Stelle beliebiger C++-Code angegeben werden kann, ergeben sich aber prinzipiell auch ganz andere Möglichkeiten. Beispielsweise kann beim Schalten einer Transition ebenso gut ein Systemaufruf oder der Aufruf einer Routine aus einer dazugebundenen Bibliothek (z. B. Xlib) erfolgen. Die Verwendung solcher Seiteneffekte ist im Hinblick auf eine verteilte Simulation allerdings sehr problematisch.

3.2 Zeitkonzepte

Erst die Möglichkeit zur adäquaten Beschreibung zeitlicher Abläufe und Abhängigkeiten eines Systems erlaubt es, auch zeitbezogene Aussagen über ein System aus einem Modell abzuleiten. Je genauer Ausführungs- und Wartezeiten einzelner Prozesse, chronologisch korrekte Bearbeitungsfolgen, deren Synchronisation und andere zeitliche Zwänge modelliert werden können, desto aussagekräftiger sind die aus dem Modell gewonnenen Eigenschaften zu bewerten.

Für Petrinetze wurden zu diesem Zweck eine Reihe ganz unterschiedlicher Konzepte vorgeschlagen. Das zeitliche Verhalten wurde dabei sowohl mit Stellen und Transitionen als auch mit Kanten und Marken in Verbindung gebracht. Ein Überblick über die verschiedenen Zeitbeschriftungen der Netzelemente ist in [Bau90] und [Sta90] angegeben. Am verbreitesten in der Literatur sind jedoch Ansätze, bei denen das zeitliche Verhalten eines Systems durch eine erweiterte Beschriftung von Transitionen modelliert wird. Diese Tatsache mag darin begründet sein, daß Transitionen üblicherweise zur Beschreibung von Aktivitäten verwendet werden und diese in der Regel mit einer Bearbeitungszeit oder Wartezeit verbunden sind.

Die ersten Arbeiten zu zeitbewerteten Transitionen gehen auf Ramchandani [Ram74] und Merlin [Mer74] zurück. Bei den von Ramchandani eingeführten *Timed Petri Nets* (s. a. [HV87]) werden Transitionen mit einer festen Schaltdauer versehen, mit der beispielsweise die Bearbeitungszeit eines Prozesses beschrieben werden kann. Die beim Schaltbeginn konsumierten Marken befinden sich während des Schaltens im Innern der Transition und werden erst am Schaltende auf den Nachbereichsstellen produziert – in der Zwischenzeit sind sie für andere Transitionen nicht verfügbar. Bei Merlins *Time Petri Nets* (s. a. [Pop89, BD91]) werden die Transitionen mit einem Intervall $[\alpha,\beta]$ ($\alpha \leq \beta$) beschriftet, wobei mit α der früheste und mit β der späteste Schaltzeitpunkt nach einer ununterbrochenen Aktiviertheit festgelegt wird. Das Schalten einer Transition erfolgt ohne Zeitverbrauch zu einem beliebigen Zeitpunkt in dem Intervall. Die zum Schalten benötigten Marken werden dabei nicht reserviert, d. h. sie können während der Wartezeit von anderen Transitionen konsumiert werden. Auch mit diesem Ansatz lassen sich Bearbeitungszeiten beschreiben.

Beide Ansätze wurden als Ergänzungen für Stellen/Transitionen-Netze definiert. Sie dienten als Vorbild für viele der nachfolgenden Petrinetzklassen, die die Möglichkeit zur Beschreibung zeitlicher Aspekte bieten. Dazu zählen insbesondere die im Rahmen der Leistungsbewertung von Rechnersystemen intensiv untersuchten *Stochastic Petri Nets* (vgl. [Ajm89, Mol82] und [JR91, Abschnitt G]. Aufgrund der besseren Ausdrucksmöglichkeiten bei höheren Petrinetzen wurden auch hier diverse Erweiterungen eingeführt, um zeitliche

Vorgänge beschreiben zu können [Aal93, GMMP91, KKT96, Stu95]. Häufig sind dabei Konzepte entstanden, bei denen das Zeitverhalten direkt den Marken bzw. Objekten zugeordnet wird oder von diesen abhängig gemacht werden kann.

Das in THOR-Netzen realisierte Zeitkonzept besteht aus einer Kombination der bekannten Verfahren, wobei zusätzliche Möglichkeiten integriert wurden, die auf dem Objektkonzept basieren. Wie üblich werden auch hier Transitionen mit einer Zeitbeschriftung versehen. Im Gegensatz zu vielen anderen Ansätzen kann für jede Transition jedoch sowohl eine *Verzögerungszeit* als auch eine *Schaltdauer* angegeben werden. Die Verzögerungszeit legt dabei den Zeitraum fest, über den eine Transition ununterbrochen aktiviert sein muß, bevor sie mit dem Schalten beginnen kann. Sie wird meistens dazu verwendet wird, eine Vorbereitungs-, Warte- oder Rüstzeit eines Prozesses zu modellieren. Die Schaltdauer definiert die Zeit, die für den Schaltvorgang selbst erforderlich ist – die Zeit, die zwischen Schaltbeginn und Schaltende vergeht. Sie wird häufig zur Modellierung einer Bearbeitungs- oder Ausführungszeit einer Aktivität benutzt. Beide Zeiten können von den Attributen referenzierter Vorbereichsobjekte abhängen und mit Hilfe von Zufallszahlengeneratoren variiert werden.

Insgesamt liegt dem THORN-Zeitkonzept eine kontinuierliche Zeitskala zugrunde, so daß beide Zeiten nichtnegative reelle Werte annehmen können. Insbesondere ist es auch erlaubt, Nullzeiten anzugeben. Im Gegensatz zu GSPNs [Ajm89] haben Transitionen mit nullwertiger Schaltdauer jedoch bei der Ausführung keine höhere Priorität als Transitionen, die mit einer positiven Schaltdauer beschriftet sind.

Die Spezifikation der Verzögerungszeit und der Schaltdauer erfolgt in Form eines gültigen C++-Ausdrucks, dessen Auswertung ein nichtnegatives Ergebnis ($\in \mathbb{R}_0^+$) liefert. Dabei können Konstanten, Variablen eingehender Kanten, C++-Operationen, -Methoden und -Funktionen sowie Zufallszahlengeneratoren aus der bereitgestellten Bibliothek verwendet werden. Die Auswertung der Zeitbeschriftungen darf allerdings keine Seiteneffekte auf die gebundenen Vorbereichsobjekte haben, d. h. auf die Attribute dieser Objekte darf nur lesend zugegriffen werden.

3.2.1 Schaltregel

Bevor die Möglichkeiten des hierarchischen Entwurfs von THOR-Netzen näher beschrieben werden, soll zunächst die Schaltregel für flache zeitbeschriftete THORNs erläutert werden. Da die Schaltregel in höheren Petrinetzklassen nicht mehr unabhängig von der Belegung der Variablen einer Transition mit einer Objektkombination betrachtet werden kann, wird sie hier für Transitionen und Variablenbelegungen definiert. Diese Kombination wird in THOR-Netzen auch als *Schaltereignis* bezeichnet.

Die Schaltregel für zeitbeschriftete flache Netze besteht im wesentlichen aus einem zweistufigen Aktiviertheitsbegriff für Schaltereignisse und den Zustandsänderungen, die beim Schaltbeginn bzw. beim Schaltende eines Schaltereignisses durchgeführt werden.

Ein Schaltereignis erfüllt in einem Zustand (einer Markierung) die *Vorbereichsbedingung*, wenn

- alle inhibitorisch wirkenden Stellen im Vorbereich leer sind,

- alle Vorbereichsstellen, die über Standard- oder aktivierende Kanten mit der Transition verbunden sind, nicht weniger Objekte enthalten als durch das Gewicht der verbindenden Kante gefordert wird und
- durch die Belegung der Variablen dieser Kanten mit Objekten der inzidenten Stellen die Schaltbedingung der Transition erfüllt wird.

Die Vorbereichsbedingung umfaßt einen Teil der Bedingungen, die erfüllt sein müssen, um eine Transition zu aktivieren. Bedingungen bzgl. der Schaltkapazität und des Nachbereichs werden zunächst nicht geprüft.

Alle Schaltereignisse, die in einem Zustand die Vorbereichsbedingung erfüllen, werden *verzögert*. Die Verzögerungszeit wird zu Beginn der Verzögerung durch Auswerten des entsprechenden Ausdrucks der Transition ermittelt. Schaltereignisse, die über den Zeitraum der Verzögerung (oder länger) ununterbrochen die Vorbereichsbedingung erfüllen, heißen *vollständig verzögert*. Ein Grund für die Unterbrechung bzw. den Abbruch der Verzögerung eines Schaltereignisses ist das Belegen einer inhibitorisch wirkenden Stelle mit einem Objekt bzw. das Entfernen von Objekten, die an dem Schaltereignis beteiligt sind, durch das Schalten anderer Transitionen.

Ein Schaltereignis ist in einem Zustand *aktiviert*, wenn es vollständig verzögert ist, die Transition über freie Schaltkapazität verfügt und ausreichend Stellenkapazitäten im Nachbereich der beteiligten Transition vorhanden sind. Im Fall von Schlingen zwischen Stellen und Transitionen (Kanten in beiden Richtungen) wird die benötigte Stellenkapazität entsprechend der schwachen Schaltregel [Bra84] ermittelt, d. h. es reicht aus, wenn der Platz auf der Stelle nach Abzug der konsumierten Objekte groß genug ist. Ein aktiviertes Schaltereignis kann mit dem Schalten beginnen.

Beim *Schaltbeginn* eines Schaltereignisses werden zunächst die in der Variablenbelegung gebundenen Vorbereichsobjekte unter Berücksichtigung der Stellenstrukturen und der Kantentypen von den Vorbereichsstellen abgezogen bzw. im Fall von Enabling-Kanten kopiert. Danach werden die ausgehenden Objekte der Transition entsprechend dem Typ der jeweiligen Nachbereichsstelle und in der Vielfachheit des jeweiligen Kantengewichts mit Hilfe des Standard-Konstruktors ihres Objekttyps erzeugt. Schließlich wird mit diesen Objekten die Schaltdauer für das Schaltereignis bestimmt und die Schaltaktion durchgeführt. Ist die Schaltdauer Null, werden die Ausgangsobjekte direkt auf den Nachbereichsstellen abgelegt. Andernfalls wird die Schaltkapazität der Transition dekrementiert und im Nachbereich der Transition entsprechend der Kantengewichte ausgehender Kanten ausreichend Platz für die Ausgangsobjekte reserviert.

Für Schaltereignisse mit positiver Schaltdauer wird zum Zeitpunkt des *Schaltendes* die Schaltkapazität der Transition wieder inkrementiert, reservierte Plätze werden freigegeben und die beim Schaltbeginn berechneten Ausgangsobjekte im Nachbereich abgelegt.

Das Schalten eines Ereignisses ist ein atomarer Vorgang, der im Gegensatz zur Verzögerung nach dem Schaltbeginn nicht mehr unterbrochen werden kann. Anschaulich befinden sich die konsumierten Objekte während des Schaltens im Innern der Transition; implementierungstechnisch werden sie mit einem Zeitstempel versehen, der größer als die aktuelle Simulationszeit ist, und auf den Nachbereichsstellen gespeichert.

Zu jedem Zeitpunkt der Ausführung eines zeitbeschrifteten Netzes beginnt eine maximale Menge aktivierter Schaltereignisse zu schalten und alle Ereignisse, deren Schaltdauer abgelaufen ist, beenden ihren Schaltvorgang. Im Gegensatz zu vielen anderen Petrinetzklassen, bei denen eine Menge gemeinsam aktivierter Schaltereignisse bestimmt und geschlossen gefeuert wird (wie z. B. bei CP-Netzen [Jen92]), werden bei THOR-Netzen sukzessive einzelne Schaltereignisse bearbeitet. Dabei wird entweder ein aktiviertes Schaltereignis gestartet oder ein Schaltereignis, dessen Schaltdauer abgelaufen ist, beendet. Die Auswahl erfolgt nichtdeterministisch. Dieser Vorgang wird so lange wiederholt, bis keine Schaltereignisse dieser Art mehr vorhanden sind. Da auf diese Weise zu jedem Zeitpunkt eine maximale Menge aktivierter Schaltereignisse geschaltet wird, wird die Schaltregel auch als *Maximum-Sofort-Schaltregel* bezeichnet.

Man beachte, daß durch die Bearbeitung eines Schaltbeginn- oder Schaltende-Ereignisses andere Schaltereignisse aktiviert oder deaktiviert werden können und sich damit die Menge der aktivierten Schaltereignisse zu einem Zeitpunkt ständig ändert. Insbesondere kann es vorkommen, daß es durch wiederholtes Schalten von Transitionen mit nullwertiger Schaltdauer zu einem sogenannten *livelock* oder *timelock* kommt, bei dem die Simulationszeit nicht mehr fortschreitet.

Der *Zustand* eines zeitbeschrifteten flachen THOR-Netzes wird zu jedem Zeitpunkt durch die aktuelle Markierung und die reservierten Plätze, sowie durch die aktuellen Restverzögerungs- und Restschaltzeiten der Schaltereignisse eindeutig bestimmt.

3.3 Hierarchiekonzepte

Hierarchiekonzepte dienen in höheren Modellierungssprachen zur Strukturierung und Modularisierung von Modellen. Erst durch diese Konzepte wird es möglich, auch komplexe Systeme übersichtlich und kompakt darzustellen und die Struktur des Systems in natürlicher Weise nachzubilden. Des weiteren unterstützen Hierarchiekonzepte die Modellierung von Systemen auf unterschiedlichen Abstraktionsebenen und die Wiederverwendbarkeit einzelner Modellteile an verschiedenen Stellen des Modells.

Früher wurde das Fehlen von hierarchischen Strukturierungsmöglichkeiten bei Petrinetzen häufig kritisiert. In der jüngeren Vergangenheit sind deshalb eine Vielzahl unterschiedlicher Hierarchiekonzepte zur Beseitigung dieses Defizits vorgeschlagen worden. Der naheliegendste Ansatz bei Petrinetzen besteht in der Verfeinerung von Stellen und Transition. Dabei handelt es sich um eine einfache Ersetzung eines Knotens durch ein verfeinerndes Unternetz. In der Terminologie der Programmiersprachen entspricht dieser Vorgang einer Makroexpansion. Dieses Verfahren wurde erstmals von Genrich, Lautenbach und Thiagarajan [GLT80] diskutiert; eine Übersicht zur Theorie dieser Verfeinerungskonzepte bieten Brauer, Gold und Vogler [BGV90]; im Gegensatz zu den ursprünglich eingeführten Verfahren erlaubt Fehling in seinen Arbeiten [Feh91, Feh92] auch die Verfeinerung benachbarter Knoten. Theoretisch wird durch die Verfeinerung von Stellen und Transition die Semantik eines Netzes nicht verändert – jedes hierarchische Netz kann leicht in ein äquivalentes flaches Netz überführt werden. Dieses Konzept dient damit in erster Linie zur Strukturierung und übersichtlichen Darstellung von Modellen. Ein Vorteil dieses

Hierarchiekonzepts besteht darin, daß bekannte Analysemethoden der zugrundeliegenden Netzklasse direkt auf die hierarchischen Netze übertragen werden können.

Ein anderer Ansatz wurde von Cherkasova und Kotov [CK81] vorgeschlagen. Bei diesem Konzept werden Unternetze von sogenannten strukturierten Transitionen aufgerufen – im Programmiersprachgebrauch ist dieses Konzept vergleichbar mit einem Prozeduraufruf. Wenn eine strukturierte Transition schaltet, konsumiert sie zunächst ihre Vorbereichstoken und startet anschließend ein Unternetz mit einer vordefinierten Initialmarkierung, die für jeden Aufruf eines Unternetzes gleich ist. Während ein Unternetz aktiv ist, kann es über sogenannte externe „Share"-Stellen mit der Aufrufumgebung Token austauschen. Falls in dem Unternetz keine Transitionen mehr schalten können, terminiert es und die aufrufende Transition produziert ihre Nachbereichstoken. Rekursive Aufrufe von Unternetzen sind bei diesem Ansatz nicht erlaubt. Die Erweiterung um Rekursion wurde erst von Kiehn [Kie89a, Kie89b] eingeführt. Außerdem definierte sie ein anderes Terminierungskriterium für Unternetze. Hier muß eine definierte Endmarkierung erreicht werden und es darf keine Aufruftransition mehr aktiv sein, damit ein Unternetz terminiert.

Die bisher vorgestellten Hierarchiekonzepte basieren auf einfachen Stellen/Transitionen-Netzen. Die Übertragung und Erweiterung dieser Konzepte auf gefärbte Petrinetze wurde ausführlich von Huber, Jensen und Shapiro [HJS90] behandelt. In ihrer Arbeit werden insgesamt fünf verschiedene Hierarchiekonzepte präsentiert: Stellen- und Transitionen-Verfeinerung, Aufruf-Transitionen und Stellen- und Transitionen-Verschmelzung. Die Verfeinerungskonzepte sind im wesentlichen mit den entsprechenden Konzepten einfacher Netzklassen vergleichbar. Im Gegensatz dazu wird das Konzept der Aufruf-Transitionen hier so erweitert, daß die aufrufende Transition Objekte (Token) an das Unternetz übergibt und umgekehrt. Rekursion ist erlaubt und die Terminierung eines Unternetzes wird durch die Belegung einer ausgezeichneten Ausgangsstelle oder das Schalten einer ausgezeichneten Ausgangstransition festgestellt. Bei Terminierung eines Unternetzes werden alle Objekte von den Ausgabestellen auf ihre korrespondierenden Stellen im Nachbereich der aufrufenden Transition übergeben und die Instanz des Unternetzes gelöscht. Die Konzepte der Verschmelzung von Knoten (fusion sets) ermöglichen es einem Modellierer, verschiedene Transitionen oder Stellen miteinander zu identifizieren, ohne sie graphisch als einzelne Objekte darzustellen. Die Verschmelzung ist auch zwischen Knoten unterschiedlicher Unternetze erlaubt. Die Stellen-Verschmelzung ist damit mit den Share-Stellen von Cherkasova und Kotov vergleichbar.

Die hier aufgeführten Arbeiten zu Hierarchie in Petrinetzen stellen nur einen kleinen Ausschnitt der insgesamt verfügbaren Veröffentlichungen zu diesem Thema dar. Allerdings basieren die meisten Arbeiten auf den vorgestellten Konzepten. Für PrT-Netze wurde bspw. von Rammig [Ram93] ein Konzept eingeführt, das eng an den Konzepten von Cherkasova und Kotov angelehnt ist, zusätzlich aber rekursive Unternetzaufrufe erlaubt.

Im folgenden wird das Hierarchiekonzept der THOR-Netze beschrieben. Im Vergleich zu den vorgestellten Verfahren stellt es eine Kombination der Konzepte der Aufruftransition und Stellenverschmelzung von Huber, Jensen und Shapiro [HJS90] bzw. den Share-Stellen von Cherkasova und Kotov [CK81] und klassischer Transitionenverfeinerung dar. Es erlaubt Rekursion, Objektübergabe an Unternetze, partielle Objektrückgabe aus Unternetzen und besitzt ein an das Schalten ausgezeichneter Stop-Transitionen geknüpftes Terminierungskriterium. Verschiedene Unternetze können über Share-Stellen miteinander

kommunizieren und der Einsatz von Schaltkapazitäten bei Aufruftransitionen ermöglicht das Aufrufen einer beschränkten Anzahl von parallel existierenden Unternetzen.

Ein hierarchisches Netz (ein THOR-Netz) besteht aus einer endlichen Folge von Unternetzen. Dabei existiert genau ein ausgezeichnetes Hauptnetz oder Startnetz. Unternetze können sich gegenseitig dynamisch mittels *Aufruftransitionen* erzeugen. Die Terminierung eines Unternetzes wird durch das Schalten von *Stop-Transitionen* bestimmt. Jedes Unternetz kann keine, eine oder mehrere Aufruf- und Stop-Transitionen enthalten (nur das Hauptnetz darf keine Stop-Transition haben). Beide Transitionsarten sind bis auf die Schaltaktion so beschriftet wie Standardtransitionen. Aufruftransitionen werden anstelle mit einer Schaltaktion mit dem Namen des Unternetzes versehen, das sie repräsentieren und Stop-Transitionen haben keine Schaltaktion. Zur Unterscheidung von Standardtransitionen haben diese Transitionen eine andere Darstellung (vgl. Abb. 3.3).

Die Interaktion zwischen dem aufrufenden Netz und dem Unternetz erfolgt über die Randstellen der Aufruftransition. Jede Randstelle einer Aufruftransition muß dazu genau eine korrespondierende Stelle im Unternetz besitzen. Es wird dabei zwischen *Eingabe-*, *Ausgabe-* und *Einbettungsstellen* unterschieden. Die Unterteilung ergibt sich durch die Art der verbindenden Kante: Eingabe- und Ausgabestellen – auch Übergabestellen genannt – sind mit der Aufruftransition durch ein- bzw. ausgehende Standardkanten verbunden während Einbettungsstellen durch den zusätzlichen Kantentyp einer *Hyperkante* mit der Aufruftransition in Beziehung gesetzt werden – Hyperkanten sind ungerichtete Kanten, können aber mit Pfeilspitzen versehen werden, um einen bestimmten Objektfluß zwischen dem aufrufenden Netz und dem Unternetz graphisch zu illustrieren. Die Mengen der Übergabe- und Einbettungsstellen müssen disjunkt sein. Stellen, die durch Enabling-, Inhibitor- oder Consuming-Kanten mit der Aufruftransition verbunden sind, werden nicht als Randstellen bezeichnet. Sie werden von der Aufruftransition zwar genauso behandelt wie von Standardtransitionen, haben aber keine entsprechende Stelle im Unternetz.

Die zu den Randstellen korrespondierenden Stellen im Unternetz werden mit den entsprechenden englischen Übersetzungen bezeichnet: *Input-*, *Output-* und *Share-Stellen*. Damit die Schnittstelle wohldefiniert ist, müssen die Typen korrespondierender Stellen zueinander passen – weitere Details zur konsistenten Beschriftung dieser Stellen werden im Handbuch [Wie96a] beschrieben. Zur Unterscheidung der Randstellen eines Unternetzes von anderen Stellen haben diese eine besondere graphische Darstellung. Abbildung 3.3 zeigt alle hierarchischen Netzelemente eines THOR-Netzes im Überblick.

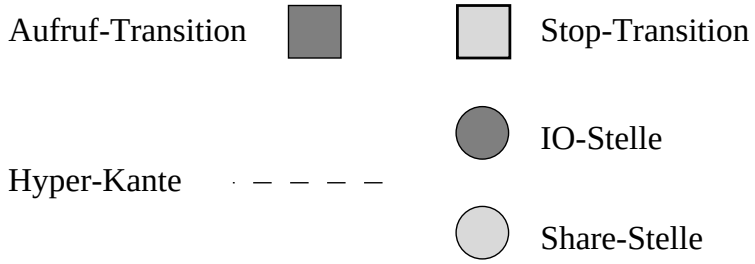


Abbildung 3.3: Darstellung der hierarchischen Netzelemente

Anschaulich entspricht der Aufruf eines Unternetzes einem Prozeduraufruf in einer höher-

en Programmiersprache. Die Input- und Output-Stellen des Unternetzes sind dabei mit formalen Parametern vergleichbar während die Share-Stellen globalen Parametern entsprechen. Die Parameterübergabe an Input- und von Output-Stellen erfolgt nach dem call-by-value- bzw. call-by-result-Prinzip. Die genaue Semantik wird durch folgende Schaltregel definiert.

3.3.1 Schaltregel

Die Schaltregel für hierarchische Netze basiert im wesentlichen auf der Schalteregel für zeitbeschriftete flache Netze aus Abschnitt 3.2.1. Für Aufruf- und Stop-Transitionen gelten dieselben Bedingungen zur Verzögerung und Aktiviertheit wie für Standardtransitionen – Hyper-Kanten haben dabei wie Consuming-Kanten keine Auswirkungen auf die Aktiviertheit einer Aufruftransition – und bei der Auswahl von Schaltereignissen werden sie ebenfalls gleichberechtigt behandelt. Daher wird an dieser Stelle nur noch das unterschiedliche Verhalten der Transitionen beim Schalten genauer beschrieben.

Wenn ein Schaltereignis einer Aufruftransition aktiviert ist, kann es mit dem Schalten beginnen. Dabei werden die an dem Schaltereignis beteiligten Objekte von den Eingabestellen konsumiert, die Schaltkapazität der Transition wird dekrementiert und auf den Stellen im Nachbereich wird entsprechend den Kantengewichten ausgehender Kanten ausreichend Platz reserviert. Nach Ablauf der Schaltdauer (welche üblicherweise Null ist) wird eine neue Instanz des aufgerufenen Unternetzes generiert, die zuvor konsumierten Objekte werden auf die entsprechenden Input-Stellen des Unternetzes transferiert, die Share-Stellen des Unternetzes werden mit „ihren“ Einbettungsstellen identifiziert (fusioniert im Sinne der Stellen-Verschmelzung von [HJS90]) und eine vordefinierte Initialmarkierung für die internen Stellen des Unternetzes wird geladen. Anschließend können die Transitionen in dem neuen Unternetz gleichberechtigt und nebenläufig zu den bestehenden Unternetzen schalten, bis eine Stop-Transition in dem Netz schaltet.

Stop-Transitionen dienen lediglich dem Zweck, Unternetze zu terminieren. Deshalb haben sie keine Schaltaktion und einen leeren Nachbereich. Beim Schaltbeginn einer Stop-Transition werden die an dem Schaltereignis beteiligten Objekte unter Berücksichtigung der Kantentypen aus dem Vorbereich abgezogen und es wird ein Stop-Signal für das Unternetz und alle davon aufgerufenen Unternetze gesetzt. Das Stop-Signal verbietet es allen Transitionen in diesen Netzen, einen neuen Schaltvorgang zu beginnen. Die Unternetze werden blockiert – dementsprechend muß die Vorbereichsbedingung aus Abschnitt 3.2.1 um einen Test auf blockierte Netze erweitert werden. Schaltende Transitionen (inkl. der Stop-Transition selbst) dürfen ihren Schaltvorgang allerdings noch beenden. Sobald alle Transitionen ihre Schaltvorgänge abgeschlossen haben, terminiert das Unternetz und damit die entsprechende Aufruftransition. Der Inhalt der Output-Stellen wird an die korrespondierenden Ausgabestellen im aufrufenden Netzes übergeben und der dort beim Aufruf des Unternetzes reservierte Platz wieder freigegeben – dabei kann es durchaus vorkommen, daß der reservierte Platz nicht komplett benötigt wird. Schließlich wird die Schaltkapazität der aufrufenden Transition inkrementiert und die Instanz des Unternetzes und die der aufgerufenen Unternetze gelöscht.

Ein Spezialfall des vorgestellten Konzepts ist die klassische Transitionenverfeinerung. Sie ergibt sich, wenn ein Unternetz weder Input- und Output-Stellen noch Stop-Transitionen

enthält und die aufrufende Transition eine Schaltkapazität von eins hat. In diesem Fall wird das Unternetz zusammen mit dem aufrufenden Netz erzeugt. Da es nicht terminiert, existiert es genauso lange wie das aufrufende Netz. Auf der anderen Seite können sich Unternetze aber auch durch rekursive Aufrufe oder unbegrenzte Schaltkapazitäten von Aufruftransitionen beliebig oft dynamisch vermehren. Weitere Einzelheiten zu der Beschriftung von Netzelelementen und der Schaltregel im Zusammenhang mit der hierarchischen Strukturierung von THOR-Netzen sind im Handbuch angegeben [Wie96a].

3.4 Analysemöglichkeiten

Aufgrund der Komplexität der Modellierungssprache ist eine formale Analyse von THORN-Modellen in der klassischen Form [Sta90] nicht möglich. In der Literatur sind zwar Analyseverfahren auch für höhere Netzklassen bekannt, dabei werden aber oft starke Einschränkungen vorausgesetzt. Beispielsweise können (zeitlose) CP-Netze formal oft nur untersucht werden, wenn sie zu flachen Stellen/Transitionen-Netzen entfaltet werden können, d. h. wenn die Objekttypen (Farben) endliche Wertebereiche besitzen und ein statisches Hierarchiekonzept zugrundegelegt wird, oder wenn ihr Erreichbarkeitsgraph endlich ist [Jen95]. In ähnlicher Form wird Analyse auch bei OBJSA-Netzen durchgeführt [BDM96]. Wenn bei Netzklassen mit individuellen Marken zusätzlich zeitliche Aspekte berücksichtigt werden sollen, so ergeben sich daraus weitere Schwierigkeiten. Solche Modelle können häufig nur mit Hilfe von Simulationen untersucht werden. Eine Übersicht zu den Analysemethoden höherer Petrinetze und eine Einschätzung der Anwendungsmöglichkeiten ist in [Rei94] angegeben.

Da in THOR-Netzen individuelle Objekte mit unendlichen Wertebereichen definierbar sind, das Hierarchiekonzept dynamische Unternetzaufrufe zuläßt und das Zeitkonzept objektabhängige, deterministische und stochastisch verteilte Zeitbeschriftungen vorsieht, wurde in dem Projekt DNS zur Analyse von THOR-Netz-Modellen von vornherein die Methode der Simulation gewählt. Das vorrangige Ziel des Projekts bestand darin, effiziente Simulatoren zu entwickeln, um auch bei größeren Modellen und längeren Simulationsläufen möglichst kurze Ausführungszeiten zu erreichen.

3.5 Fallstudien

Die Einsatzmöglichkeiten von THOR-Netzen bei der Modellierung komplexer diskreter Systeme wurden in einer Reihe von Fallstudien in ganz unterschiedlichen Anwendungsbereichen gezeigt. Darunter sind Modelle zu industriellen Fertigungsprozessen und Steuerungsproblemen [KTWZ93], Kommunikationsprotokollen [Kos94, SSW95a], ökologischen Systemen [GS94], Geschäftsprozessen [Bar94] und Verkehrssystemen in den Bereichen des Indivudualverkehrs [Wie95] und des öffentlichen Personennahverkehrs [Sch96a].

Gerade das letzte Modell, das im Rahmen einer Diplomarbeit an der Universität Stuttgart entwickelt wurde, demonstriert die Einsatzmöglichkeiten von THOR-Netzen bei der Modellierung und Simulation von komplexen realen Anwendungen. Mit dem THOR-Netzmodell des Stuttgarter U- und S-Bahn-Netzes, das aus 722 Transitionen, 1531 Stellen und 333

Aufruftransitionen besteht, wurden verschiedene Untersuchungen wie z. B. die simulative Bestwegsuche für vorgegebene Start- und Zielorte angestellt (vgl. [Sch96a]).

Bei der Durchführung der ersten Fallstudien wurden schnell kleinere Mängel und Verbesserungsmöglichkeiten an der ursprünglichen Definition [FLSW93b] festgestellt. Daraufhin wurde die Netzdefinition in einigen Punkten modifiziert, um die Modellbildung insgesamt zu verbessern und zu vereinfachen. Die wesentlichen Änderungen bestanden in der Einführung einer kontinuierlichen (reellwertigen) Zeitskala, der Möglichkeit, nullwertige Schaltdauern zu definieren, dem Verzicht auf echte Multimengen-Stellen und einer Vereinfachung der Zuordnung von Stellen einer Aufruftransition zu den Stellen des Unternetzes (vgl. dazu [SWK+94, Anhang A]). Bei der in diesem Abschnitt beschriebenen Definition der THOR-Netze wurden diese Änderungen berücksichtigt.

Die Fallstudien haben insgesamt gezeigt, daß eine Modellbildung mit THOR-Netzen zu einfachen und kompakten Modellen führt, solange sich die Systeme in natürlicher Weise durch diskrete Zustandsänderungen beschreiben lassen. Sobald jedoch kontinuierliche Veränderungen des Systems in das Modell integriert werden müssen, fällt eine Modellierung mit THOR-Netzen schwer oder ist gar nicht mehr korrekt möglich. Um in Zukunft auch Systeme mit höheren Netzen modellieren zu können, in denen sowohl diskrete als auch kontinuierliche Zustandsänderungen auftreten, wurde in der zweiten Projektphase eine hybride Netzklasse entwickelt, die auf THORNs basiert und eine Modellbildung hybrider Systeme ermöglicht (vgl. dazu Kapitel 8).

3.6 Zusammenfassung

THOR-Netze sind eine spezielle Klasse höherer Petrinetze, in der viele bekannte high-level Konzepte miteinander kombiniert sind. THOR-Netze verfügen durch die Verwendung der objektorientierten Beschriftungssprache C++ über ein mächtiges Objektkonzept. Damit lassen sich auch sehr komplexe Objekttypen und umfangreiche Schaltaktionen spezifizieren. Das Zeitkonzept der THOR-Netze ermöglicht die adäquate Modellierung zeitlicher Abläufe und Abhängigkeiten und mit Hilfe des Hierarchiekonzepts können auch große Modelle kompakt und übersichtlich beschrieben werden. Diese Kombination von high-level Konzepten zeichnet THOR-Netze gegenüber anderen höheren Petrinetzklassen aus, die häufig eines oder mehrere dieser Konzepte vernachlässigen oder gar nicht anbieten.

Kapitel 4

Werkzeuge für THOR-Netze

Zur Erstellung, Bearbeitung und Auswertung von THOR-Netz-Modellen wurde in dem Projekt DNS eine Entwicklungsumgebung bestehend aus einem interaktiven graphischen Netzeditorsystem, einem Netzcompiler und effizienten Simulatoren für eine sequentielle und verteilte Ausführung der Netze entwickelt und prototypisch implementiert.

In diesem Kapitel wird zunächst ein allgemeiner Überblick über die Systemarchitektur der entwickelten Werkzeuge angegeben. Anschließend wird die graphische Oberfläche, der Netzcompiler und der sequentielle Simulator beschrieben. Die Beschreibung des verteilten Simulators erfolgt in einem eigenen Kapitel.

4.1 Systemübersicht

Das grundlegende Konzept für eine effiziente Ausführung von THOR-Netz-Modellen ist deren Übersetzung in ausführbare Simulationsprogramme. Das Zusammenspiel der einzelnen Systemteile bei der Erstellung eines Simulators für ein Netz wird schematisch in Abbildung 4.1 dargestellt.

Nachdem ein Netzmodell mit Hilfe des graphischen Netzeditors eingegeben wurde, kann durch Verwendung des Netzcompilers wahlweise Code für den sequentiellen oder den verteilten Simulator erzeugt werden. Für den sequentiellen Simulator wird dazu das Netz inklusive sämtlicher Beschriftungen nach C++ compiliert. Für den verteilten Simulator wird die Netzstruktur in ein kompaktes Austauschformat transformiert und die C++-Beschriftungen (Transitionsbeschriftungen und Objekttyp-Definitionen) nur soweit gekapselt, daß sie vom Simulator angesprochen werden können. Der generierte C++-Code wird anschließend von einen C++-Compiler mit einem Simulatorkern zu einem ausführbaren Programm gebunden. Je nach Art der gewünschten Simulationsform entsteht so ein sequentieller oder verteilter Simulator.

Die Simulation eines Netzes wird direkt aus der Editorumgebung heraus gestartet. Dabei werden dem jeweiligen Simulator einige Simulationsoptionen (max. Schrittzahl, max. Simulationszeit, etc.) und der Anfangszustand (Markierung) übergeben. Für den verteilten Simulator wird der Anfangszustand zuvor automatisch durch den Netzcompiler in ein kompaktes Austauschformat transformiert. Nach der Ausführung eines Netzes wird der

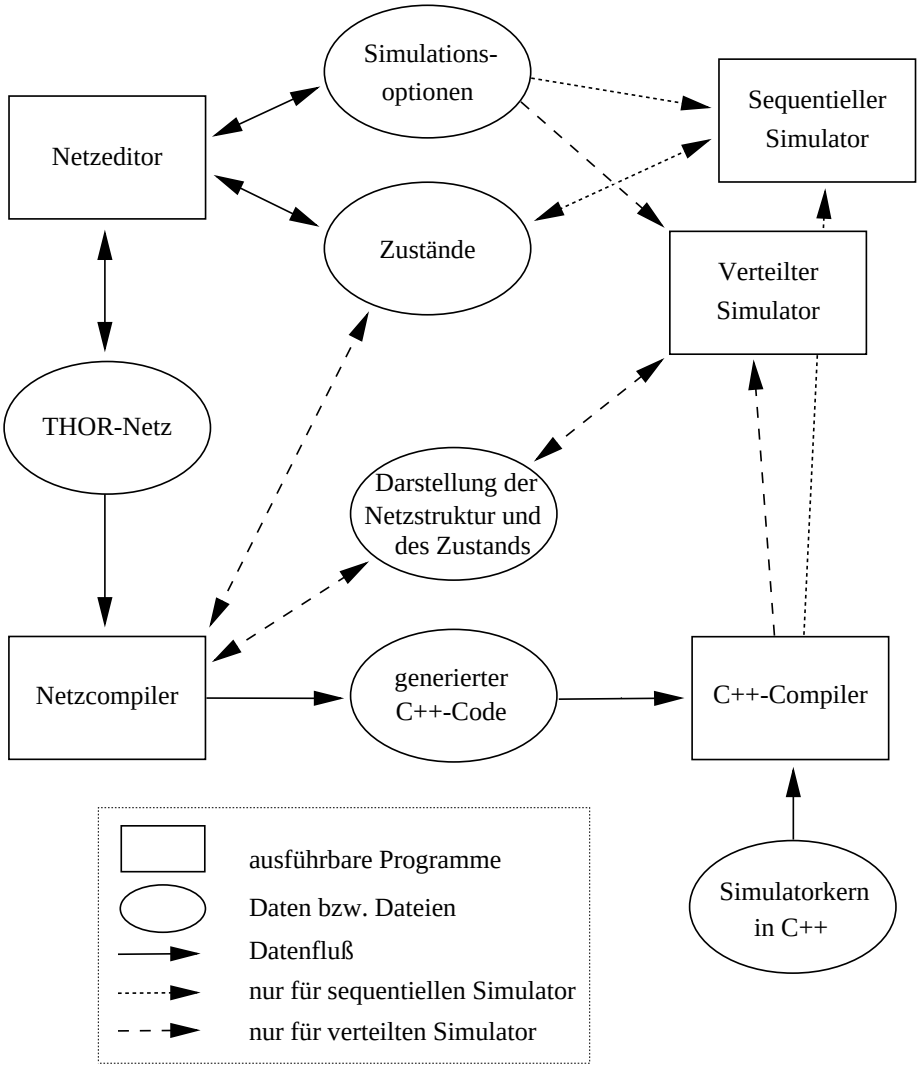


Abbildung 4.1: Systemarchitektur der Entwicklungsumgebung

erreichte Endzustand wieder über eine Datei an den Editor zurückgegeben – bei einer verteilten Simulation erfolgt dabei wiederum eine Rückübersetzung des Austauschformats in ein vom Editor lesbares Format durch den Netzcompiler. Die Benutzerinteraktion findet während des gesamten Prozesses ausschließlich über den graphischen Editor statt.

Sämtliche Daten zu einem THOR-Netz werden lokal zu dem Netz in einem Netzordner gespeichert. Dazu gehören z. B. Dateien für die textuelle Repräsentation des Netzes, für den generierten C++-Code, die ausführbaren Simulatoren und Parameter-Daten.

In den folgenden Abschnitten werden der graphische Netzeditor, der Netzcompiler und der sequentielle Simulator etwas näher betrachtet. Der verteilte Simulator wird anschließend in einem eigenen Kapitel beschrieben.

4.2 Der Netzeditor

Der graphische Netzeditor repräsentiert die interaktive Schnittstelle zum Benutzer. Mit dem Editor können folgende Aufgaben bearbeitet werden:

- *Eingabe, Darstellung und Änderung von THOR-Netz-Modellen.* Die Bedienung des Editors erfolgt dabei interaktiv mit dem Benutzer. Für jedes Unternetz wird ein eigenes Editorfenster bereitgestellt, in dem die Netzelemente direkt über entsprechende Pulldown- und Popup-Menüs oder Tastaturkombinationen eingegeben und dargestellt werden. Die einzelnen Netzelemente werden anschließend in eigenen Editoren beschriftet. Zur Eingabe von Klassenbeschreibungen für komplexe Objekttypen und anderem C++-Code werden dem Benutzer verschiedene Texteditoren angeboten und für die Erstellung hierarchischer Netze erlaubt der Editor sowohl den Top-Down- als auch den Bottom-Up-Entwurf. Zur Illustration zeigt Abbildung 4.2 den Editor für das Haupt- und Unternetz eines Beispielnetzes zusammen mit einem Transitionseditor.

- *Darstellung der Netzhierarchie.* Mit Hilfe eines Design-Graphen, der die Aufruf-Beziehungen zwischen den Teilnetzen eines Netzmodells darstellt, kann der Benutzer zwischen den einzelnen Unternetzen navigieren. Der Design-Graph ist von seiner Konzeption und Funktionalität mit der Page Hierarchy von CP-Netzen vergleichbar (vgl. [HJS90]).

- *Eingabe, Darstellung und Modifikation von Zuständen.* Der Zustand eines THOR-Netzes besteht im wesentlichen aus der Markierung der Stellen mit Objekten, den existierenden Unternetzen und dem aktuellen Zeitpunkt. Die Objekte werden in den jeweiligen Stelleneditoren in ihrer externen Repräsentation eingeben und dargestellt. Zur Bearbeitung hierarchischer Zustände bietet die Entwicklungsumgebung weitere Hilfsmittel an. Dazu gehört ein State-Tree, mit dem der Benutzer durch die Aufrufgeschichte eines Zustands navigieren kann, und ein weiterer Editor zur Darstellung und Änderung von Markierungen in einem Unternetz. Im Gegensatz zum Design-Graphen ist der State-Tree baumartig aufgebaut – ansonsten bietet er eine ähnliche Funktionalität. Jeder Zustand kann unter einem Namen (z. B. `start`) in eine Datei gespeichert werden.

- *Konsistenzprüfungen* werden vom Editor während des gesamten Erstellungsprozesses eines Netzmodells durchgeführt. Dabei stellt der Editor u. a. sicher, daß Stellen und Transitionen nicht mit unerlaubten Kanten verbunden werden und daß die Schnittstellen zwischen verfeinerten Transitionen und ihren Unternetzen zueinander passen. Die vom Benutzer eingegebenen Netzbeschriftungen, insbesondere der C++-Anteil, werden jedoch nicht vom Editor auf syntaktische Korrektheit geprüft. Entsprechende Fehler werden somit erst beim Compilieren entdeckt.

- *Eingabe von Simulationsoptionen.* Vor dem Start einer Simulation müssen in einem speziellen Fenster neben der Art der Simulation (sequentiell oder verteilt) weitere Simulationsparameter wie etwa der Startzustand, die maximale Simulationszeit oder die maximale Anzahl zu schaltender Transitionen festgelegt werden.

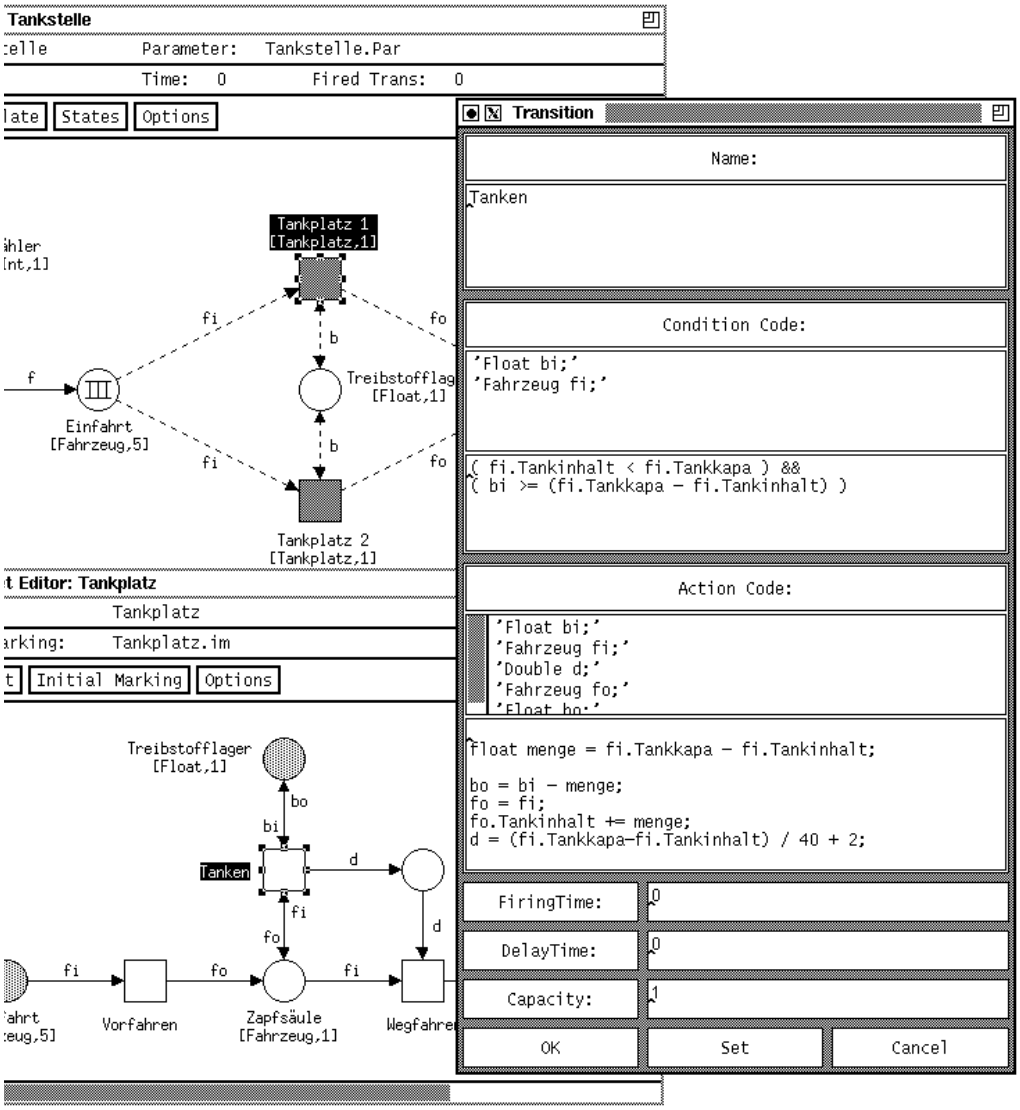


Abbildung 4.2: Verschiedene Editorfenster der Entwicklungsumgebung

- *Aufruf des Compilers und der Simulatoren.* Sowohl die Übersetzung als auch die Simulation von Netzmodellen wird direkt aus dem Editor heraus gesteuert. Der erreichte Endzustand kann im Anschluß an eine Simulation wieder im Editor dargestellt werden (s.o.).

Eine detaillierte Beschreibung der einzelnen Funktionen der graphischen Benutzungsoberfläche und ihrer Bedienung ist im Handbuch zur THORN Entwicklungsumgebung [Wie96a] angegeben.

Implementiert wurde der Editor in C++ unter Verwendung der Graphik-Bibliothek des X Window Systems (X11 Release 5), dem X Toolkit und dem Athena Widget Set [Mau95]. Er läuft auf Rechnern mit einem UNIX-Betriebssystem, einer X11-Oberfläche und einer 3-Tasten-Maus; getestete Plattformen sind Sun Computer mit SunOS 4.1.3 und PCs mit Linux 4.2. Grundlage der Implementierung ist eine C++-Klassenbibliothek [Sei93],

die zur Erstellung von Editoren für einfachere Petrinetz-Klassen im Rahmen des Projekts MOBY [FL93] an der Universität Oldenburg entwickelt wurde. Diese Klassenbibliothek basiert auf dem Model-View-Controler Konzept von Smalltalk80 und stellt neben Petrinetz-spezifischen Klassen nützliche Klassen für zahlreiche Datenstrukturen bereit.

Um das Editorsystem für THOR-Netze auch unter Windows NT auf PCs zugänglich zu machen, wurde eine weitere Version des Editors in Visual C++ mit dem Microsoft Developer Kit implementiert [Mie97]. Abbildung 4.3 zeigt das Hauptfenster dieses Editors.

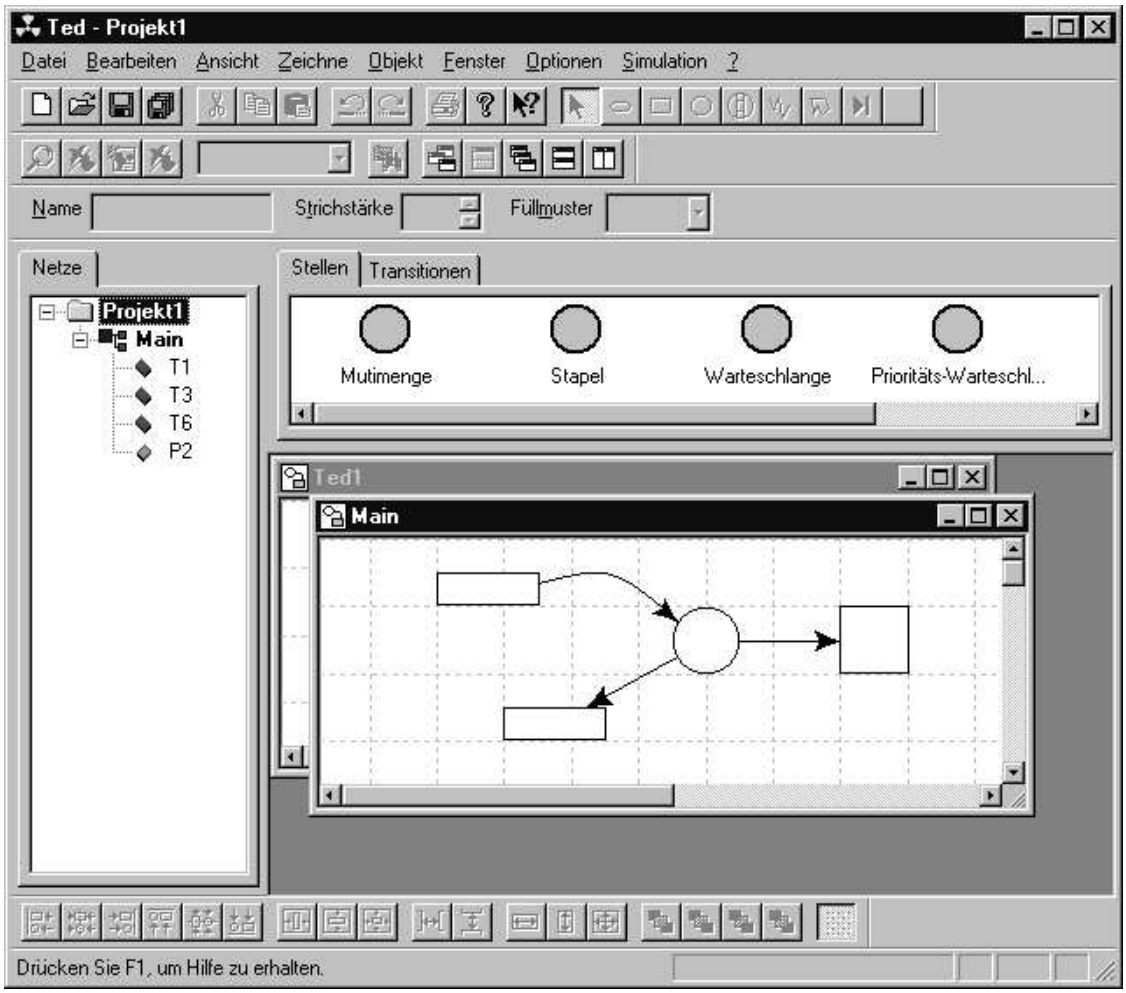


Abbildung 4.3: THOR-Netzeditor für Windows

Der Windows-Editor für THOR-Netze unterscheidet sich von seinem X Windows Vorgänger im wesentlichen durch ein völlig anderes „Look and Feel“ (siehe Abbildung 4.2 und 4.3) und ein etwas modifiziertes Konzept zur Erstellung hierarchischer Netze, da er zwischenzeitlich inkonsistente Netzbeschreibungen zuläßt. Des weiteren sind in diesem Editor verschiedene Windows-spezifische Konzepte intergriert. So verfügt er beispielsweise über eine OLE-Schnittstelle, über die THOR-Netze in andere Dokumente importiert werden können.

4.3 Der Netzcompiler

Die Aufgabe des Netzcompilers besteht im wesentlichen darin, die externe Darstellung eines THOR-Netzes, wie sie vom Editor erzeugt wird, in eine ausführbare Darstellung zu transformieren. Je nachdem, welcher Simulator verwendet werden soll, wird dabei unterschiedlicher Code generiert (vgl. Abb. 4.1).

Für den sequentiellen Simulator wird ein Netz komplett in C++-Code transformiert. Für jedes (Unter-)Netz und jede Transition wird eine eigene Klasse generiert. Für Transitionen werden dabei jeweils zwei Methoden erzeugt, in die die Überprüfung der Aktiviertheit und das Schalten einer Transition fest codiert ist. Die vom Benutzer angegebene C++-Beschriftung wird dabei unverändert übernommen. Eine detaillierte Beschreibung des generierten C++-Codes ist in [Mie96] angegeben. Der vom Netzcompiler erzeugte Code kann anschließend von einem C++-Compiler übersetzt und mit dem Simulatorkern zu einem ausführbaren Programm gebunden werden.

Für den verteilten Simulator wird weniger netzspezifischer C++-Code erzeugt als für den sequentiellen Simulator. Die Netzstruktur wird zusammen mit der Anfangsmarkierung in eine Zwischendarstellung – die sogenannte ANR-Darstellung (ASCII Net Representation) – transformiert, die es ermöglicht, (Unter-)Netze zwischen verschiedenen Prozessoren effizient auszutauschen. Der vom Benutzer angegebene C++-Code (für Objekttypen, Transitionsbeschriftungen etc.) wird wiederum direkt übernommen und so gekapselt, daß der Simulator ihn verwenden kann. Anschließend wird er zum Simulatorkern dazugebunden. Weitere Details zum erzeugten Code für den verteilten Simulator sind in [SWK+94] und [Twe95a] beschrieben. Da beim verteilten Simulator der Zustand mit in der ANR-Darstellung enthalten ist, müssen die Zustandsinformationen am Ende einer Simulation wieder aus dieser Darstellung extrahiert werden, um sie für den Editor lesbar zu machen. Diese Aufgabe wird ebenfalls vom Netzcompiler übernommen.

Der Netzcompiler stellt bei der Übersetzung eines Netzes nur die Fehler fest, die aus einer inkonsistenten bzw. inkorrekten Beschriftung der Netzelemente resultieren. Viele dieser Fehler werden schon durch den Editor abgefangen, so daß der Compiler hier nur eine zusätzliche Überwachungsfunktion hat. Fehlerhafter C++-Code wird vom Netzcompiler nicht entdeckt, da der Code ohne Analyse direkt übernommen wird. Fehler im C++-Code werden also erst vom C++-Compiler bemängelt, der den generierten Code übersetzt und bindet.

Eine ausführliche Beschreibung des Aufbaus und der Funktionsweise des Compilers ist in [Twe95b] angegeben. Implementiert wurde der Netzcompiler ebenfalls in C++.

4.4 Der sequentielle Simulator

Der sequentielle Simulator wird im wesentlichen zur Validierung von Netzmodellen und zu Laufzeitvergleichen mit dem verteilten Simulator eingesetzt. Das vorrangige Entwicklungsziel bestand in einer möglichst effizienten Implementierung eines Simulators auf einem Prozessor. Damit kann die Laufzeit des verteilten Simulators auf n Prozessoren mit

der Laufzeit eines optimierten sequentiellen Simulators verglichen werden, anstatt diesen Vergleich mit der 1-Prozessor-Version des verteilten Simulators durchzuführen.

Die grundlegende Idee zur Optimierung der Laufzeiteffizienz des Simulators besteht in der Übersetzung eines kompletten Netzmodells in ein ausführbares Programm (s. a. Abschnitt 4.3). Für jede Transition in einem Netz wird dabei eine eigene C++-Klasse erzeugt, die im wesentlichen die Methoden `Check` und `FireEnd` beinhaltet. Diese Methoden führen entsprechend der Schaltregel für THOR-Netze (vgl. Abschnitt 3.2.1 und 3.3.1) die Überprüfung auf Aktiviertheit, den Schaltbeginn und das Schaltende einer Transition aus. Dabei sind die ersten beiden Aktionen in der Methode `Check` kodiert. Netze werden mit ihren Komponenten (Stellen und Transitionen) ebenfalls in einer eigenen Klasse gespeichert. Kanten werden entsprechend ihrer Funktion für die Objektübergabe an und von Transitionen in den Code der Methoden `Check` und `FireEnd` integriert.

Die vom Compiler generierten Klassen und Methoden werden mit dem Simulatorkern zu einem ausführbaren sequentiellen Simulator zusammengebunden (vgl. Abschnitt 4.1). Der Simulatorkern besteht im wesentlichen aus einer globalen Ereignisschlange, die durch einen Scheduler verwaltet wird. Die Ereignisschlange ist eine nach Zeitpunkten sortierte Liste (vgl. Abbildung 4.4). Jeder Eintrag der Liste kann für einen diskreten Zeitpunkt Verweise auf Transitionen enthalten, die zu diesem Zeitpunkt einen Schaltvorgang beenden oder zu diesem Zeitpunkt möglicherweise aktiviert werden.

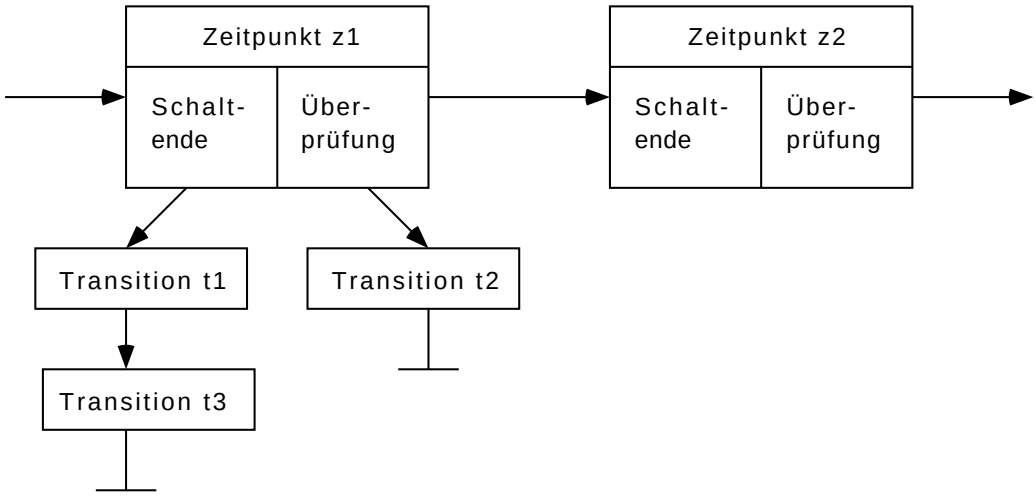


Abbildung 4.4: Ereignisschlange

Der Scheduler arbeitet die Ereignisschlange unter Berücksichtigung der Schaltregel für THOR-Netze ab (vgl. Abschnitt 3.2.1). Etwas vereinfacht werden dazu folgende Aktionen durchgeführt:

1. Falls die Ereignisschlange leer ist oder ein Abbruchkriterium der Simulation (max. Schrittzahl, max. Simulationszeit) erreicht wurde, bricht die Ausführung ab.

2. Existieren in dem ersten Eintrag der Ereignisschlange Transitionen, die einen Schalt-

vorgang beenden oder möglicherweise aktiviert werden, so wird eine dieser Transitionen zufällig ausgewählt:

(a) Bei einer Überprüfung einer Transition auf Aktiviertheit wird zunächst eine Belegung der eingehenden Kantenanschriften bestimmt, so daß die Vorbereichsbedingung der Transition erfüllt ist.
Für dieses Schaltereignis wird – unter Berücksichtigung der Verzögerungszeit und der Zeitstempel der gebundenen Objekte – festgestellt, ob es vollständig verzögert ist.
Falls das Schalterergnis aktiviert ist (vgl. Abschnitt 3.2.1), beginnt es direkt zu schalten:

i. Die Schaltaktion wird ausgeführt und die Schaltdauer berechnet.
ii. Die produzierten Objekte werden mit dem Zeitstempel ihrer Entstehung (aktuelle Simulationszeit + Schaltdauer) auf den Nachbereichsstellen eingefügt.
iii. Ist die Schaltdauer des Schaltereignisses größer Null, so wird zusätzlich Platz auf den Nachbereichsstellen reserviert, die Schaltkapazität der Transition dekrementiert und ein entsprechender Schaltende-Eintrag in die globale Ereignisschlange zum Zeitpunkt des Schaltendes einfügt.
iv. Die konsumierten Objekte werden von den Vorbereichsstellen entfernt und für alle Transitionen, deren Aktiviertheit durch die gelöschten Objekte beeinflußt wird, wird ein Eintrag in die Überprüfungsliste des aktuellen Zeitpunkts eingefügt.

Falls das Schalterergnis noch nicht vollständig verzögert ist, erhält die Transition in der Ereignisschlange einen Eintrag in die Überprüfungsliste des Zeitpunkts des potentiellen Verzögerungsendes.

(b) Bei der Bearbeitung des Schaltendes einer Transition wird der beim Schaltbeginn reservierte Platz auf den Nachbereichsstellen freigegeben und die Schaltkapazität der Transition wieder inkrementiert.
Des weiteren erhalten alle Transitionen, deren Aktiviertheit durch die neu produzierten Objekte beeinflußt wird, einen Eintrag in die Überprüfungsliste des aktuellen Zeitpunkts.

3. Ist der erste Eintrag der Ereignisschlange vollständig abgearbeitet, wird er gelöscht.

4. Fahre mit Punkt 1 fort.

Die Aktionen in Punkt 2 a) und 2 b) dieses abstrakten Algorithmus werden durch die vom Compiler generierten `Check`- bzw. `FireEnd`-Methoden einer Transition durchgeführt. Die Bearbeitung der Ereignisschlange und die Kontrolle der Abbruchkriterien sowie die Auswahl von Transitionen und der Aufruf der jeweiligen Methoden erfolgt durch den Simulatorkern.

Zu Beginn der Simulation enthält die Ereignisschlange nur einen Eintrag mit einer leeren Schaltende-Liste und eine Überprüfungsliste, in der jede Transition des Netzes einen Eintrag hat.

Weitere Einzelheiten zur Implementierung des sequentiellen Simulators werden in [Mie96] erläutert. An dieser Stelle sollen jedoch noch zwei Konzepte etwas näher betrachet werden, die wesentlich zu einer effizienten sequentiellen Simulation beitragen und in dem obigen Algorithmus nur angedeutet werden.

- Der sequentielle Simulator verwendet bei der Bestimmung von aktivierten Schaltelementen eine Strategie, bei der nicht alle Kombinationsmöglichkeiten einer Transitionsbelegung getestet werden müssen, sondern die erste passende Belegung geschaltet wird. Diese sogenannte *Direkt-* bzw. *Sofortschaltregel* kann im Fall von nicht verzögerten Transitionen mit Multiset-Stellen im Vorbereich zu einer erheblichen Beschleunigung führen (vgl. [Zie95]). Außerdem kann durch diese Strategie sehr viel Speicherplatz gespart werden, da nicht alle Schaltereignisse zu einer Transition gespeichert werden müssen.

- Eine weitere effizienzverbessernde Maßnahme ist die Berücksichtigung des Einflußbereichs von schaltenden Transitionen bei der Überprüfung der Aktiviertheit anderer Transition. In der Kernschleife des Simulationsalgorithmus werden nach dem Schalten einer Transition nicht sämtliche Transitionen auf eine mögliche Aktivierung geprüft, sondern nur solche, die potentiell aktiviert werden.
  Dazu wird zur Compile-Zeit für jede Transition der Einflußbereich bestimmt und bei der Erzeugung der `Check`- bzw. `FireEnd`-Methoden berücksichtigt. Der Einflußbereich einer Transition teilt sich in einen Bereich, der durch den Schaltbeginn beeinflußt wird, und einen weiteren, der durch das Schaltende beeinflußt wird. Abbildung 4.5 stellt beispielhaft den Einflußbereich einer Transition dar.

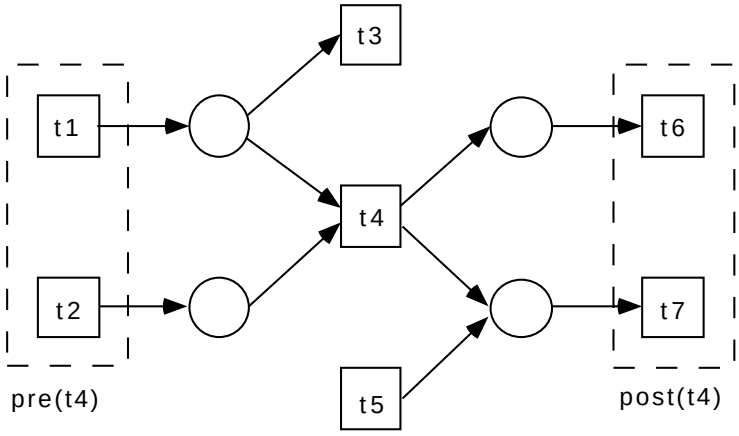


Abbildung 4.5: Einflußbereiche einer Transition

Der *Vor-Einflußbereich* zu einer Transition t (`pre(t)`) umfaßt alle Transitionen, deren Nachbereich sich mit dem Vorbereich von t überschneidet und für die gilt: Mindestens eine Stelle im Schnitt hat beschränkte Kapazität und t konsumiert Objekte von dieser Stelle. Durch den Schaltbeginn von t werden Plätze auf diesen Stellen frei, so daß Transitionen aktiviert werden können, die diese Stelle im Nachbereich haben und vorher aufgrund fehlenden Platzes im Nachbereich nicht schalten konnten.

Der *Nach-Einflußbereich* zu t (`post (t)`) umfaßt alle Transitionen, deren Vorbereich mit dem Nachbereich von t einen nichtleeren Schnitt bildet. Das Ablegen von Objekten auf einer Stelle kann Transitionen, die diese Stelle im Vorbereich haben, aktivieren.

Eine detaillierte Beschreibung aller Konzepte des sequentiellen Simulators ist in [SWK+94] angegeben.

Bei der Ausführung von THOR-Netz-Modellen aus praktischen Anwendungen erreicht der sequentielle Simulator auf einer Sun SPARCstation 10/40 eine durchschnittliche Schaltrate von 1.500 gefeuerten Transitionen pro Sekunde – bei trivialen Netzen wurden maximal 2.500 Schaltungen pro Sekunde gemessen. Die Schaltrate kann allerdings erheblich niedriger ausfallen, wenn es in einem Netz sehr viele Kombinationsmöglichkeiten für eine Transitionsbelegung gibt. In den durchgeführten Fallstudien war dies jedoch nicht der Fall.

Kapitel 5

Anwendungsbericht

Dieses Kapitel beinhaltet den Erfahrungsbericht einer umfangreichen Anwendung von THORNs, die am Lehrstuhl für Informatik I der RWTH Aachen vorgenommen wurde. Autor der folgenden Ausführung ist Helmut Boll, der als wissenschaftlicher Mitarbeiter an eben diesem Lehrstuhl beschäftigt ist.

5.1 Was wird modelliert?

In einem mittelständischen Luftfahrtgeräteunternehmen sollen mit genetischen Algorithmen erzeugte Produktionspläne für den Reparatur- und Wartungsbereich (R/W-System) simulativ auf ihre Fitness geprüft werden. Wichtigstes Ziel ist die Verkürzung der Durchlaufzeiten von Reparatur- und Wartungsaufträgen. Zu diesem Zweck werden drei Komponenten: *R/W-System*, *Umplanung/Steuerung* und *Planung* nachgebildet werden.

5.1.1 R/W-Systemkomponente

Im untersuchten Reparatur- und Wartungssystem werden Triebwerksteile (Aggregate) von öffentlichen und privaten Kunden zur Wartung und/oder Reparatur angeliefert. Die Aggregate werden am Wareneingang abgegeben und erhalten eine Auftrags-ID sowie einen Eingangsstatus. Zu diesem Zeitpunkt gibt der Kunde eine Leistungsart (spezifizierte Reparatur, Wartung, Instandsetzung, Grundüberholung) an, die seiner Meinung nach an dem Aggregat verrichtet werden muß. Nach Anlegen der Kundenstammdaten wird das Aggregat unmittelbar in den R/W-Bereich geleitet. Dort werden die Werker mit den entsprechenden Qualifikationen für die an dem Aggregat zu verrichtenden Arbeitsgänge ausgewählt. Dann beginnt die Demontage mit anschließender Begutachtung durch den entsprechenden Werker. Die Befundung kann zu einer neuen Leistungsart führen, was durch Rücksprache mit dem Kunden abgeklärt werden muß.

Nach der Begutachtung werden die verschlissenen Teile ausgetauscht. Gleichzeitig werden entsprechend neue Teile vom Lager angefordert, wobei in speziellen Fällen statt der Originalteile auch Alternativteile verwendet werden können. In gewissen Fällen müssen

aufgrund rechtlicher und technologischer Bedingungen in Kombination mit einem verschlissenen Teil auch bestimmte andere Teile unabhängig von ihrer Funktionsfähigkeit ausgetauscht werden.

Das demontierte Aggregat geht in Form der verbliebenen intakten Teile an die Montage weiter, wo es zusammen mit den angeforderten Neuteilen montiert wird.

An die Montage schließt sich ein Prüfvorgang an. Zu diesem Zweck wird das Aggregat in eine Prüfmaschine eingespannt, mit deren Hilfe es möglich ist, die einzelnen Funktionen des Aggregats zu testen. Auch dieser Prüfvorgang muß wieder von entsprechend qualifizierten Werkern durchgeführt werden. Fällt die Prüfung positiv aus, wird das Aggregat verplombt und verdrahtet und dem Warenausgang zugestellt. Bei negativem Prüfergebnis gelangt das Aggregat erneut in die Demontage und durchläuft den oben beschriebenen Prozeß erneut.

5.1.2 Planungskomponente

Aus der Beschreibung der R/W-Systemkomponente geht hervor, daß neben dem klassischen Planungs- und Steuerungsproblem der Simultanplanung bei mehrfacher Zielsetzung die Probleme

- Werkerqualifikation
- Befundungssystem
- Verschleißstücklisten
- Kombinationsteile
- Alternativteile

zu lösen sind.

Die Planung im R/W-System besteht aus einer Menge von konventionellen Verfahren zu allen Aufgabenbereichen der Produktionsplanung sowie einem Genetischen Algorithmus, der übergreifend die Planungsverfahren zur Kooperation führt.

Der Genetische Algorithmus erlaubt die Generierung einer Population von Produktionsplänen, die in zyklischer Wechselwirkung mit der Komponente R/W-System anhand ihrer Planwirkungen hinsichtlich der Ausprägung der Multiziele Fitnesswerte erhalten, die dann die nächsten Suchschritte in Richtung besserer Produktionspläne steuern.

5.1.3 Umplanung/Steuerungskomponente

Die Umplanungs- und Steuerungskomponente dient zur Behandlung von Ereignissen, die durch unerwünschte Planwirkungen ausgelöst werden.

5.2 Entwicklung einer Datenbankschnittstelle

Um mit THOR-Netzen große, bereits gespeicherte Datenmengen bearbeiten zu können, wurden die THOR-Netze auf PC- und Workstationebene um eine SQL-orientierte Datenbankschnittstelle erweitert.

Auf dem PC gibt es innerhalb der THOR-Netze zwei Möglichkeiten (Abb. 5.1), um auf eine Datenbank zuzugreifen. Die erste Möglichkeit erlaubt einen Datenbankzugriff aus einer Transition heraus. Dies wird durch die Microsoft Foundation Class Library (MFC) ermöglicht. Die Schnittstelle bietet eingebettete SQL-ähnliche Abfrage- und Änderungsbefehle für eine offene Datenbankschnittstelle an. Mittels eines Cursors kann eine Datenbankrelation Satz für Satz durchlaufen werden. Eine Kopie des Datensatzes, auf den der Cursor aktuell zeigt, kann innerhalb des Transitionscodes mit gewöhnlichen C++-Befehlen bearbeitet und gegebenenfalls in die Datenbank zurückgeschrieben werden.

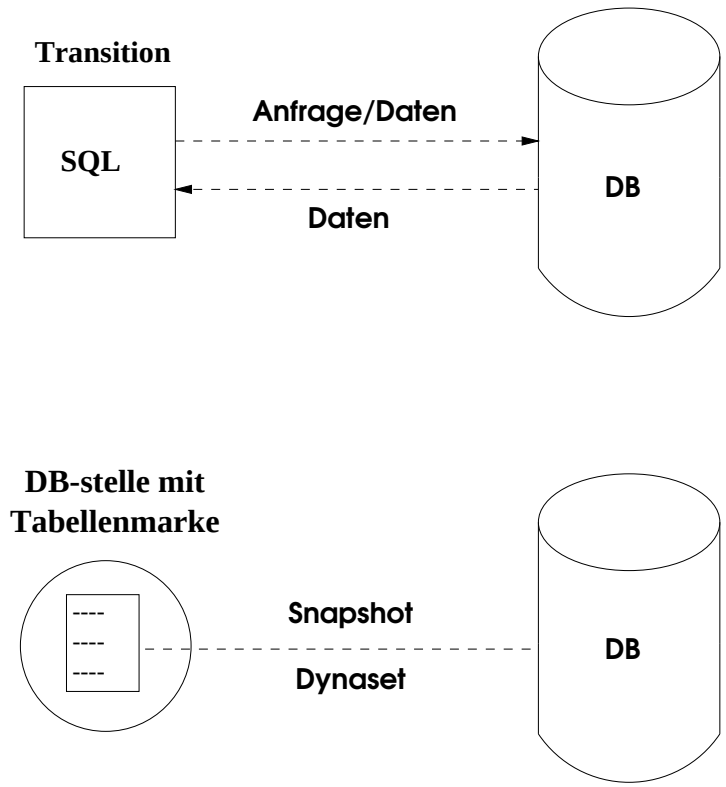


Abbildung 5.1: Datenbankzugriff

Die zweite Möglichkeit besteht in der Repräsentation einer kompletten Datenbankrelation inklusive eines Cursors, der auf einen Datensatz der Relation zeigt, als Tabellenmarke. Die Tabellenmarke ist eine THOR-Netzmarke und kann an Transitionen mit den von ihr angebotenen Klassenmethoden (SQL-Befehle) manipuliert werden. Die korrespondierende Datenbanktabelle wird bei dieser Zugriffsmöglichkeit entweder als Snapshot oder als Dynaset repräsentiert.

Beide Zugriffsmöglichkeiten werden über eine Open Database Connectivity Schnittstel-

le (ODBC-Schnittstelle) realisiert, wodurch ein THOR-Netz auf unterschiedliche Datenbanksysteme zugreifen kann. Die ODBC-Architektur stellt eine Applikationsebene zur Verfügung, an der die in der Transition verwendeten Datenbankzugriffsbefehle angeboten werden. Durch die MFC-Befehle wird die Programmierung der teilweise sehr komplexen ODBC-Schnittstelle vereinfacht.

Der Treiber-Manager wählt die richtigen Treiber für die Applikation aus. Der ausgewählte Treiber bildet dann die Befehle der Applikation auf die Syntax der ausgewählten Datenbank ab und überträgt sie anschließend. In der umgekehrten Richtung überträgt er die Anfrageergebnisse von der Datenquelle zur Applikation. Dabei setzt sich die Datenquelle aus einer Datenbank und einem Datenbankmanagementsystem, wie z.B. MS-ACCESS oder ORACLE zusammen.

Auf Workstationebene wird nur die erste Repräsentationsart von Datenbanktabellen unterstützt. Ferner ist die Datenbankanknüpfung konkret an eine relationale objektorientierte Postgres-Datenbank gebunden. Es handelt sich also um keine offene Schnittstelle.

Die Einrichtung und Nutzung der Datenbankschnittstelle bringt folgende Vorteile:

- Zugriff auf große Datenmengen, wie beispielsweise Stücklisten oder Simulationsdaten
- bereits vorhandene Datenstrukturen können genutzt werden
- Datenbankfunktionalität, wie beispielsweise Datensuche, Bildung von Sichten oder Wahrung der Datenintegrität sind im THOR-Netz ohne zusätzlichen Aufwand verfügbar

5.3 Modellierung der Komponenten

Alle Phasen der Modellierung der Komponenten *R/W-System*, *Umplanung/Steuerung* und *Planung*, also vom konzeptuellen Modell bis zum Computerprogramm, werden durchgängig mit THOR-Netzen beschrieben bzw. generiert.

5.3.1 Modellierung der R/W-Systemkomponente

Auf der obersten konzeptuellen Ebene befinden sich die drei Komponenten *R/W-Sytem*, *Planung* und *Umplanung/Steuerung*. Alle drei Komponenten sind als Aufruftransitionen modelliert und werden durch Einbettungsstellen miteinander verbunden. Zu diesem Zeitpunkt besteht nur Klarheit darüber, daß die beiden Komponenten logisch voneinander getrennt sein sollen. Über den konkreten Inhalt der Aufruftransitionen sowie die Schnittstellen gibt es in dieser Modellierungsphase noch keine konkreten Vorstellungen.

Die Aufgabe der Aufruftransition `R/W-System` besteht in der Nachbildung der Arbeitsabläufe des R/W-Systems. Da Rückschlüsse auf das reale System nur dann sinnvoll sind, wenn die Simulation auf einem akutellen Ist-Zustand aufsetzt, ist eine Initialisierung der Simulation mit dem momentanen Zustand des realen R/W-Systems nötig.

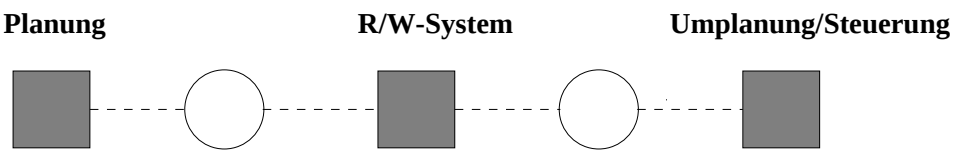


Abbildung 5.2: Hauptnetz

Da aufgrund früherer Tests bereits ein THOR-Netz zur Initialisierung einer Testumgebung existiert, wird dieses THOR-Netz nach gewissen Modifikationen in die R/W-Systemtransition importiert. Dieser Vorgang wird durch die THOR-Netzentwicklungsumgebung unterstüzt und kann damit schnell durchgeführt werden.

Als zweite Komponente wird die Aufruftransition `Simulation` kreiert (Abb. 5.4). Sie soll in der Anzahl der Maschinen und Werker konfigurierbar sein. Daher werden Maschinen und Werker durch Marken (Objekte) und Stellen modelliert. Mit der THOR-Netzentwicklungsumgebung können auf einfache Weise die entsprechenden Netzstrukturen erzeugt werden, ohne daß die Stellentypen der inzidenten Stellen in allen Details bereits festliegen müssen. Dadurch kann der Modellierer seinen Fokus wahlweise auf die Nachbildung der Prozesse oder der Simulationsobjekte richten.

Der Arbeitsfluß des R/W-Systems wird durch die Bearbeitung von Aufträgen und deren Störung in Form von Werker- und Maschinenausfällen bestimmt. Entsprechend wird die Aufruftransition Simulation in drei weitere Aufruftransitionen *Werkerausfall*, *Maschinenausfall* und *Bearbeitung* verfeinert (Abb. 5.4).

Die Aufruftransitionen *Werkerausfall* und *Maschinenausfall* werden jeweils über eine gemeinsame Einbettungstelle mit der Bearbeitungstranstion verbunden. Beide Aufruftransitionen erzeugen stochastisch Störereignisse, die über je eine Einbettungsstelle in FIFO-Reihenfolge von der Transition *Bearbeitung* behandelt werden sollen. Dies ist leicht modellierbar, da die THOR-Netzentwicklungsumgebung zum einen eine große Auswahl an Wahrscheinlichkeitsverteilungen bereit hält, die das Schaltverhalten von Transitionen festlegen und zum anderen Stellen in THOR-Netzen mit einer Stack-, Queue- oder Priorityqueuestruktur versehen werden können. Der Modellierer braucht als keine eigenen Listenstrukturen zu entwickeln.

Die Aufruftransition *Bearbeitung* (Abb. 5.5) bildet den Auftragsfluß und die Montage/Demontagevorgänge nach. Der Auftragsfluß wird durch die Netzstruktur modelliert. Da in THOR-Netzen das Schaltverhalten flacher Transitionen durch einen C++-Codeblock spezifiziert werden kann, läßt sich ein komplexer Arbeitsgang durch eine einzelne flache Transition modellieren.

Durch den Freiheitsgrad, Systemteile als Netzstruktur oder als Transtionsbeschriftung zu modellieren, kann die Netzstruktur von geschlossenen Vorgängen, wie Arbeitsgängen, freigehalten werden.

Im Bereich der Materialwirtschaft hat sich die Verwendung inhibitorischer Kanten als ausgezeichnete Unterstützung zur Modellierung von Ereignissen, die bei Unterschreitung von Lagerbeständen im allgemeinen und Sicherheitsbeständen im besonderen ausgelöst

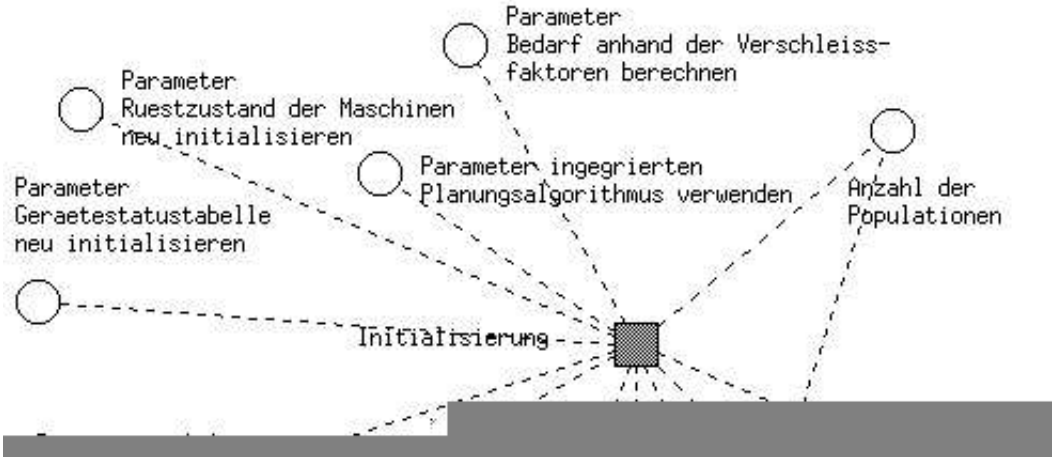


Abbildung 5.3: R/W-System

werden, erwiesen.

Durch attributabhängige Schaltdauern lassen sich störfallbedingte Ressourcenausfälle und Arbeitsgangdauern als Attribute der entsprechenden Objekte modellieren. Durch diese

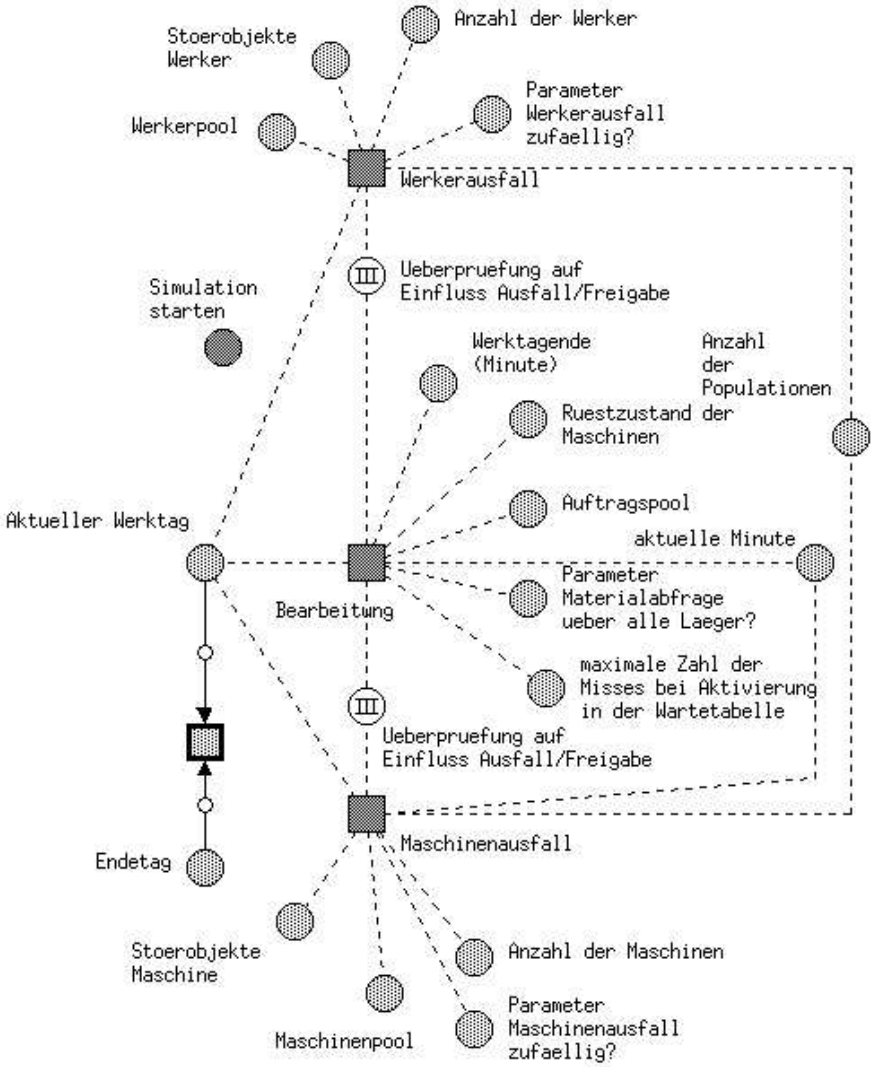


Abbildung 5.4: Simulation

Möglichkeit können beispielsweise Maschinenlaufzeiten für jeden Arbeitsgang individuelle Dauern erhalten, was realistischer ist, als diese Daten aus Ressourcen-Arbeitsgang-Tabellen mit statischen Einträgen auszulesen.

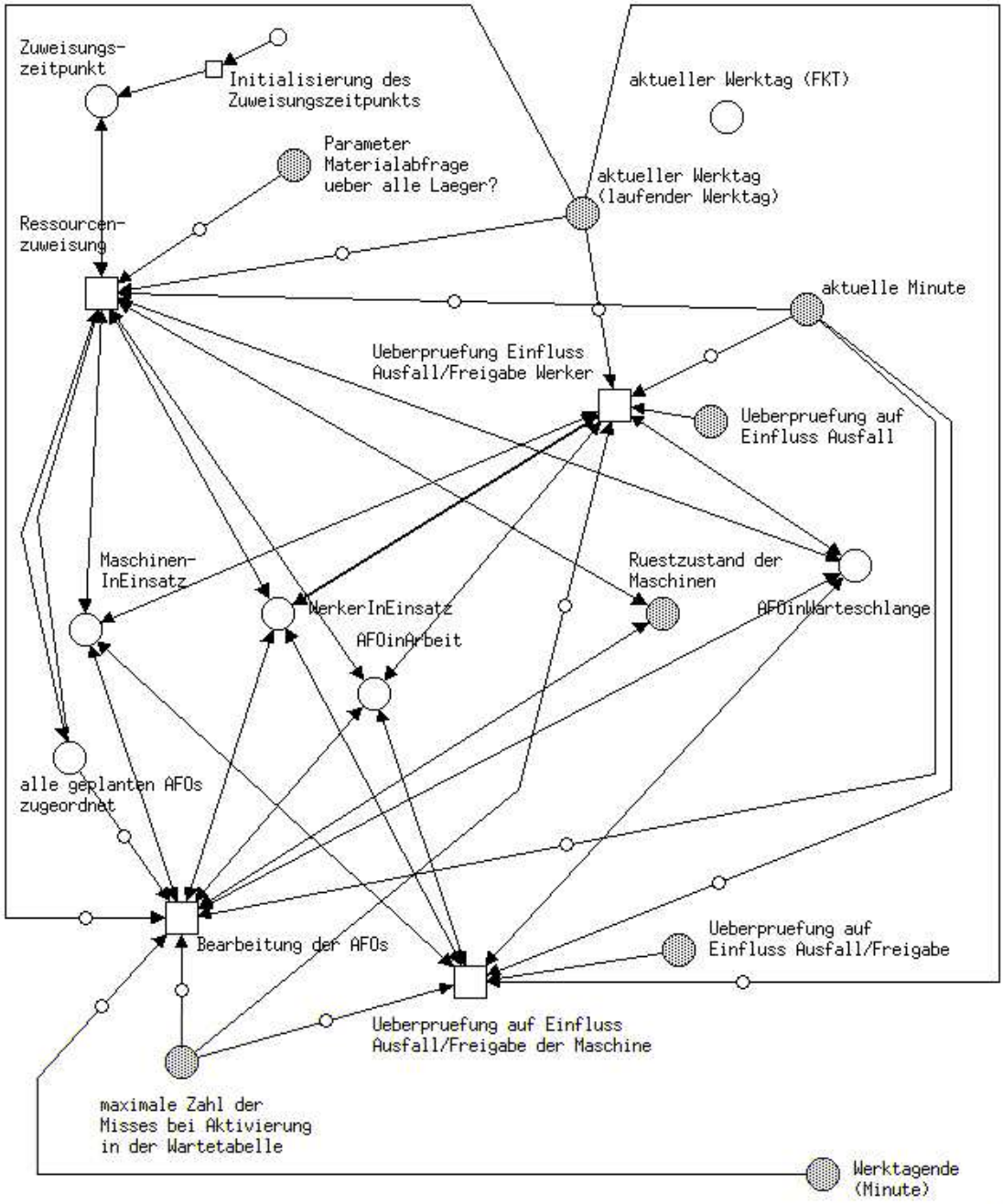


Abbildung 5.5: Bearbeitung

5.3.2 Modellierung der Planungskomponente

Die Planungskomponente ist die zweite Aufruftransition im Hauptnetz. Sie beinhaltet einen Pool von Planungsverfahren, die für bestimmte Planungsaufgaben im R/W-System benutzt werden. Aus diesem Pool wird nach entsprechender Initialisierung der Aufruftransition eine Teilmenge zur Erzeugung eines Produktionsplans für die R/W-Systemkomponente ausgewählt. Jede Planungsfunktion wird durch einen C++-Block spezifiziert, mit dem dann jeweils eine flache Transition beschriftet wird. Die Menge dieser Transitionen

bildet neben einem genetischen Algorithmus die Verfeinerung der Planungstransition.

Jeder Produktionsplan besteht konzeptuell aus einer Menge von Resultaten von Teilplanungen, deren Realisierung eine Folge von Zustandsänderungen im Realsystem bewirkt. Dabei gibt es eine zeitliche Abhängigkeit im Ablauf der einzelnen Teilplanungen.

Die entsprechende Modellierung dieser Zusammenhänge ist durch die Konzeption der THOR-Netze einfach beschreibbar. Die Resultate der Teilplanungen werden in Form von Marken auf Einbettungstellen mit entsprechenden Typen abgelegt. Die Zusammenfassung der Marken aller Teilplanungen stellt den Produktionsplan dar. Die Zusammengehörigkeit der Marken wird, ähnlich wie in relationalen Datenbanken, durch die Vergabe eines eindeutigen Werts eines Attributs, das alle Schnittstellen-Objekttypen besitzen, hergestellt. Jede Menge von Marken mit gleichem Attributwert repräsentiert ein PPS-Objekt. Jedes PPS-Objekt steht aufgrund der Anknüpfung mit Einbettungsstellen in der Aufruftransition *R/W-System* zur Verfügung.

5.4 Modellierungsqualität der THOR-Netze und Ergebnisse

Da Produktionssysteme und speziell Reparatur- und Wartungssysteme hinsichtlich der operativen Planung zu verteilten Systemen mit diskreter Zeit, diskreten Zuständen und festen Ereignisketten gezählt werden können, bieten sich THOR-Netze zur Simulation solcher Systeme an. Durch das Objektkonzept der THOR-Netze können individuelle Objekte wie Teile, Stücklisten, Werkzeuge usw. problemlos als Marken zur Verfügung gestellt werden. Die Spezifikation von Transitionen mit beliebigem C++-Code erlaubt die einfache Integration komplexer Planungsmethoden, wie beispielsweise mehrstufige Losgrößenplanung. Ebenso einfach gestaltet sich die Manipulation komplexer Unternehmensobjekte.

Alle Funktionen werden von der THOR-Netzentwicklungsumgebung unterstützt. Einfache Verwendung von Hotkeys und ein kontext-sensitives Mausmenu erlauben die schnelle Erstellung von umfangreichen Netzen. Etwas umständlich gestaltet sich die Code-Eingabe, die nur wenige Zeilen des Codetextes gleichzeitig anzeigt.

Durch das zweifache Zeitkonzept: Verzögerungszeit und Schaltzeit, ist die Nachbildung von First-Come-First-Served Vorgängen an Maschinen sowie die Nachbildung von Arbeitsgangdauern einfach durchzuführen. Durch die Strukturierung der Stellen in Schlangen mit Prioritäten ist es möglich, unmittelbar eine prioritätsgesteuerte Maschinenbelegungsplanung zu modellieren. Durch die Stellen mit FIFO-Struktur können Warteschlangen vor Ressourcen eins-zu-eins dem Realsystem nachgebildet werden. Durch die Möglichkeit, Transitionen stochastisches Schaltverhalten mitzugeben, können realistische Störereignisse, wie Maschinenausfälle oder Eilaufträge nachgebildet werden.

Mittels des Hierarchiekonzepts kann das konzeptuelle Modell des Reparatur- und Wartungssystems durchgängig mit THOR-Netzen zu einem Computermodell verfeinert werden. Durch Bereitstellen geeigneter Schnittstellen in Form von Stellen im THOR-Netz ist es möglich, gewisse Abschnitte des Netzes parallel zu entwickeln.

Durch die gemeinsame Verwendung von Zeit-, Hierarchie- und Objektkonzept lassen sich mit THOR-Netzen komplexe Objekte sehr einfach in komplexe Ablaufstrukuren integrieren, eine Eigenschaft die viele Modellierungstools vermissen lassen.

5.5 Zukünftige Aktivitäten

Durch die implementierungstechnische Konzeption ist es einfach, die THOR-Netze um eine SQL-orientierte offene Datenbankschnittstelle zu erweitern. Dadurch ist es möglich, aus THOR-Netzen auf große Datenbestände, die in Datenbanken abgelegt sind lesend und schreibend zuzugreifen. Dadurch ergeben sich neue Einsatzmöglichkeiten für THOR-Netze, wie unsere prototypische Implementierung zeigt. Systemstruktur und Systemverhalten können kombiniert im THOR-Netzgraph, der Datenbank und dem Transitionscode modelliert werden. Dadurch erhält der Anwender zusätzliche Freiheitsgrade bei der Modellerstellung.

Durch eine verteilte Ausführung von THOR-Netzen werden rechenintensive Verfahren, wie z.B. Genetische Algorithmen, die in letzter Zeit immer mehr an Bedeutung gewinnen, in vielen Anwendungsbereichen wirksamer einsetzbar. Dadurch wird es möglich, schwierige Probleme, wie die Simultanplanung in produzierenden Unternehmen, durch praktikable Ansätze zu lösen.

Kapitel 6

Verteilte Simulation von THORNs

Um Parallelrechner oder Workstation-Cluster zur Beschleunigung von Simulationsanwendungen einzusetzen, muß das Modell geeignet in sogenannte *logische Prozesse* (*LPs*) partitioniert werden, die ereignisgesteuert arbeiten. Die einzelnen LPs müssen auf die zur Verfügung stehenden Prozessoren verteilt werden. Natürlich muß gewährleistet werden, daß das Ergebnis der verteilten Simulation korrekt ist, d. h. dem durch eine entsprechende sequentielle Simulation erreichbaren entspricht. Eine hinreichende Bedingung hierfür ist, daß jeder LP seine Ereignisse in aufsteigender Reihenfolge ihrer Zeitstempel bearbeitet (*local causality constraint* [Fuj90, S. 32]).

Zur Gewährleistung dieser Bedingung wurden verschiedenste Verfahren entwickelt. Für die Simulation zeitbeschrifteter Petrinetze sind hierbei besonders asynchrone Verfahren geeignet. Diese lassen sich in zwei verschiedene Zweige unterscheiden. Zum einen werden *konservative Verfahren* verwendet, bei denen Bearbeitungsreihenfolgefehler a priori vermieden werden, indem die LPs blockieren, falls nicht gewährleistet ist, daß keine Ereignisnachricht mit kleinerem Zeitstempel als das gerade zur Bearbeitung anstehende eintreffen wird. Bei *optimistischen Verfahren* bearbeiten LPs sofort ihre Ereignisse, wenn ihnen freie Rechenzeit zur Verfügung steht. Die auf diese Weise möglicherweise auftretenden Bearbeitungsreihenfolgefehler werden jedoch später erkannt und behoben.

Zur Simulation von THORNs sind konservative Simulationsverfahren i. allg. nicht besonders gut geeignet, da sie zum Erreichen hoher Beschleunigungswerte eine gute Vorausschau auf das zukünftige Verhalten der einzelnen Modellteile benötigen. Anders als in einfachen Petrinetzklassen besteht in THORNs die Möglichkeit, Zeitbeschriftungen durch evtl. objektabhängige C++-Funktionen anzugeben. Hierdurch ist das Gewinnen einer hinreichend großen Vorausschau nicht in jedem Fall möglich. Da jedoch für bestimmte Unterklassen von THORNs (etwa mit einem eingeschränkten Zeitkonzept) auch eine effiziente konservative Simulation möglich sein könnte, wurde diese im Rahmen einer Diplomarbeit untersucht.

Hauptsächlich wurde jedoch die optimistische Simulation von THOR-Netzen betrachtet. Aus diesem Grund soll nun zunächst das eingesetzte optimistische Time-Warp-Verfahren mit seinen wichtigsten Varianten näher erläutert werden und einige der in diesem Projekt durchgeführten Untersuchungen zu Datenstrukturen und Algorithmen bei der optimistischen Simulation vorgestellt werden. Danach wird die Adaption des Time-Warp-

Verfahrens für die verteilte THORN-Simulation genauer beschrieben. Schließlich wird das Laufzeitverhalten des verteilten optimistischen THORN-Simulators durch experimentelle Resultate dokumentiert.

6.1 Das Time-Warp-Verfahren

Im Time-Warp-Verfahren bearbeitet jeder LP sofort das nächste ihm bekannte Ereignis (d. h. das Ereignis mit dem kleinsten Zeitstempel), wenn ihm freie Rechenzeit zur Verfügung steht. Hierdurch ist jedoch prinzipiell die Möglichkeit gegeben, daß das Ursache-Wirkungs-Prinzip verletzt wird, d. h. die optimistische Annahme eines LPs, daß ihm kein Ereignis mit einem kleineren Zeitstempel als der des augenblicklich bekannten nächsten Ereignisses mehr eingeplant wird, kann sich als falsch erweisen. In diesem Fall sind alle Berechnungen, die durch die zu frühe Bearbeitung von Ereignissen verursacht wurden, umsonst gewesen: die Berechnung aller zu früh ausgeführten Ereignisse muß zurückgenommen werden — ein sogenannter *Rollback* in der Zeit findet statt.

Jeder LP hat seine eigene lokale Uhr, die den Zeitstempel des zuletzt bearbeiteten Ereignisses enthält und auf diese Weise den Fortschritt des LPs in der Simulationszeit anzeigt. Der Wert dieser Uhr wird als *lokale virtuelle Zeit* (LVT) des LPs bezeichnet. Mit dem bereits gesagten folgt, daß ein Rollback stattfindet, wenn eine Ereignisnachricht mit einem kleineren Zeitstempel als die LVT einen LP erreicht; eine solche Nachricht wird auch als *Nachzüglernachricht* bezeichnet.

Um einen Rollback durchführen zu können, muß ein LP regelmäßig Sicherungen seines lokalen Zustandes in der sogenannten *State-Queue* durchführen und sich außerdem versendete Nachrichten in der *Output-Queue* und empfangene Nachrichten in der *Input-Queue* merken (siehe Abb. 6.1). Das Durchführen eines Rollbacks besteht dann darin, einen korrekten Zustand durch Rücksichern eines der gesicherten Zustände wiederherzustellen (und zwar durch einen Zustand, der mit einer Zeit assoziiert ist, die kleiner oder gleich dem Zeit-

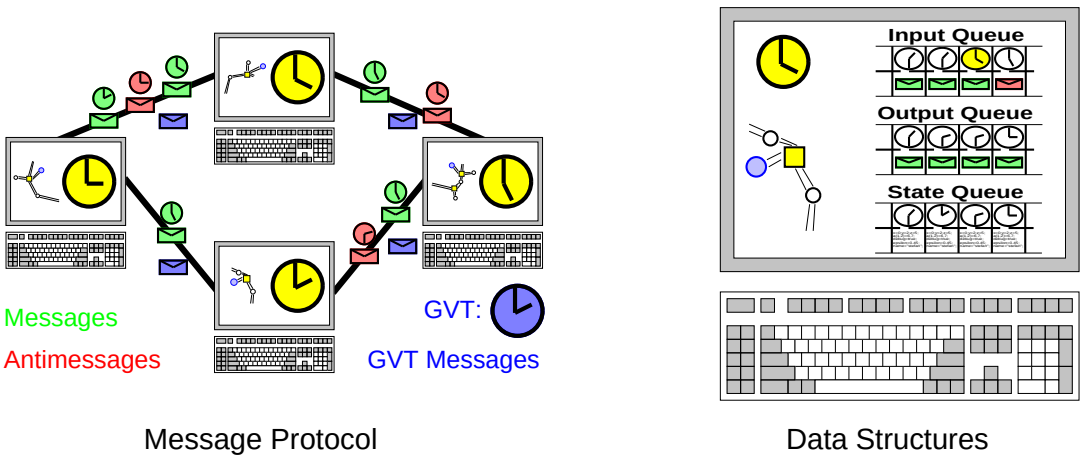


Abbildung 6.1: Grundzüge des Time-Warp-Verfahrens

stempel der Nachzüglernachricht ist) und alle zu Unrecht verschickten Nachrichten durch das Versenden sogenannter Antinachrichten zu annullieren. Hierbei gibt es zwei verschiedene Varianten: die sogenannte *Aggressive-Cancellation* und die *Lazy-Cancellation*.

Der sehr hohe Speicherplatzverbrauch, den das Sichern lokaler Zustandskopien verursacht, ist eines der Hauptprobleme des Time-Warp-Verfahrens. Deshalb gibt es zusätzlich zu diesem gerade beschriebenen lokalen Protokoll noch einen globalen Kontrollmechanismus, der notwendig ist, um das Time-Warp-Verfahren korrekt durchzuführen. Dieses globale Protokoll bestimmt eine untere Schranke der Zeit, die das Simulationssystem insgesamt erreicht hat, d. h. eine Zeit, vor die keiner der LPs jemals zurückrollen wird. Diese untere Schranke, *globale virtuelle Zeit* (GVT) genannt, wird benötigt zur

- Fossil-Collection (Zustandskopien und Nachrichten mit einer Zeit kleiner als die GVT werden nicht mehr benötigt und können deshalb gelöscht werden),
- Ein- und Ausgabe (z. B. dürfen nur solche Operationen stattfinden, bei denen sicher ist, daß sie nicht mehr durch einen Rollback rückgängig gemacht werden können, d. h. die einen Zeitstempel kleiner der GVT haben),
- Fehlerbehandlung (Fehler könnten nur aufgrund der falschen Bearbeitungsreihenfolge von Ereignissen auftreten und später durch Rollbacks wieder verschwinden),
- Entdeckung der Terminierung der Simulation (siehe [Jef85]).

Trotz dieser GVT-Ermittlung kann der durch die Zustandsliste benötigte Speicherplatz ein Problem darstellen. Deshalb gibt es Varianten der Zustandssicherung, die den benötigten Speicherplatz durch gute Checkpoint-Intervalle oder inkrementelle Zustandssicherung weiter zu reduzieren trachten. Ein großes Problem kann auch die Erzeugung einer ausgewogenen Lastsituation auf den Prozessoren sein. Schließlich stellt – wie auch bei der sequentiellen Simulation – die effiziente Verwaltung der Ereignisse ein Problem für die Implementierung eines verteilten Simulators dar.

Im Rahmen des Projektes DNS wurden die verschiedenen Varianten des Time-Warp-Protokolls im Hinblick auf ihren erfolgreichen Einsatz bei der THORN-Simulation überprüft und implementiert. Zwei Probleme, die Bereitstellung effizienter Datenstrukturen und die effektive Durchführung der GVT-Approximation, wurden in einem allgemeinen Kontext behandelt und sollen nun näher erläutert werden.

6.2 GVT-Approximation

Der benötigte Speicherplatz zur Sicherung der Zustände und zur Verwaltung von Ereignisnachrichten in der Input- und Output-Queue kann ein großes Problem optimistischer Simulationsverfahren sein. Zur Lösung dieses Problems kann eine effiziente GVT-Ermittlung zumindest teilweise beitragen. Schließlich müssen auch Fehlermeldungen und andere Ausgaben solange zurückgehalten werden, bis ihre Irreversibilität anhand des Fortschritts der GVT festgestellt wird. Auch die Leistung verschiedener Varianten des Time-Warp-Protokolls hängen von einer genauen GVT-Ermittlung ab. Schließlich zeigen Messungen

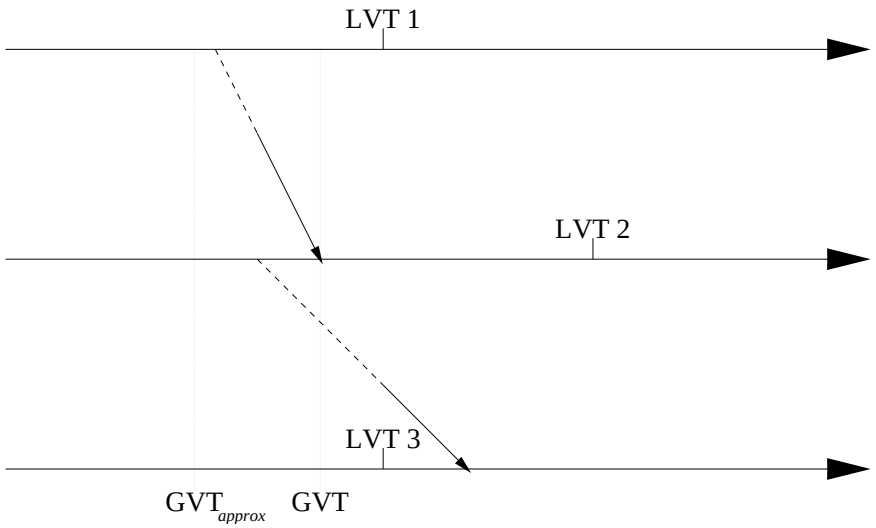


Abbildung 6.2: GVT-Approximation

(etwa [SE96, Abb. 3]), daß die Effizienz des verwendeten GVT-Approximationsalgorithmus einen starken Einfluß auf den erreichbaren Speedup haben kann.

Für die Aufgabenstellungen, bei denen die GVT benötigt wird, muß nicht unbedingt die exakte GVT bekannt sein, sondern es genügt eine (möglichst genaue) Annäherung von unten. In die GVT-Ermittlung müssen dabei die LVTs sowie die Zeitstempel der unterwegs befindlichen Nachrichten eingehen (siehe Abb. 6.2).

Zur Bewertung von GVT-Approximationsalgorithmen wurde im Rahmen des Projektes DNS in einer Diplomarbeit [Kru95] ein Testprogramm entwickelt. Hierin sind verschiedene Parameter der zugrundeliegenden Simulationsanwendung wie etwa die Topologie und Prozessorzahl des verteilten Systems, die Geschwindigkeit der Kommunikation und die Häufigkeit von Ereignisnachrichten frei wählbar. Durch das Testprogramm ermittelte Kenngrößen sind: die *GVT-Differenz*, d. h. die Differenz zwischen der durch den Algorithmus approximierten und der tatsächlichen GVT, die *Speicherbelastung* der Simulation, die einen Indikator für die Leistungsfähigkeit des verwendeten GVT-Approximationsalgorithmus darstellt, und die *Simulationsgeschwindigkeit*, die angibt, wie stark der eigentliche Simulationsalgorithmus durch die für die GVT-Approximation benötigte Rechenzeit in seinem Fortschritt gehindert wird.

Es wurden verschiedene GVT-Algorithmen hinsichtlich ihrer theoretischen Eigenschaften untersucht. Aufgrund dieser Analyse wurden schließlich die Algorithmen aus [Bel90], aus [LL90] und aus [Mat93] mit Hilfe des Testprogramms auf ihre Leistungsfähigkeit hin überprüft.

In Abb. 6.3 ist das Verhalten der GVT-Approximationsverfahren für steigende Kommunikationsaktivität der Simulationsanwendung dokumentiert. Die Kommunikationsrate ist in Prozent angegeben; eine Rate von n Prozent bedeutet, daß n Prozent der Ereignisausführungen das Versenden einer externen Ereignisnachricht zur Folge haben. Der Algorithmus aus [Bel90] wurde dabei jeweils nur bis zu einer Kommunikationsrate von knapp 50 Prozent getestet, da ab dieser Größe durch die in diesem Algorithmus notwendi-

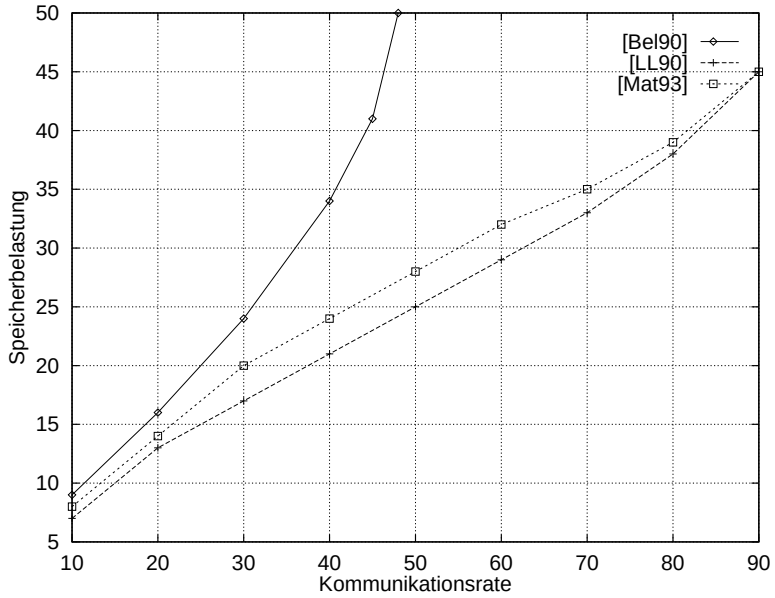


Abbildung 6.3: Veränderliche Kommunikationsrate: Speicherbelastung

gen Quittungsnachrichten für jede Ereignisnachricht die internen Kommunikationspuffer des Systems überliefern. Wie zu sehen ist, verhalten sich der Algorithmus aus [LL90] und der aus [Mat93] sehr ähnlich zueinander. Das Verhalten ist annähernd linear: mit steigender Kommunikationsrate steigt auch die Speicherbelastung etwa proportional an. Dies liegt daran, daß mit der steigenden Kommunikationsaktivität im System die GVT-Nachrichten erst entsprechend später bearbeitet werden können, also zunehmend ältere Prozessorzustände widerspiegeln. Dadurch steigt die GVT-Differenz an und damit auch die Größe des benötigten Speicherplatzes in den Datenstrukturen der LPs. Beim Algorithmus aus [Bel90] verstärkt sich dieser Effekt noch durch die zusätzliche Belastung des Netzwerks mit den notwendigen Quittungsnachrichten.

Auch die Simulationsgeschwindigkeit der Algorithmen aus [LL90] und [Mat93] unterscheidet sich nicht sehr (siehe Abb. 6.4). Aufgrund der Quittungsnachrichten ist die Simulation allerdings unter Verwendung des Algorithmus aus [Bel90] insbesondere bei höheren Kommunikationsraten wesentlich langsamer.

In Abb. 6.5 ist das Verhalten der Simulation bei veränderlichen Geschwindigkeiten des Kommunikationsnetzwerks dokumentiert. Die Algorithmen aus [LL90] und [Mat93] weisen wiederum große Ähnlichkeit auf. Schnelle Kommunikation hat durchweg eine niedrigere GVT-Differenz zur Folge als langsame Kommunikation bei gleicher Kommunikationsrate; auch bei schneller Kommunikation steigt die GVT-Differenz mit wachsender Kommunikationsrate. Der Algorithmus aus [Mat93] zeigt durchgängig eine geringfügig schlechtere Approximation als der Algorithmus aus [LL90]. Der Algorithmus aus [Bel90] zeigt gleiche Tendenzen, ist aber deutlich schlechter als die anderen beiden Algorithmen.

Es soll nun vorgestellt werden, welchen Einfluß die Topologiegröße auf das Verhalten der GVT-Approximationsalgorithmen hat. Getestet wurde auf unterschiedlich dimensionierten Hypercube-Topologien. Es wird dabei angenommen, daß die LPs nur in der durch die

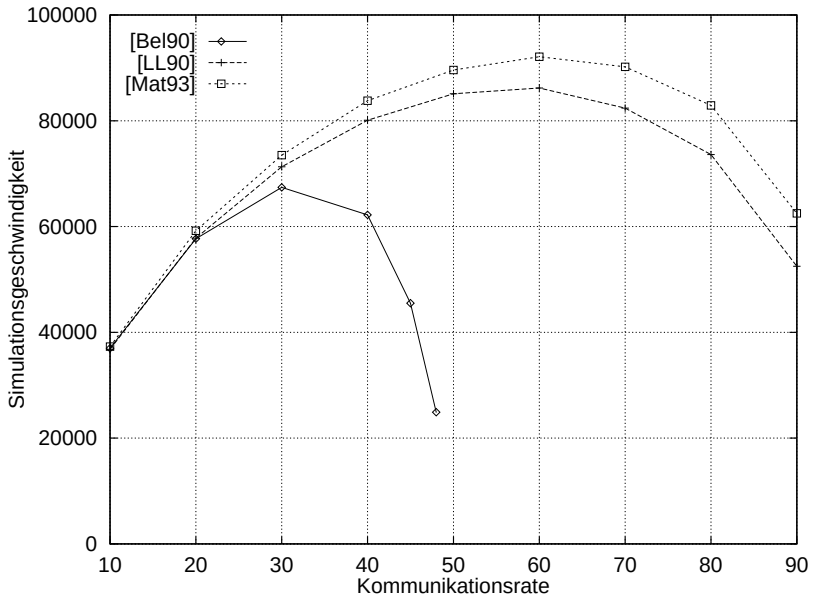


Abbildung 6.4: Veränderliche Kommunikationsrate: Geschwindigkeit

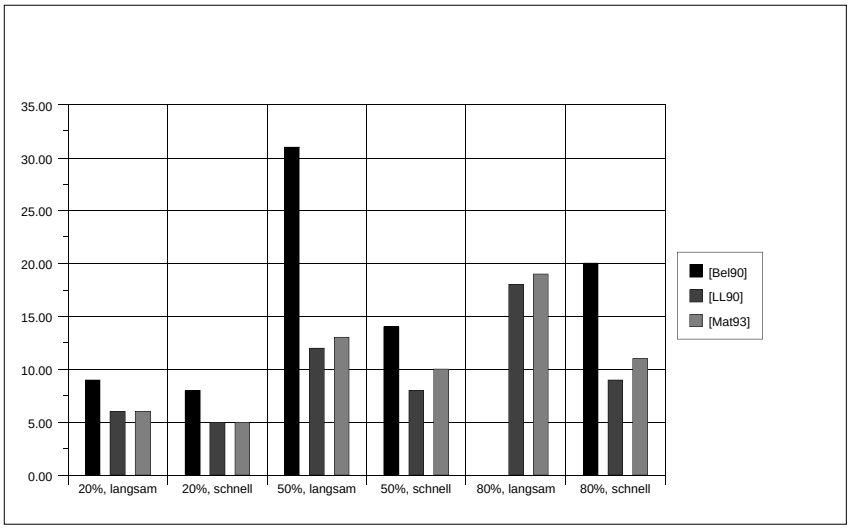


Abbildung 6.5: Veränderliche Kommunikationsgeschwindigkeit: GVT-Differenz

Topologie vorgegebenen Art und Weise miteinander kommunizieren. Interessant ist diese Messung vor allem, wenn die theoretisch ermittelten Laufzeitkomplexitäten der Algorithmen betrachtet werden. Der Algorithmus aus [Bel90] besitzt eine Laufzeitkomplexität in $O(\log n)$, benötigt aber für jede Ereignisnachricht eine zusätzliche Quittungsnachricht. Der Algorithmus aus [LL90] hat eine Laufzeit in $O(\log n + v)$, wobei v die durchschnittliche Anzahl der Vorgänger eines Prozessors ist, in der verwendeten Hypercube-Topologie also $\ln n$. Damit ist die Laufzeit also insgesamt in $O(\log n)$, Bestätigungsnachrichten sind nicht erforderlich. Der Algorithmus aus [Mat93] hat ebenfalls eine Laufzeitkomplexität in

mindestens $O(\log n)$, kann aber zusätzlich durch die Laufzeiten der Ereignisnachrichten verzögert werden.

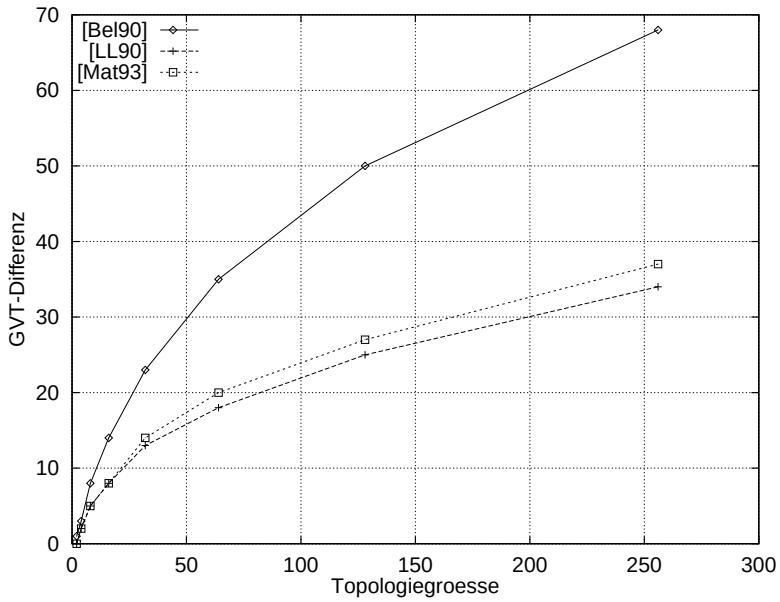


Abbildung 6.6: Veränderliche Topologiegröße: GVT-Differenz

Die theoretischen Laufzeitkomplexitäten werden durch die empirischen Untersuchungen bestätigt (siehe Abb. 6.6). Alle drei Algorithmen spiegeln in der gemessenen durchschnittlichen GVT-Differenz die logarithmische Laufzeit wider. Die zusätzlichen Bestätigungsnachrichten im Algorithmus aus [Bel90] schlagen sich allerdings in einer höheren Differenz zwischen approximierter und tatsächlicher GVT nieder.

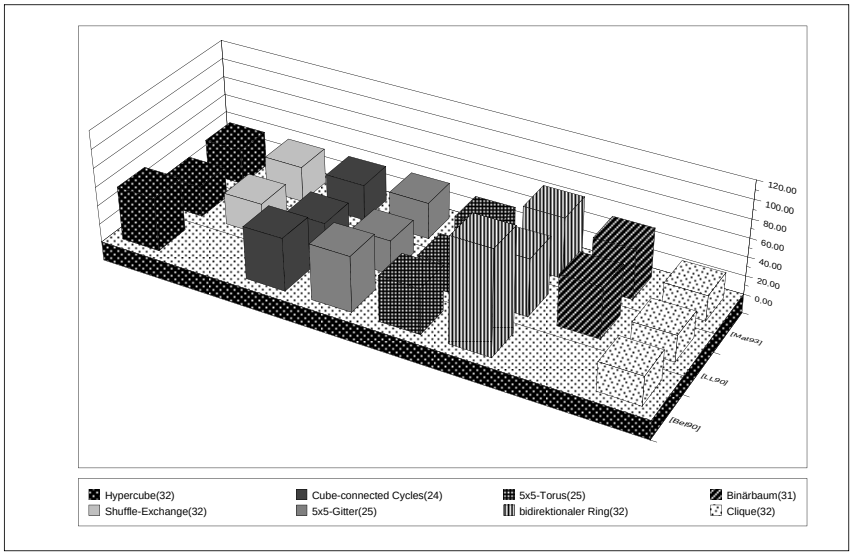


Abbildung 6.7: Veränderliche Topologie: Speicherverbrauch

Der Einfluß der Kommunikationstopologie auf den Speicherverbrauch ist in Abb. 6.7 zu sehen. Auch hier bestätigt sich wieder das annähernd gleiche Verhalten der Algorithmen aus [LL90] und [Mat93], während der Algorithmus aus [Bel90] einen deutlich höheren Speicherverbrauch der Simulation bewirkt.

Insgesamt zeigen also die Algorithmen aus [LL90] und [Mat93] ein ähnlich gutes Verhalten. Der Algorithmus aus [LL90] hat allerdings den Vorteil, daß er sich in Systemen, die FIFO-Kommunikation gewährleisten, noch weiter vereinfachen läßt. Aus diesem Grunde wurde er im verteilten THORN-Simulator implementiert.

6.3 Die Ereignisliste bei der verteilten Simulation

Ein wichtiger Faktor für die gute Leistung eines Simulators kann eine effiziente Implementierung der Liste zur Verwaltung der Ereignisse sein. Dies gilt insbesondere dann, wenn – wie im verteilten THORN-Simulator – bedingt durch die möglichen objektabhängigen Verzögerungszeiten eine große Anzahl von Ereignissen während der Simulation anfällt. Datenstrukturen, wie sie in der sequentiellen Simulation benutzt werden, lassen sich nicht unmittelbar für die Ereignisliste des verteilten Simulators einsetzen, da auch bereits verarbeitete Ereignisse im Falle von Rollbacks noch benötigt werden (siehe [Sch96b, Sch97] für Details). Es wurde deshalb – ausgehend von einer effizienten Datenstruktur für die sequentielle Simulation – eine Datenstruktur entwickelt und implementiert, die unabhängig vom Anwendungsgebiet der verteilten THORN-Simulation einen effizienten Zugriff auf die Ereignisliste erlaubt.

Diese Datenstruktur, die *Time-Warp-Calendar-Queue* (TWCQ), soll im folgenden kurz vorgestellt werden. Eine detailliertere Beschreibung wurde in [Sch97] veröffentlicht.

Die TWCQ ist wie ein Taschenkalender organisiert, wobei für jeden Tag des Jahres eine Seite zur Verfügung steht. Jede Seite enthält eine sortierte, doppelt verkettete lineare Liste (DLL) mit den an diesem Tag stattfindenden Ereignissen, wobei jedes Ereignis seinen Zeitstempel mit sich führt. Implementiert wird die TWCQ als Feld von Buckets. Jeder Bucket ist dabei für einen kompletten Tag verantwortlich, d. h. enthält alle Ereignisse eines bestimmten Zeitraums. Dieser Zeitraum, d. h. die *Tageslänge*, ist für alle Buckets gleich, nur ihre Startzeitpunkte unterscheiden sich voneinander. Die TWCQ besitzt einen Zeiger, der immer den aktuellen Bucket referenziert. Wenn der letzte Bucket, d. h. das *Jahresende*, erreicht wurde, findet ein Sprung zum ersten Bucket, zum *Jahresanfang*, statt. Diese Idee wird dazu benutzt, um die in anderen listenbasierten Implementierungen üblichen Überlauflisten einzusparen: die Nummer des Buckets für ein Ereignis mit einem Zeitstempel, der mehr als ein Jahr in der Zukunft liegt, wird dadurch bestimmt, daß immer der Zeitstempel eines Ereignisses modulo der *Jahresweite* (Anzahl der Buckets multipliziert mit der Tageslänge) betrachtet wird.

Um zu verhindern, daß die DLLs in den Buckets zu lang werden (dies würde die Leistung der Einfügeoperation mindern) oder daß zu viele Buckets leer werden (dies würde die Entnahme von Elementen verlangsamen), wird von Zeit zu Zeit eine Reorganisation der TWCQ durchgeführt.

In Abb. 6.8 ist eine TWCQ zum Zeitpunkt 14.7 dargestellt, wobei der aktuelle Bucket

durch den Pfeil und das augenblickliche Element in jedem Bucket durch die Unterstriche angedeutet ist.

Eine Ereignisnachricht mit einem Zeitstempel, der größer als die aktuellen LVT ist, wird dadurch in die TWCQ einsortiert, daß zunächst die Nummer des Buckets berechnet wird, in den diese Ereignisnachricht fällt. Diese Nummer läßt sich durch eine einfache arithmetische Operation aus dem Zeitstempel der Ereignisnachricht und der zeitlichen Weite sowie der Anzahl der Buckets ermitteln. Ist die Nummer des Buckets gefunden, wird die DLL in diesem Bucket ausgehend vom augenblicklichen Element durchsucht, bis die Einfügestelle gefunden ist.

Zur Entnahme von Ereignissen wird ausgehend vom aktuellen Element des momentanen Buckets seine DLL durchsucht, bis das nächste Ereignis gefunden wird. Hierbei dürfen allerdings nur solche Ereignisse zurückgegeben werden, die ins aktuelle Jahr fallen, d. h. die Ereignisse, die quasi zu den implizit in den Buckets mitverwalteten Überlauflisten gehören, werden zunächst nicht zurückgegeben. Sind im aktuellen Bucket keine Ereignisse, die zum aktuellen Jahr gehören, vorhanden, so wird der Zeiger auf den aktuellen Bucket inkrementiert; wird hierbei der letzte bucket erreicht, so wird er zurück auf den ersten Bucket gesetzt. Das zurückgegebene Ereignis wird nicht aus der TWCQ gelöscht, sondern es werden nur die entsprechenden Zeiger auf den aktuellen Bucket und das aktuelle Element verändert.

Ein tatsächliches Löschen findet bei der im Rahmen des Time-Warp-Verfahrens durchgeführten Fossil-Collection statt. Hierbei wird, angefangen mit dem Bucket, der das kleinste Element enthält, die jeweilige DLL von ihrem Kopf aus traversiert und alle Elemente mit einem Zeitstempel, der kleiner als die GVT ist, gelöscht. Ähnlich wird bei einem Rollback verfahren, wo ausgehend vom aktuellen Bucket der jeweilige Zeiger in der DLL zurückgesetzt wird.

Leistungsbewertungen von Ereignislistenimplementierungen finden üblicherweise gemäß dem sogenannten *Hold-Model* statt [Jon86]. Hierbei wird vereinfachend davon ausgegangen, daß in der Simulationsanwendung auf eine Entnahmeoperation ein Einfügen eines neuen Elementes mit einem höheren Zeitstempel erfolgt. Der Zeitstempel des eingefügten Elementes ist dabei um einen gewissen Betrag größer als der Zeitstempel des entnommenen Elementes. Hierbei wird der Betrag mittels gewissen Verteilungen genügenden Zufallszahlengeneratoren berechnet.

```
   Bucket 0:  12.3 ↔ _16.2              /* 12.0-12.5  */
   Bucket 1:  12.8 ↔ _16.6              /* 12.5-13.0  */
   Bucket 2:  13.4_                     /* 13.0-13.5  */
   Bucket 3:  _17.8                     /* 13.5-14.0  */
   Bucket 4:  _                         /* 14.0-14.5  */
→ Bucket 5:  14.5 ↔ _14.7 ↔ 14.8        /* 14.5-15.0  */
   Bucket 6:  _15.2 ↔ 15.3 ↔ 19.1       /* 15.0-15.5  */
   Bucket 7:  11.4 ↔ _15.9              /* 15.5-16.0  */
```

Abbildung 6.8: Inhalt einer TWCQ

Bei Ereignislisten für das Time-Warp-Verfahren müssen zusätzlich zu den durch das Hold-Model abgedeckten Operationen auch noch Fossil-Collection und Rollback überprüft werden. Theoretische Analysen der TWCQ ergaben ein $O(1)$-Verhalten im Hold-Model im Gegensatz zu einem $O(n)$-Verhalten der üblicherweise im Time-Warp-Verfahren benutzten doppelt verketteten linearen Liste (hierbei ist n die Anzahl der Ereignisse). Rollbacks und Fossil-Collection können in beiden Datenstrukturen in $O(m)$ durchgeführt werden (m entspricht der Anzahl der betroffenen Ereignisse), wobei die TWCQ allerdings einen gewissen Zusatzaufwand durch den notwendigen Zugriff auf die Buckets verursacht.

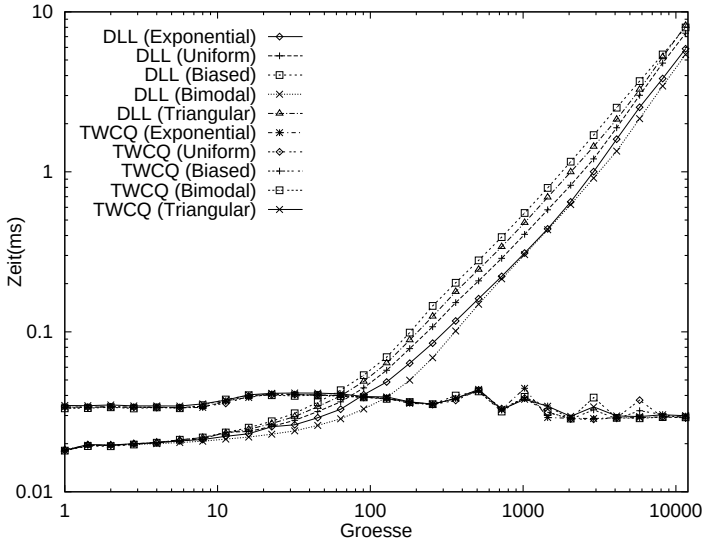


Abbildung 6.9: Aufwand der Hold-Operation

Die Leistung der TWCQ im Vergleich zur üblicherweise benutzten linearen Liste wurde deshalb auch experimentell bewertet. In Abb. 6.9 ist zu sehen, daß bei den durchgeführten Experimenten bei der TWCQ eine Hold-Operation tatsächlich eine von der Größe der Input-Queue unabhängige Zeit beansprucht. Es ist ferner zu sehen, daß das Zeitverhalten der TWCQ weitestgehend unabhängig von der verwendeten Verteilung zur Bestimmung der Zeitstempel ist. Auch bei relativ kurzen Ereignislisten ist der Aufwand der TWCQ-Implementierung für eine Hold-Operation nur etwa doppelt so groß wie der der DLL-Implementierung. Bei größeren Input-Queues liegt der Aufwand der DLL um mehrere Größenordnungen über dem Aufwand der TWCQ (bei 11585 Elementen ist er je nach gewählter Verteilung etwa 185- bis 280mal so groß). Außerdem ist die Laufzeit der DLL-Implementierung wesentlich stärker abhängig von der verwendeten Verteilungsfunktion für die Zeitstempel als die der TWCQ. Der Aufwand von Rollback und Fossil-Collection ist bei der TWCQ aufgrund des Zusatzaufwands für die Bucket-Traversierung schlimmstensfalls anderthalbmal so groß wie bei einer linearen Liste.

Schließlich sollte experimentell bewertet werden, wie der große Leistungsgewinn für das Einfügen und Entnehmen von Ereignissen und der geringfügige Leistungsverlust bei Rollbacks und Fossil-Collection sich im Zeitverhalten einer realistischen Simulationsanwendung niederschlägt. Hierzu wurde ein Testprogramm entwickelt, das einen LP simuliert, bei dem die vier Zugriffsoperationen auf die Ereignisliste in einem einstellbaren Verhältnis

Ereignisliste	Profil	E_{min}	E_{max}	p_r	ϱ	f_f	φ
abnehmend	1	0	1	0.1	5	30	5
abnehmend	2	0	1	0.02	20	50	10
stabil	1	0	2	0.1	5	30	5
stabil	2	0	2	0.02	20	50	10

Tabelle 6.1: Verhaltensprofile der getesteten LPs

stehen. Der LP holt jeweils das nächste Ereignis aus seiner Ereignisliste und bearbeitet es. Während dieser Bearbeitung wird ihm (von sich selbst oder von anderen LPs) eine variable Zahl von Ereignisnachrichten geschickt, die er in seine Input-Queue eingliedern muß. Mit einer bestimmten Wahrscheinlichkeit ist der Zeitstempel einer ankommenden Nachricht kleiner als die LVT, so daß der LP einen Rollback durchführen muß. Außerdem muß der LP in bestimmten Abständen eine Fossil-Collection durchführen. Hierbei wächst die GVT um einen zufällig gewählten Betrag. Es wird jeweils nur ein bestimmtes Zeitintervall betrachtet, das beginnt, sobald die Ereignisliste eine vorgegebene Größe erreicht hat und nach 1000 Arbeitsschleifen endet.

Die folgenden Parameter der Simulation sind frei wählbar.

- die Anzahl der mindestens (E_{min}) und höchstens (E_{max}) pro Ereignisbearbeitung auftretenden neuen Ereignisse;
- die Wahrscheinlichkeit p_r für das Auftreten von Rollbacks;
- die zeitliche Weite ϱ von Rollbacks;
- die Häufigkeit der Fossil-Collection f_f;
- die Stärke des GVT-Anstiegs φ.

Es wurde jeweils für vier realistische Verhaltensprofile der Simulation, deren Parameterbelegung in Tabelle 6.1 aufgeführt ist, sowohl die DLL- als auch die TWCQ-Implementierung der Input-Queue untersucht. Bei den ersten beiden Verhaltensprofilen ist die durchschnittliche Anzahl der in die Ereignisliste eingefügten Elemente gleich der Anzahl der entfernten Elemente, bei den anderen beiden wird durchschnittlich nur die halbe Anzahl der entnommenen Elemente wieder in die Ereignisliste eingefügt. Dadurch, daß Rollbacks auftreten, muß die Ereignislistengröße bei den ersten beiden Profilen nicht unbedingt schrumpfen, bei den anderen beiden nicht unbedingt konstant bleiben. Beide Profile werden nun mit jeweils zwei unterschiedlichen Verhalten bzgl. Rollbacks und Fossil-Collection kombiniert. Hierbei sind einmal Rollbacks und Fossil-Collection relativ häufig, umfassen aber nur wenige Elemente, beim anderen Mal werden sie selten durchgeführt, umfassen allerdings viele Elemente. Die im folgenden präsentierten Meßwerte beziehen sich auf die für die Ereignislisten-Zugriffsfunktionen aufgewendetete Zeit.

In Abb. 6.10 sind die Meßwerte für die beiden Simulationsprofile mit (abgesehen von Rollbacks) abnehmender Größe der Ereignisliste aufgeführt. Es zeigt sich, daß die DLL-

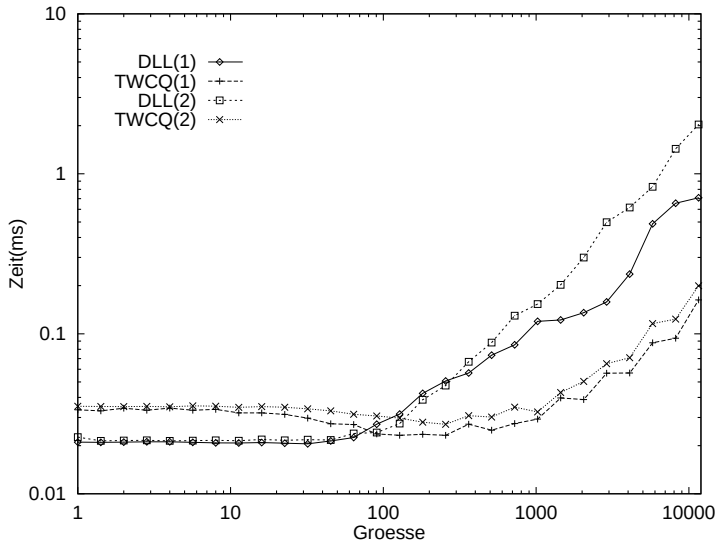


Abbildung 6.10: Zeitverhalten einer Anwendung mit abnehmender Ereignislistengröße

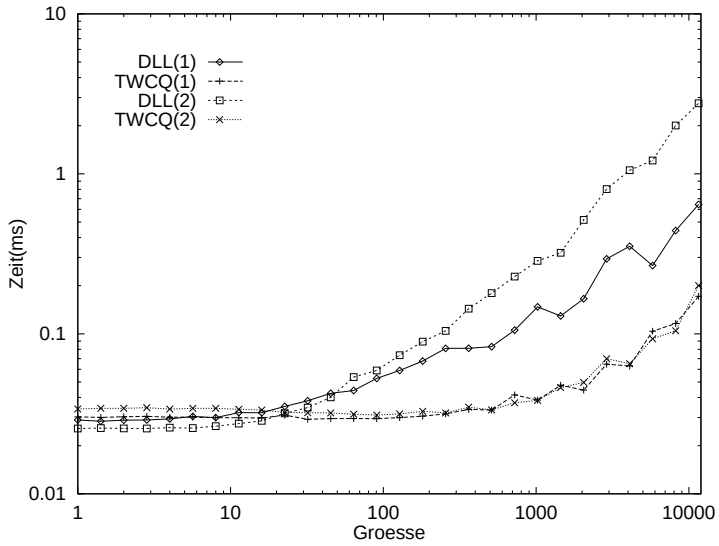


Abbildung 6.11: Zeitverhalten einer Anwendung mit stabiler Ereignislistengröße

Implementierung für kleine initiale Ereignislistengrößen um maximal den Faktor 1.6 schneller ist. Hingegen verhält sich die TWCQ für Größen ab etwa 90 bzw. 180 (je nach Profil) besser und bewirkt bei großen initialen Ereignislisten einen 7- bzw. 11.5mal schnelleren Zugriff auf die Ereignislisten. Schon bei initialen Ereignislistengrößen von 1024 Elementen beträgt das Geschwindigkeitsverhältnis ca. 4,1 bzw. 4,7. Es ist außerdem zu sehen, daß die Leistung der DLL-Implementierung wesentlich stärker vom konkreten Verhalten der Applikation bzgl. Rollbacks und Fossil-Collection bestimmt wird als die TWCQ-Implementierung.

Bei den Applikationen, bei der durchschnittlich genau so viele Ereignisse wieder in die

Input-Queue eingefügt werden wie entnommen werden, zeigt die TWCQ für fast alle initialen Ereignislistengrößen ein signifikant besseres Zeitverhalten (siehe Abb. 6.11). Lediglich für sehr kleine initiale Ereignislisten (unter 11 bzw. 32 Elemente) wurde ein geringfügig schlechteres Zeitverhalten der TWCQ (Faktor 1,04 bzw. 1,32) gemessen. Das schon für kleine Größen eine Leistungssteigerung zu beobachten ist, liegt daran, daß auch bei kleinen Initialgrößen durch die auftretenden Rollbacks die Ereignisliste im Laufe der 1000 Meßschleifen signifikant anwächst. Für Profil 1 (häufige, aber kurze Rollbacks bzw. Fossil-Collections) wird eine Geschwindigkeitssteigerung von bis zu 5,6 durch den Einsatz der TWCQ erreicht. Schon für eine initiale Ereignislistengröße von 1024 Elementen ist sie über 3,8mal schneller. Für seltenere, aber mehr Elemente umfassendere Rollbacks und Fossil-Collections wird schon für 90 Elemente in der initialen Ereignisliste fast eine Verdopplung der Geschwindigkeit erreicht, für 256 Elemente beträgt der Faktor bereits 3,25, für 1024 Elemente 7,45 und für 8192 Elemente 19,2. Insgesamt verhalten sich also bei einer Anwendung mit stabiler Ereignislistengröße TWCQs bis zu 19mal besser als DLLs und schlimmstenfalls für sehr kleine Anwendungen höchstens 30 Prozent schlechter. Zusätzlich ist wiederum zu sehen, daß DLLs in ihrem Zeitverhalten wesentlich stärker vom verwendeten Profil abhängen als TWCQs.

Insgesamt ist also ein signifikanter Leistungsgewinn durch TWCQs zu sehen, der unabhängig von der Art der Anwendung schon bei einer moderaten Ereignislistengröße auftritt. Dies bestätigte sich auch in Experimenten, bei denen der Einsatz der TWCQ im optimistischen THORN-Simulator überprüft wurde. Bei verschiedenen Experimenten zeigte sich, daß abhängig von der Anzahl der Schaltereignisse, die in einem THORN-Modell anfallen, die TWCQ der DLL beliebig stark überlegen ist. Bei einem Modell mit insgesamt 10000 anfallenden Ereignissen konnte etwa der Anteil der für die Operationen auf der Ereignisliste notwendigen Berechnungen an der Gesamtsimulationszeit durch die Verwendung der TWCQ von 17.1 auf 5.8 Prozent gesenkt werden. Die Gesamtlaufzeit der Simulation war dadurch unter Verwendung der DLL-Implementierung um ca. 23.8 Prozent höher als bei Verwendung der TWCQ.

6.4 Der verteilte THORN-Simulator

Der im Rahmen des Projektes DNS entwickelte verteilte Simulator für THORNs wurde komponentenweise entwickelt. Die verschiedenen Konzepte von THORNs wurden dabei so weit als möglich einzeln behandelt. Auf diese Weise wurden Simulatoren für einfachere Netzklassen implizit mitentwickelt. Der Vorteil dieser Methode ist außerdem, daß zur Änderung der unterstützten Netzklasse der Simulationsalgorithmus nur an festdefinierten Stellen geändert werden muß; so muß etwa für andere Kantentypen als die in THOR-Netzen unterstützten nur der entsprechende Teil des Algorithmus angepaßt werden. Die Entwicklung eines Algorithmus für eine solch komplexe Modellierungssprache, wie THORNs sie darstellen, stellt eine sehr komplizierte Aufgabe dar, insbesondere, wenn gleichzeitig verschiedene Nebenbedingungen, wie etwa der Verzicht auf globale Daten bei der verteilten Simulation, eingehalten werden müssen. Auch aus diesem Grund hat sich die schrittweise Vorgehensweise beim Entwickeln und Testen des verteilten THORN-Simulators im Laufe des Projektes bewährt.

Der Simulationsalgorithmus abstrahiert weitestgehend von der verwendeten Synchronisationsmethode und basiert lediglich auf der den meisten verteilten Simulationsmethoden gemeinen Annahme, daß das Simulationsmodell in einzelne LPs aufgeteilt ist, die über Ereignisnachrichten miteinander kooperieren. Dadurch ist er für verschiedene Synchronisationsverfahren gleichermaßen anwendbar. Aufgrund ihrer besseren Eignung für die Simulation höherer Petrinetze wurden vornehmlich optimistische Konzepte bei der THORN-Simulation eingesetzt.

Um die verteilte Simulation eines THOR-Netzes durchzuführen, wird es – ähnlich wie bei der sequentiellen Simulation (vgl. Abb. 4.1) – übersetzt und auf den verschiedenen Prozessoren einer verteilten Architektur ausgeführt. Hierzu sind folgende Schritte nötig (siehe Abb. 6.12). Zunächst wird das aus dem Netzeditor abgespeicherte THORN zusammen mit einer Anfangsmarkierung durch einen speziellen Netz-Compiler übersetzt. Die Ausgabe dieses Netz-Compilers ist einerseits C++-Code, der hauptsächlich die modellspezifischen C++-Beschriftungen enthält (Schaltaktionen, Schaltbedingungen und Zeitfunktionen der Transitionen sowie den modellspezifischen C++-Code) sowie andererseits eine Zwischendarstellung, die sogenannte ANR-Darstellung [SWK+94], die Informationen über die Netzstruktur enthält. Der generierte C++-Code wird durch einen Standard-C++-Compiler übersetzt und mit einer spezifischen Simulationsbibliothek zu einem ausführbaren Programm gebunden. Die Simulationsbibliothek besteht dabei aus zwei getrennten Teilen, deren einer die Unterstützung zur Durchführung des Tokenspiels entsprechend der THORN-Semantik, deren anderer das jeweilige Protokoll zur Synchronisation der einzelnen LPs beinhaltet. Beide Teile sind dabei in gewissen Grenzen austauschbar, allerdings muß natürlich eine bestimmte Schnittstelle zwischen den Teilen eingehalten werden. Das erzeugte Netzsimulationsprogramm benötigt die erzeugte Zwischendarstellung zur Initialisierung gewisser Datenstrukturen und liest außerdem Simulationsoptionen – etwa die gewünschte Dauer

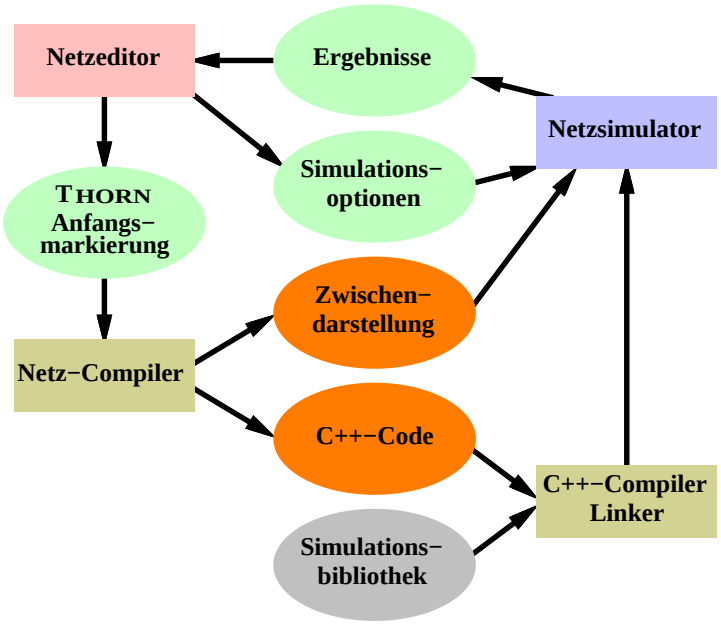


Abbildung 6.12: Software-Architektur des verteilten Simulators

der Simulation – aus einer durch den Netzeditor bereitgestellten Datei ein. Schließlich werden die Simulationsergebnisse, d. h. vor allem die erzeugte Endmarkierung, in einer Datei abgelegt, die wiederum in den Netzeditor eingelesen und dort angezeigt werden kann.

Das erzeugte verteilte Programm instantiiert pro Prozessor nur einen Prozeß, *PCC* (processor control and communication) genannt. Die PCCs kooperieren mittels Nachrichtenkommunikation unter Verwendung der Bibliothek PVM [GBD+94]. Jede Unternetzinstanz des THOR-Netzes induziert einen eigenen logischen Prozeß; zur Generierung und Löschung von Unternetz-LPs und zur Verwaltung von Share-Stellen wird zusätzlich ein eigener LP benötigt.

Die einzelnen LPs werden nicht als eigenständige Prozesse auf Betriebssystemebene, sondern als Objekte in der PCC realisiert. Dies hat den Vorteil, daß

- die Kommunikation zwischen LPs auf einem Prozessor kostengünstiger durchgeführt werden kann;
- das Scheduling der LPs direkt beeinflußt werden kann;
- das Generieren und Löschen von LPs nicht durch entsprechende Prozeßkonstrukte auf Betriebssytemebene durchgeführt werden muß, sondern – im Vergleich hierzu relativ kostengünstig – durch Objektinstantiierungen in einem Programm.

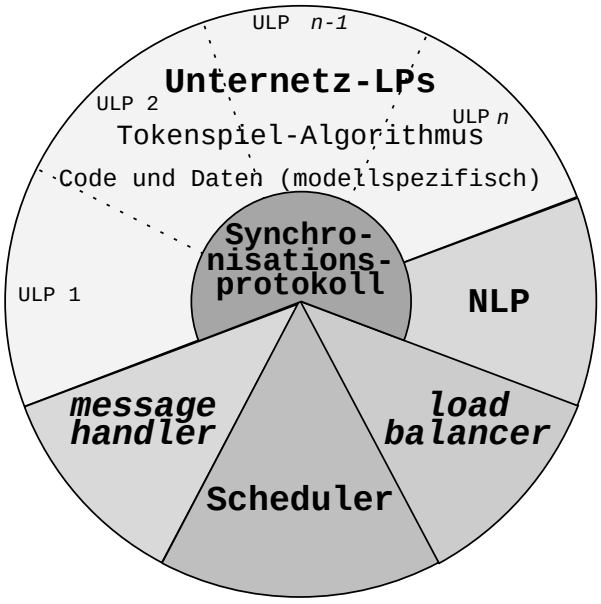


Abbildung 6.13: Schematische Darstellung der PCC

Die LPs werden durch ein Synchronisationsprotokoll unterstützt. Im Rahmen des Projektes DNS wurden vor allem Varianten des Time-Warp-Verfahrens eingesetzt; in einer Diplomarbeit [Poh96] wurde allerdings auch ein konservatives Verfahren untersucht, das sich allerdings aufgrund der mangelnden Möglichkeiten zur Vorausschau nicht als sonderlich

effizient erwies. Teile des Synchronisationsprotokolls, wie etwa die GVT-Approximation bei der optimistischen Simulation, sind nicht an die LPs gebunden, sondern treten pro PCC genau einmal auf.

Außerdem führt die PCC noch folgende Aufgaben durch:

- Über den *message handler* wird die Kommunikation zwischen LPs ermöglicht. Prozessorinterne Kommunikation kann dabei über das Kopieren entsprechender Daten realisiert werden, während Kommunikation mit LPs auf anderen Prozessoren durch Versenden von Nachrichten über die PVM durchgeführt wird.
- Der *Scheduler* wählt aus den auf dem Prozessor ausgeführten LPs denjenigen aus, der das nächste Ereignis bearbeiten darf. Die Arbeitsweise des Schedulers unterscheidet sich je nach verwendetem Synchronisationsprotokoll. Bei den Time-Warp-Verfahren wird hier ein MMT-Scheduling [PML92] verwendet.
- Der *load balancer* unterstützt notwendige Maßnahmen zum Lastausgleich. Insbesondere wird hier entschieden, ob ein neu instantiiertes Unternetz auf einem anderen Prozessor ausgeführt werden soll. Außerdem wirkt dieser Teil der PCC bei Maßnahmen zum dynamischen Lastausgleich mit (genaueres siehe [Ree96]).

Eine schematische Darstellung der PCC enthält Abb. 6.13.

6.5 Laufzeitverhalten

In einer Diplomarbeit [Ree96] wurde das Laufzeitverhalten des optimistischen THORN-Simulators überprüft und untersucht, welche Varianten des Time-Warp-Protokolls effizienzsteigernd eingesetzt werden können. Die wichtigsten Ergebnisse hieraus sollen im folgenden vorgestellt werden. Insbesondere wird anhand von Laufzeitresultaten die Leistung des verteilten Simulators bewertet. Meßplattform für die präsentierten Resultate war ein via 10-MBit-Ethernet verbundenes Netzwerk von DEC-Alpha-Workstations mit Prozessoren des Typs Alpha 21064.

Es werden im folgenden hauptsächlich keine absoluten Laufzeiten, sondern Speedup-Werte angegeben. Die angegebene Beschleunigung bezieht sich jeweils auf einen Vergleich mit der Leistung des verteilten Simulators auf einem Prozessor, wobei allerdings die zeitaufwendigen Teile des Time-Warp-Protokolls, die auf einem Prozessor nicht benötigt werden (etwa Zustandssicherung und GVT-Ermittlung), abgeschaltet sind. Die Verwendung dieser Vergleichsmeßgröße trägt der Tatsache Rechnung, daß die Zeit zur Entwicklung und Implementierung der Simulatoren trotz der unterschiedlich komplexen Aufgabenstellung etwa gleich war und deshalb der verteilte Simulator, dessen Implementierung etwa dreieinhalbmal soviele Programmzeilen wie der sequentielle Simulator umfaßt (ca. 45000 im Vergleich zu ca. 13000), zwangsläufig weniger stark optimiert werden konnte.

Soll also von unterschiedlichen Implementierungsansätzen und unterschiedlichem Optimierungsaufwand abstrahiert werden, so eignet sich der um die Time-Warp-Anteile bereinigte optimistische Simulator auf einem Prozessor besser zum Speedup-Vergleich. Soll

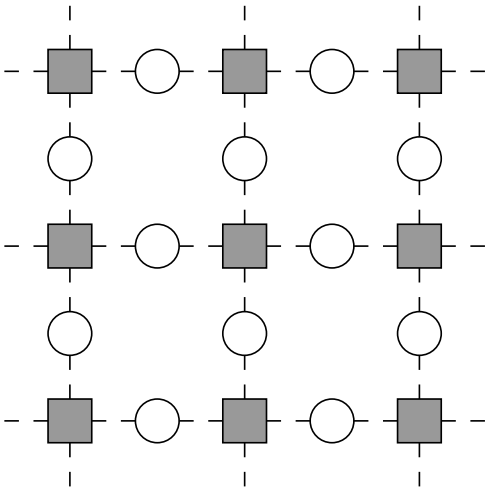


Abbildung 6.14: Ausschnitt des Zellularautomatennetz

hingegen der erhöhte Aufwand, der für das Design und die Implementierung des verteilten Simulators gegenüber dem sequentiellen nötig war, miteinberechnet werden (entsprechend der Fragestellung, welche Leistung bei gleichen Entwicklungskosten erreicht werden kann), so sollte der sequentielle Simulator zum Vergleich herangezogen werden.

Im folgenden werden ausgewählte Testergebnisse vorgestellt, die anschließend bewertet und den Resultaten vergleichbarer Arbeiten gegenübergestellt werden.

6.5.1 Ausgewählte Testergebnisse

Es wurden insgesamt vier verschiedene größere Testnetze untersucht. Ein Modell stellt dabei einen 6-Bit-Pipeline-Multiplizierer als THORN dar. Dieses Netz ist bis ins Detail modelliert und umfaßt zur Laufzeit einen Zustandsraum von 4929 Stellen, auf dem 2363 echte (d. h. nicht verfeinerte) Transitionen operieren. Ein weiteres, allerdings auf C++-Ebene nicht ausformuliertes, sondern mit synthetischen Lasten arbeitendes Netzmodell stellt einen Prototyp für einen einfachen zellularen Automaten wie etwa das „Game of Life“ [Gar70] dar. Besonderheit dieses Modells ist es, daß alle Unternetze sehr stark über Share-Stellen mit anderen Netzen verbunden sind (bis auf die Randnetze hat jedes Unternetz vier Nachbarn, siehe Abb. 6.14), daß aber in der Regel nur in jedem vierten Fall tatsächlich ein Zugriffskonflikt entsteht. Bei diesem Netz kann sich also zeigen, inwieweit der Simulationsalgorithmus fähig ist, trotz der potentiell engen Kopplung aller Unternetze noch eine effektive Nutzung der Ressourcen des verwendeten Rechnernetzes zu ermöglichen. In einem weiteren Beispiel wurde der Anwendungsraum der Modellierung verlassen und die Fähigkeiten von THOR-Netzen als Erweiterung von C++ zur Unterstützung automatischer Parallelisierungstechniken ausgenutzt. In diesem THORN wird eine Heuristik für das bekannte Problem des Handlungsreisenden (Suche kürzester Touren) nachgebildet. Die in THORNs gegebenen Möglichkeiten zur Zeitbeschriftung werden hier benutzt, um die sequentielle Ordnung der Operationen anzugeben. Ein viertes Netz diente dazu, Grenzen der verteilten Simulation aufzuzeigen: aufgrund hoher Abhängigkeiten zwischen den

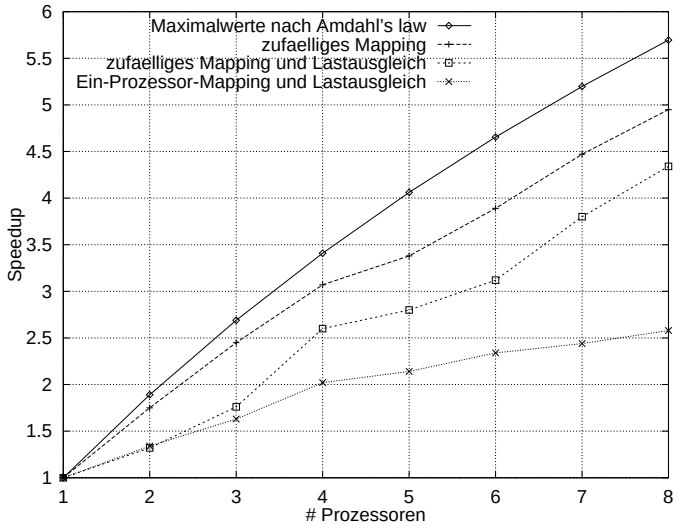


Abbildung 6.15: Auswirkung der Lastausgleichsverfahren beim TSP-Netz

einzelnen Netzteilen kann eine effiziente Parallelisierung hier nicht vorgenommen werden.

Das TSP-Netz zeigt aufgrund seiner einfachen, regelmäßigen Struktur ein sehr gutes Speedup-Verhalten. In Abb. 6.15 ist dokumentiert, daß bei einer gleichverteilt zufälligen Zuordnung der Unternetze zu den Prozessoren auf 8 Workstations immerhin ein Speedup von nahezu 5 erreicht wurde.

Mit dem TSP-Netz ist ein Modell gegeben, bei dem die Menge der aktiven Unternetze nach relativ kurzer Zeit statisch wird. Trotzdem werden die verschiedenen Unternetze zu unterschiedlichen Simulationszeitpunkten generiert, so daß Verfahren zum effizienten Mapping statischer Netze hier nicht anwendbar sind. Es wurde allerdings ein Verfahren zum Lastausgleich an diesem Netz überprüft. Hierbei wurde von zwei unterschiedlichen Szenarien ausgegangen. Zum einen wurde das Lastausgleichsverfahren gestartet, nachdem alle generierten Unternetze zunächst einem einzigen Prozessor zugeteilt worden waren, zum anderen wurden zunächst die Netze wiederum gleichverteilt zufällig an die Prozessoren vergeben, bevor dann das Lastausgleichsverfahren angewendet wurde.

Es zeigt sich (Abb. 6.15), daß das implementierte Lastausgleichsverfahren durchaus das Potential hat, ungünstige Ausgangsverteilungen zu beheben. Trotz des anfänglichen Ein-Prozessor-Mappings wurde mit der Hinzunahme von Prozessoren stets eine weitere Beschleunigung erreicht; auf 8 Prozessoren immerhin noch ein Speedup von knapp 2.6. Die Hoffnung, das Verhalten des Simulators auch bei einem günstigeren initialen Mapping verbessern zu können, erfüllte sich allerdings nicht. Es zeigte sich, daß aufgrund unnötiger Verschiebungen der LPs sogar ein gewisser Zusatzaufwand induziert wird, der sich in einer insgesamt etwas geringeren Beschleunigung der Simulation (4.35 auf 8 Prozessoren) äußert.

Ein Grund für dieses Verhalten kann es sein, daß sich die bei der zufälligen Verteilung einstellende Lastsituation bei diesem Netzmodell schon nahe am Optimum befindet; einen möglichen Indikator hierfür stellt der erreichte hohe Speedup dar. Aus genaueren Lauf-

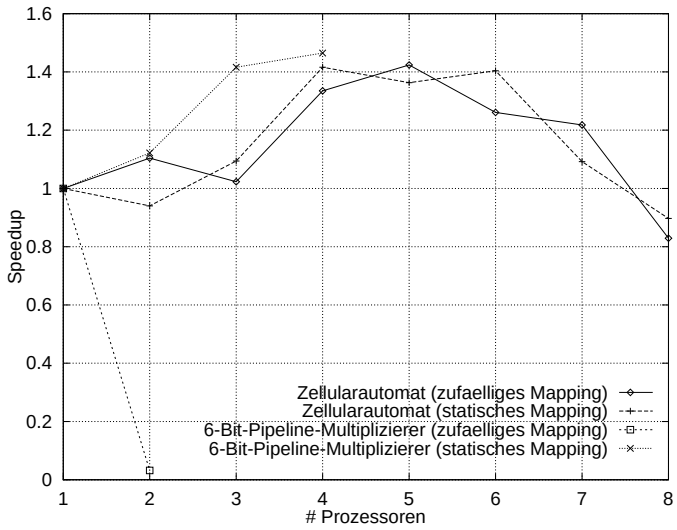


Abbildung 6.16: Auswirkung des statischen Mappings

zeitmessungen (etwa [Ree96, Abb. 7.3]) läßt sich ableiten, daß der Simulator mindestens ca. 22 Sekunden in nicht parallelisierbaren Modellteilen verbraucht (dies läßt sich deshalb leicht ableiten, da alle Parallelarbeit der Unternetze im TSP-Modell zum Simulationszeitpunkt 2 stattfindet). Verglichen mit seiner Laufzeit auf einem Prozessor sind dies immerhin fast 5.5 Prozent. Nach Amdahl's Law [Amd67] liegt damit der maximal erreichbare Speedup auf 8 Prozessoren knapp unter 5.8. Der erreichte Wert von 4.95 stellt also schon ein sehr gutes Ergebnis dar, insbesondere wenn man noch die in Amdahl's Law nicht berücksichtigte Kommunikationsbelastung miteinberechnet.

Für THORN-Modelle, bei denen das Netz statisch ist (d. h. alle Unternetze gleichzeitig beim Simulationsbeginn instantiiert werden), ist oft ein statisches Lastverteilungsverfahrens anwendbar. So konnte etwa das 6-Bit-Pipeline-Multiplizierwerk erst durch Verwendung eines solchen Verfahrens auf mehreren Prozessoren überhaupt sinnvoll ausgeführt werden: war bei einem zufälligen Mapping etwa noch ein Anstieg der Laufzeit beim Wechsel von einem auf zwei Prozessoren von ca. 660 auf über 20000 Sekunden zu beobachten, konnte unter Verwendung des effizienten statischen Mappings die Ausführungszeit auf zwei Prozessoren auf ca. 590 Sekunden gesenkt werden. Damit zeigt sich, daß die verwendeten Lastverteilungsmethoden, die auf durch Beobachtung von Simulationsläufen gewonnenen Verhaltensprofilen der Modelle für geeignete Testdatensätze basieren (siehe [Kös96]), geeignete Heuristiken zur Generierung einer ausgeglichenen Prozessorlast liefern können.

Messungen mit dem Zellularautomatennetz zeigten allerdings keine spürbare Verbesserung durch das Lastteilungsverfahren gegenüber dem zufälligen Mapping. Bei diesem Netz wäre eine effiziente Verteilung des Netzes sicherlich dadurch gegeben, daß das Gesamtmodell bei n Prozessoren in n zusammenhängende Gebiete aufgeteilt würde, die dann an die einzelnen Prozessoren verteilt würden. Das automatische Verfahren liefert allerdings keine solche Aufteilung, sondern partitioniert das Netz in eine größere Anzahl kleiner, zusammenhängender Gebiete. Der notwendige Synchronisationsaufwand ist dabei eher hoch, so daß insgesamt nur ein Speedup von etwas mehr als 1.4 auf 6 Prozessoren zu beobachten

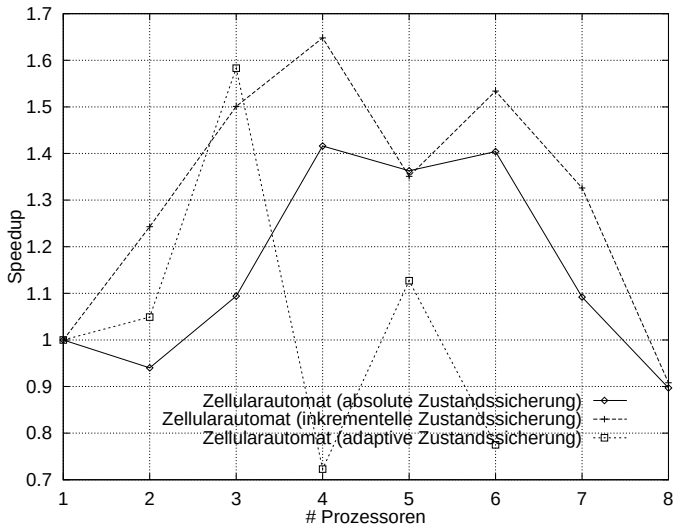


Abbildung 6.17: Auswirkung verschiedener Zustandssicherungsverfahren

ist. Die Auswirkung des statischen Mappingverfahrens auf die Simulation dieser beiden Netzmodelle ist in Abb. 6.16 wiedergegeben.

Gerade das 6-Bit-Pipeline-Multiplizierwerk konnte aufgrund seines großen Zustandsraums mit normaler Zustandssicherung, d. h. Sicherung des kompletten Zustandsraums nach jeder Ereignisbearbeitung, aufgrund von Speichermangel nicht auf mehreren Prozessoren simuliert werden (auf einem Prozessor wurde als Optimierung die Zustandssicherung abgeschaltet). Die in Abb. 6.16 wiedergegebenen Messungen wurden unter Verwendung von Methoden zur inkrementellen Zustandssicherung vorgenommen. Auch hier war der Speicherverbrauch der Simulation noch sehr hoch, so daß aufgrund fehlender Ressourcen das Netz nur auf bis zu 4 Prozessoren ausgeführt wurde. Den Aufwand der Zustandssicherung bei solchen Modellen belegten Messungen, die zeigten, daß die Simulation auf einem Prozessor bei (hier unnötiger) Verwendung der inkrementellen Zustandssicherung im Vergleich zur Laufzeit eines vollständig auf Zustandssicherung verzichtenden Simulators um über 20 Prozent wuchs. Die Beschleunigung des Simulators gegenüber dieser unoptimierten Version beträgt auf 4 Prozessoren immerhin mehr als 1.76, d. h. durch die reine Verteilung ist hier immerhin eine Effizienz von 0.88 zu erreichen. Da aber der durch die Zustandssicherung verursachte Aufwand bei einer sequentiellen Ausführung auf einem Prozessor nicht nötig ist, geben die in Abb. 6.16 dargestellten Werte einen realistischeren Wert für die erzielte Beschleunigung an.

Auch beim Zellularautomatennetz zeigt sich durch den Einsatz inkrementeller Zustandssicherung gegenüber absoluter Zustandssicherung ein signifikanter Fortschritt. Lediglich auf 5 Prozessoren ist gegenüber absoluter Zustandssicherung eine geringfügige Verschlechterung zu sehen (siehe Abb. 6.17). Da jedoch die gemessene Standardabweichung auf 5 Prozessoren ebenfalls auffällig hoch gegenüber den Werten bei allen anderen Prozessorzahlen war, liegt die Vermutung nahe, daß äußere Einflüsse (etwa durch andere Benutzer verursachte Last) für die beobachtete Singularität eher eine Rolle spielen als das Verhalten des Simulators selbst. Für das TSP-Netz wurden keine Messungen mit inkrementeller Zu-

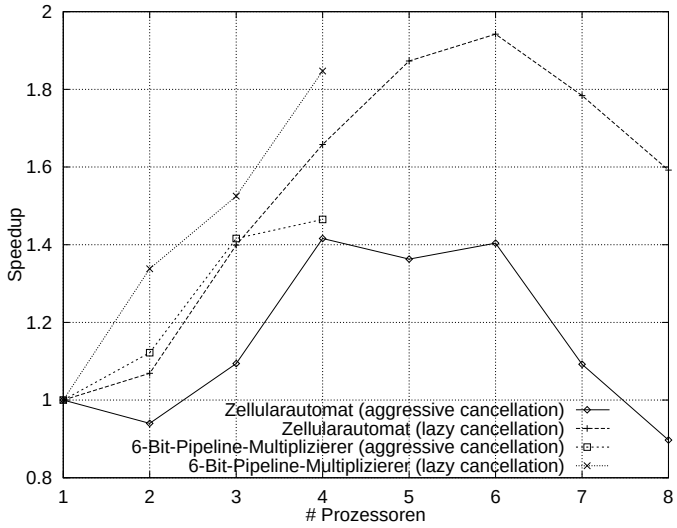


Abbildung 6.18: Auswirkung verschiedener Rollback-Verfahren

standssicherung vorgenommen, da aufgrund des sehr kleinen Zustandsraumes hier keine Verbesserung des Laufzeitverhaltens zu erwarten ist.

Sowohl beim 6-Bit-Pipeline-Multiplizierer als auch beim Zellularautomatennetz wurde das in [FW95] angegebene Verfahren zur adaptiven Bestimmung günstiger Checkpoint-Intervalle eingesetzt. Der hohe Speicherbedarf des 6-Bit-Pipeline-Multiplizierwerks konnte jedoch nicht so weit gesenkt werden, daß eine Simulation auf mehreren Prozessoren möglich war. Das Zellularautomatennetz zeigt zwar bei kleinen Prozessorzahlen (2 und 3) ein günstigeres Verhalten als unter Verwendung des Standardverfahrens zur Zustandssicherung, bei hohen Prozessorzahlen wird der Simulator jedoch deutlich langsamer. Dies könnte daran liegen, daß oft mit Erhöhung der Prozessorzahl auch die Anzahl auftretender Rollbacks zunimmt (siehe etwa [PEWJ92, WHF+92, ELP+92]); diese sind aber bei seltenerer Zustandssicherung teurer. Außer auf zwei Prozessoren führte die adaptive Zustandssicherung nie zu günstigeren Ergebnissen als die inkrementelle (siehe Abb. 6.17).

Für das 6-Bit-Pipeline-Multiplizierwerk und auch für das Zellularautomatennetz wurden des weiteren Messungen zum Vergleich von Aggressive und Lazy-Cancellation durchgeführt. Bei beiden Netzen ist durch die Verwendung von Lazy-Cancellation eine spürbare Verbesserung festzustellen. Insbesondere beim Zellularautomatennetz ist diese sehr hoch. Das entspricht dem zu erwartenden Verhalten: wenn ein Nachbarnetz eines Unternetzes auf einer gemeinsamen Share-Stelle ein Token verspätet ablegt (siehe Abb. 6.14), so muß der entsprechende Unternetz-LP bei Verwendung von Aggressive Cancellation einen Rollback durchführen, auch wenn sich das Netzverhalten durch dieses Token gar nicht ändert. Zumindest die Antinachrichten, die dieser Rollback auslöst, können bei Verwendung von Lazy-Cancellation eingespart werden, möglicherweise werden dadurch dann auch sekundäre Rollbacks vermieden. Der jeweilige Effizienzgewinn der Simulation für die beiden Modelle durch Verwendung von Lazy Cancellation ist in Abb. 6.18 wiedergegeben.

Daß aus der Wirksamkeit einzelner Verfahren nicht unbedingt auch auf ein gutes zeitliches Verhalten bei ihrer Kombination geschlossen werden kann, wurde ebenfalls am Beispiel

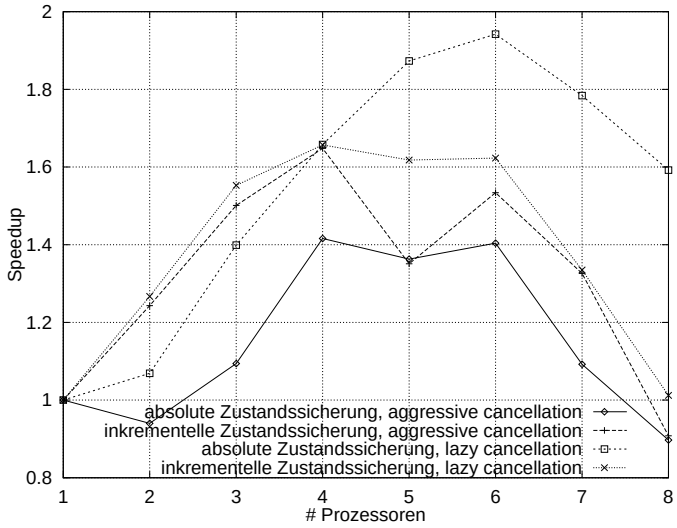


Abbildung 6.19: Kombination inkrementelle Zustandssicherung/Lazy-Cancellation

des Zellularautomatennetzes sichtbar (siehe Abb. 6.19). Obwohl sowohl die inkrementelle Zustandssicherung als auch die Lazy-Cancellation ein jeweils besseres Ergebnis als die absolute Zustandssicherung mit Aggressive-Cancellation erbrachten, konnte die Kombination von Lazy Cancellation und inkrementeller Zustandssicherung nicht unbedingt eine weitere Steigerung bringen. Bis zu vier Prozessoren ist der positive Effekt der inkrementellen Zustandssicherung auch auf das Verfahren mit Verwendung von Lazy-Cancellation zu beobachten. Bei einer größeren Zahl von Prozessoren verlangsamt sich jedoch die Simulation wieder. Der Positiveffekt von Lazy-Cancellation und der bei diesem Modell zu beobachtende Negativeffekt der inkrementellen Zustandssicherung bei größerer Prozessorzahl resultieren darin, daß insgesamt das Kombinationsverfahren Werte zwischen den beiden Einzeloptimierungen annimmt. Daß sich der Wert des Kombinationsverfahrens mit zunehmender Prozessorzahl immer mehr von der reinen Lazy-Cancellation entfernt, ist mit der Theorie konform, daß die Prozessorzahl für die Zahl der Rollbacks eine signifikante Rolle spielt, so daß der Positiveffekt der Lazy-Cancellation kaum noch ins Gewicht fällt.

6.5.2 Bewertung der Simulationsergebnisse

Abb. 6.20 enthält einen Vergleich des optimierten sequentiellen mit dem verteilten Simulator. Man sieht, daß beim TSP-Netz der verteilte Simulator bereits auf einem Prozessor ein besseres Laufzeitverhalten als der sequentielle Simulator zeigt. Dieses eigentlich paradoxe Verhalten hat allerdings eine sehr einfache Begründung: aufgrund gewisser Schwierigkeiten, die beim sequentiellen Simulator mit diesem Modell auftraten, konnte der erzeugte C++-Code nicht mit voller Optimierung des C++-Compilers übersetzt werden. Da der modellinhärente C++-Code bei diesem Netz die Laufzeit der Simulatoren maßgeblich bestimmt, ist der verteilte Simulator, der alle Optimierungen des C++-Compilers nutzen kann, hier wesentlich schneller. Eine Hochrechnung der Laufzeit des sequentiellen Simula-

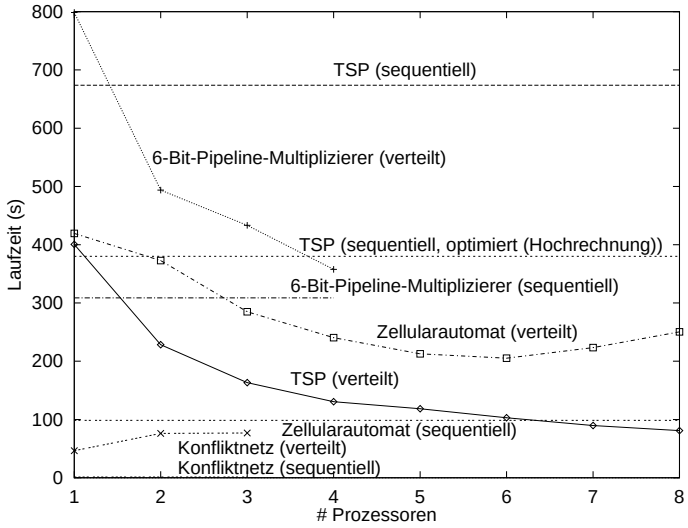


Abbildung 6.20: Vergleich sequentielle und verteilte Simulation

tors für dieses Modell für die höchste Optimierungsstufe aufgrund von Laufzeitvergleichen mit verschiedenen Optimierungsstufen bei ähnlichen Modellen ergibt allerdings eine ungefähre Laufzeit des voll optimierten sequentiellen Simulators für dieses Modell von etwa 380 Sekunden. Dieser Wert liegt erwartungsgemäß unter der Laufzeit der Ein-Prozessor-Version des verteilten Simulators. Bereits auf zwei Prozessoren wird er jedoch durch den verteilten Simulator deutlich unterboten. Für die anderen Netze ist das Verhältnis allerdings weniger günstig. Der 6-Bit-Pipeline-Multiplizierer erreicht immerhin auf 4 Prozessoren fast das Laufzeitverhalten des sequentiellen Simulators. Der Verlauf der Kurve deutet an, daß durch den Einsatz weiterer Prozessoren tatsächlich eine Beschleunigung gegenüber dem sequentiellen Simulator erreicht werden kann. Die verteilte Simulation des Zellularautomatennetzes kann zwar bei Verwendung von bis zu 6 Prozessoren deutlich beschleunigt werden. Hierdurch wird das Verhältnis zwischen verteiltem und sequentiellem Simulator von einer über vierfachen Laufzeit auf nur noch die doppelte Laufzeit gesenkt. Trotzdem bleibt hier der sequentielle Simulator deutlich schneller, die weitere Hinzunahme von Prozessoren bei der verteilten Simulation ändert hieran nichts mehr. Das Konfliktnetz, das bislang noch gar nicht besprochen wurde, zeigt schließlich das erwartete, deutlich ungünstige Verhältnis zwischen sequentieller und verteilter Simulation. Durch keine der implementierten Varianten verteilter optimistischer Simulation konnte eine Beschleunigung des verteilten Simulators bei der Hinzunahme von Prozessoren erreicht werden. Aufgrund des immensen Synchronisationsaufwands und der sehr berechnungsunaufwendigen Unternetze ist der verteilte Simulator hier um ein Vielfaches langsamer als der sequentielle. Wie bereits eingangs erwähnt wurde, gibt ein direkter Vergleich des sequentiellen Simulators mit dem verteilten Simulator, wie er gerade vorgenommen wurde, aufgrund unterschiedlicher Entwicklungszeiten und Optimierungsansätze in dieser Form aber ein verzerrtes Bild der Effizienz verteilter Simulation wieder.

Eine Schwierigkeit stellt etwa die Tatsache dar, daß in der vorliegenden Form des verteilten Simulators ein großer Anteil des Modells, insbesondere die Netzstruktur, interpretiert wird. Diese Vorgehensweise wurde gewählt, um bei der Verteilung einzelner Netzteile auf

die Prozessoren so große Freiheit wie möglich zu haben. In der momentan unterstützten Implementierung wird ein Netzmodell allerdings nur auf Unternetzebene verteilt. Eine naheliegende Optimierung, die allerdings nicht mehr realisiert wurde, ist es, wie im sequentiellen Simulator eine stärkere Übersetzung der einzelnen Unternetze vorzunehmen. Eine weitergehende Maßnahme wäre dann die engere Kopplung aller Unternetze, deren geringer Berechnungsaufwand eine getrennte Behandlung auf einem eigenen Prozessor auf keinen Fall rechtfertigt, zusammen mit dem jeweiligen Aufrufnetz. Auf diese Weise könnte der im verteilten Simulator vorhandene hohe konzeptionelle Aufwand für die Konsistenzhaltung von Share-Stellen, der auch bei Unternetzen auf demselben Prozessor anfällt, vermieden werden. Wie hoch der Laufzeitunterschied zwischen Interpretation und Übersetzung des Netzes tatsächlich sein kann, deutet die Implementierung in [Tho96] an: für ein relativ kleines THORN mit 21 Netzelementen und ohne Hierarchie benötigt die Interpretation mehr als siebenmal soviel Zeit wie die Ausführung eines übersetzten Netzes. In einem anderen Netzmodell ist sogar eine starke, lineare Zunahme des Unterschieds zwischen den Ausführungszeitunterschieden mit der Anzahl der Netzelemente zu sehen. Schon bei 30 Netzelementen ist hier die Interpretation mehr als dreiundzwanzigmal so langsam wie die Ausführung des übersetzten Netzes. Auch wenn sich diese Ergebnisse nicht unmittelbar auf das Verhältnis zwischen verteiltem und sequentiellem Simulator übertragen lassen, da im verteilten Simulator nicht das komplette Netz, sondern nur seine Struktur interpretiert wird, hier allerdings noch zusätzlicher Aufwand zur Konsistenzhaltung von Share-Stellen betrieben werden muß, bieten sie doch einen Anhaltspunkt, wie schwer das Erreichen der Laufzeit des sequentiellen Simulators durch den verteilten Simulator aufgrund der aus den oben genannten Gründen verfolgten Strategie ist. Eine Reimplementierung dieser Teile des verteilten Simulators dürfte für viele Anwendungsfälle, in denen eine Verteilung kleinerer Einheiten, als sie Unternetze darstellen, nicht notwendig ist, wahrscheinlich wesentlich bessere Ergebnisse liefern.

Zu bedenken ist bei den Laufzeitvergleichen auch, daß ein Workstation-Cluster zwar eine ideale Entwicklungsplattform für einen verteilten Simulator dieser Komplexität darstellt, daß er aber mit seiner hohen Prozessorleistung und niedrigen Kommunikationsgeschwindigkeit eine relativ ungeeignete Hardware für verteilte Programme darstellt. Unter Verwendung von PVM liegt etwa die Dauer für das Versenden einer Nachricht bei Nachrichtenlängen bis etwa 100 Byte konstant um 10ms [Ree96, Abb. 4.14]. Da der Hauptteil der bei der verteilten THORN-Simulation verwendeten Nachrichten in diesen Bereich fällt, ist ein Grund für die nur geringe Beschleunigung sicherlich auch in dieser hohen Aufsetzzeit zu sehen.

Daß trotzdem auch in der vorliegenden Implementierung mit einem starken Anteil an Netzinterpretation und auf der gewählten Plattform der verteilte Simulator überhaupt in der Lage ist, die Ausführung des Netzmodells zu beschleunigen und sogar die Leistung des optimierten sequentiellen Simulators teilweise erreicht bzw. überbietet, ist als großer Erfolg zu bewerten. Dennoch fällt an den präsentierten Simulationsergebnissen auf, daß, auch wenn die optimierte Einprozessorversion des verteilten Simulators zugrundelegt wird, abgesehen vom TSP-Netz nur sehr moderate Beschleunigungswerte erreicht werden. Dies liegt sicherlich zum einen in der Natur der Testnetze begründet: das Konfliktnetz ist per se trotz genügend vorhandener Parallelität für eine Verteilung völlig ungeeignet, da alle Unternetze ständig miteinander synchronisiert werden müssen, wofür der Aufwand den erzielten Geschwindigkeitsgewinn durch die Parallelarbeit um Größenordnun-

gen übersteigt. Ein auch nur annäherndes Angleichen an die Laufzeiten des sequentiellen Simulators oder gar ihr Übertreffen durch die verteilte Simulation scheint bei Modellen dieser Art völlig ausgeschlossen. Auch das Zellularautomatennetz hat einen hohen Synchronisationsaufwand; gerade die Tatsache, daß jedes Unternetz mit seinen vier Nachbarn in einem potentiellen Wettstreit um gemeinsame Ressourcen steht, macht eine verteilte Ausführung sehr schwierig. Vor diesem Hintergrund ist die Tatsache, daß bei Verwendung von Lazy-Cancellation überhaupt ein Speedup von über 1.9 erreicht wurde, als äußerst positiv anzusehen. Daß die erzielte Beschleunigung bis auf zu 6 Prozessoren kontinuierlich ansteigt (vgl. Abb. 6.18), kann evtl. ein Indikator dafür sein, daß bei einem schnelleren Kommunikationsnetzwerk eine höhere Beschleunigung erreichbar wäre. Das 6-Bit-Pipeline-Multiplizierwerk erreicht bereits auf 4 Prozessoren einen Speedup von mehr als 1.8; der Verlauf der Kurve deutet ein höheres Beschleunigungspotential an. Trotzdem bildet dieses Beispiel keinen Trivialfall für die verteilte Simulation: mit seinem großen Zustandsraum stellt es gerade für die optimistische Simulation einen schwierigen Testfall dar. Da die insgesamt verwendeten 892 Unternetze außerdem alle in hierarchischer Weise über Share-Stellen miteinander gekoppelt sind, ist der Synchronisationsaufwand in diesem Netz durchaus beträchtlich. Auch hier ist zu vermuten, daß eine Erhöhung der Kommunikationsleistung einen weiteren Vorteil bringt. Daß der Simulator prinzipiell in der Lage ist, die zur Verfügung stehenden Ressourcen einer Parallelrechnerarchitektur gut auszunutzen, zeigt das Beispiel des TSP-Netzes. Hier werden nur wenige Kommunikationen benötigt, so daß die Gesamtleistung im wesentlichen von der durchgeführten Parallelarbeit und weniger von der Kommunikationsleistung bestimmt wird. Allerdings bildet das TSP-Netz mit seiner geringen Anzahl notwendiger Ereignisnachrichten zwischen den LPs auch einen für die verteilte optimistische Simulation sehr gut geeigneten Anwendungsfall, da hier nur wenige Rollbacks auftreten.

Eine genaue Überprüfung der Vermutung, daß die verteilte Simulation auf einem Rechnersystem mit höherer Kommunikationsleistung günstigere Ergebnisse liefern würde, anhand von Testläufen auf einem echten Parallelrechner konnte allerdings aufgrund von Schwierigkeiten bei der Portierung nicht mehr durchgeführt werden.

Kapitel 7

Beobachtung und Steuerung der verteilten Simulation

Eine Fragestellung bei der Simulation komplexer Modelle ist, ob während des Simulationslaufes gewisse Bedingungen erfüllt sind. Hierzu können etwa boolesche Formeln aufgestellt werden, deren Erfülltsein zur Laufzeit der Simulation getestet und geeignet protokolliert werden muß. Das Beobachten von Bedingungen ist nicht nur auf Grundlage des Momentanzustands der Simulation sinnvoll, auch Anfragen an den gesamten bisherigen Simulationslauf, d. h. an die *History* der Zustände, sind zum Überprüfen bestimmter zeitlicher Anforderungen wünschenswert.

Selbst das Beobachten nicht zeitbehafteter globaler Prädikate erweist sich in allgemeinen verteilten Systemen als zu komplex, so daß gewisse Einschränkungen an die zu beobachtenden Bedingungen erforderlich werden. Durch das Überprüfen konsistenter Schnitte [Mat89] können etwa nur sogenannte *stabile* Prädikate (Prädikate, die nie mehr ungültig werden, nachdem sie einmal gültig sind) ermittelt werden [CL85]; Algorithmen, die auch nichtstabile Zustände erkennen, verlangen etwa, daß die Prädikate in lokal auswertbare Teilformeln zerlegt werden können [GW94] oder daß die Bedingungen von der Form $x_1 + x_2 < C$ (x_1, x_2 lokal von jeweils einem Prozeß im verteilten System auswertbar, C Konstante) sein müssen [TG93].

Bei der Beobachtung allgemeiner verteilter Systeme kann allerdings nur von einer partiell geordneten Menge von Ereignissen ausgegangen werden. Im Gegensatz dazu ist die Fragestellung in verteilten optimistischen Simulationssystemen prinzipiell einfacher, da lokale Uhren mit gleicher Metrik vorhanden sind und nicht die Gleichzeitigkeit (im Sinne konsistenter Schnitte) der Ausführung der einzelnen Prozesse, sondern die der simulierten Ereignisse betrachtet werden soll. Es stellt sich hier die Frage, ob ein globaler Zustand zu einer Zeit t (der sich aus den jeweiligen Einzelzuständen der Prozesse zu dieser Zeit zusammensetzt) einem gewissen Prädikat genügt. Dieser Zustand muß durch einen verteilten Algorithmus ermittelt und ausgewertet werden, wobei nicht die Schwierigkeit besteht, kausale Abhängigkeiten zwischen Ereignissen feststellen zu müssen, sondern Bezug auf die lokalen Uhren und Zustandslisten [Jef85] der Prozesse genommen werden kann. Dieser prinzipielle Vorteil optimistischer Simulationssysteme bietet dann evtl. auch einen Ansatz zur Beobachtung quantitativer temporaler Prädikate, die sich in allgemeinen verteilten Systemen als sehr schwierig erwiesen hat [CM93].

Es ergibt sich in optimistischen Simulationssystemen bei der Beobachtung globaler Prädikate allerdings zusätzlich das Problem des eventuellen Zurückrollens von Prozeßzuständen. Es ist beispielsweise möglich, daß der Zustand einer Menge von Prozessen eine bestimmte zu beobachtende Bedingung erfüllt, daß diese Menge von Prozessen (oder eine Teilmenge hiervon) jedoch aufgrund des Time-Warp-Protokolls einen Rollback durchführen muß, so daß das Beobachten der Bedingung evtl. nur aufgrund einer (in diesem Fall) zu optimistischen Simulationsstrategie zustande kam. Hier darf für die Frage der Gültigkeit von Bedingungen nicht nur der Prozeßzustand herangezogen werden, sondern es muß vielmehr auch berücksichtigt werden, ob dieser Zustand *sicher* ist, d. h. ob er aufgrund der Kenntnis über die globale virtuelle Zeit als nicht mehr zurückrollbar eingestuft werden kann.

Im Rahmen einer Diplomarbeit [Rei96] wurden Algorithmen entwickelt und implementiert, um das Erfülltsein von Bedingungen in verteilten optimistischen Simulationssystemen zu testen. Hierzu wurde eine allgemeine Time-Warp-Simulationsbibliothek entwickelt, bei dem die Modelle in C++ formuliert werden. Es können dann boolesche Funktionen formuliert werden, die mit Hilfe prädikaten- und temporallogischer Quantoren Bedingungen an die Zustandsvariablen des Modells stellen. Die Unterstützung temporallogischer Quantoren wurde allerdings nicht implementiert.

Die Sprache zur Formulierung der Bedingungen soll hier nicht formal dargestellt werden, sondern es soll lediglich eine allgemeine Beschreibung mit einigen Beispielen gegeben werden.

Ein zu beobachtender Zustand wird als boolesche Bedingung über Zeitausdrücken, Variablennamen und Operatoren formuliert. Hierbei wird ein Zeitausdruck durch den Term `T(Zeitpunkt, Prozeßausdruck)` angegeben. `Zeitpunkt` kann dabei eine der folgenden Formen haben:

- Er kann eine Zahl sein, die die Simulationszeit angibt, an dem `Prozeßausdruck` gültig sein soll.
- Er kann das Schlüsselwort `all` sein, womit ausgedrückt wird, daß die Bedingung, die durch `Prozeßausdruck` angegeben wird, für alle Simulationszeitpunkte gültig sein soll.
- Er kann ein Ausdruck `?var` sein, womit nur die Existenz eines Zeitpunktes, an dem die durch `Prozeßausdruck` angegebene Bedingung gültig ist, gefordert wird. Die Variable `var` enthält dann im Rest des Ausdrucks den jeweiligen Wert (dies ist vom Prinzip der Freeze-Quantification [AH91] abgeleitet).

Bei der letzten Form kann auch als Kurzform nur `Prozeßausdruck` angegeben werden, wobei dann allerdings der Zeitpunkt der Gültigkeit des Teilausdrucks nicht im Rest der Formel zur Verfügung steht.

`Prozeßausdruck` ist wiederum eine boolesche Formel, die neben Variablennamen und Operatoren Ausdrücke der Form `P(Prozeßnummer, Formel)` enthalten darf. Die `Formel` kann Operatoren sowie Funktions- und Variablennamen, die im jeweiligen Modell gültig sind, enthalten. `Prozeßnummer` ist ähnlich wie `Zeitpunkt` aufgebaut:

- Sie kann eine Zahl sein, die die logische Nummer des zu betrachtenden LPs angibt.
- Sie kann das Schlüsselwort `all` sein, wenn die Bedingung für alle LPs gelten soll.
- Sie kann von der Form `?var` sein, womit die Existenz (mindestens) eines LPs gefordert ist, der die Bedingung erfüllt. Seine logische Nummer steht für den Rest des Teilausdrucks in `var` zur Verfügung.

Einige Beispiele sollen formulierbare Bedingungen illustrieren.

- $\texttt{P}(17, x > y + 5)$
  Ist irgendwann in LP 17 die Variable x größer als $y + 5$?
- $\texttt{P}(?n, x > y + 5) \wedge n < 50$
  Gibt es irgendwann einen LP mit logischer Nummer kleiner als 50, in dem die Variable x größer als $y + 5$ ist?
- $\texttt{P}(\texttt{all}, x > y + 5)$
  Ist irgendwann in allen LPs gleichzeitig die Variable x größer als $y + 5$?
- $\texttt{P}(17, x) > \texttt{P}(19, y)$
  Ist irgendwann in LP 17 die Variable x größer als die Variable y in LP 19?
- $\texttt{P}(?n, x) > \texttt{P}(n + 13, y + 5)$
  Gibt es irgendwann einen LP p, dessen Variable x größer als $y + 5$ in LP $p + 13$ ist?
- $(\texttt{P}(?n, x) = \texttt{P}(?m, y)) \wedge (n \neq m)$
  Gibt es irgendwann zwei verschiedene LPs, in denen die Variable x des einen den gleichen Wert wie die Variable y des anderen hat?
- $\texttt{T}(13, \texttt{P}(?n, x > y + 5))$
  Gibt es einen LP, bei dem zum Zeitpunkt 13 die Variable x größer als $y + 5$ ist?
- $\texttt{T}(\texttt{all}, \texttt{P}(?n, x > y + 5))$
  Ist immer in irgendeinem LP die Variable x größer als $y + 5$?
- $\texttt{T}(\texttt{all}, \texttt{P}(?n, x > y + 5) \vee \texttt{now}() < 40 \vee \texttt{now}() > 60)$
  Ist im Zeitraum von Simulationszeit 40 bis 60 stets in irgendeinem LP die Variable x größer als $y + 5$?
- $\texttt{T}(?t, \texttt{P}(?n, x)) = \texttt{T}(t + 3, \texttt{P}(n, y))$
  Gibt es einen Zeitpunkt t, so daß in irgendeinem LP die Variable x zum Zeitpunkt t den gleichen Wert hat wie die Variable y zum Zeitpunkt $t + 3$?
- $\texttt{T}(?t, \texttt{P}(?n, x)) = \texttt{T}(t + 3, \texttt{P}(?m, y))$
  Gibt es einen Zeitpunkt t, so daß in irgendeinem LP die Variable x zum Zeitpunkt t den gleichen Wert hat wie in irgendeinem anderen LP die Variable y zum Zeitpunkt $t + 3$?

Für diese Sprache (allerdings ohne die Temporalquantoren) wurde ein Beobachtungsalgorithmus in einem verteilten optimistischen Simulator implementiert. Der Beobachtungsalgorithmus wurde an die GVT-Ermittlung gekoppelt, auf diese Weise konnte das Protokollieren nur sicherer Zustände gewährleistet werden. Die Implementierung zeigte, daß eine Beobachtung globaler Modellzustände auch bei der verteilten Simulation prinzipiell möglich ist. Der zusätzliche Nachrichtenaufwand des Algorithmus ist sehr gering, wenn die zu beobachtenden Teilausdrücke den einzelnen LPs bereits beim Simulationsbeginn bekannt sind, da Zustandsänderungen quasi „huckepack“ mit den notwendigen GVT-Nachrichten an den zentralen Auswerter übertragen werden können.

Eine Steuerung der verteilten Simulation ist mit einem solchen Verfahren allerdings nicht realisierbar, da sich die Beobachtung in diesem Fall ja immer auf die bereits durch die GVT überholte Vergangenheit der Simulation bezieht. Soll eine solche Beobachtungs- und Steuerungskomponente in einen verteilten Simulator integriert werden, so muß dies als eigenständiger, mit dem übrigen Simulationssystem gemäß dem Time-Warp-Verfahren interagierender logischer Prozeß geschehen. Eine Implementierung eines solchen Verfahrens wurde allerdings aus Zeitgründen nicht mehr vorgenommen.

Wie bereits eingangs erwähnt, wurde dieses Verfahren im Rahmen einer Diplomarbeit in einem speziellen Simulator, d. h. nicht im verteilten THORN-Simulator implementiert. Dies hatte rein pragmatische Gründe, u. a. wäre ein direktes Integrieren der Beobachtungsverfahren in den THORN-Simulator wegen dessen Komplexität wesentlich aufwendiger geworden, ohne bzgl. der Verfahren oder ihrer Implementierung selbst neue Erkenntnisse zu bringen.

Im Rahmen einer als Lehrveranstaltung an der Universität Oldenburg durchgeführten Projektgruppe [SSW+96] wurde allerdings für THORNs eine Beobachtungssprache angegeben, mit Hilfe derer bestimmte Bedingungen über das Netzmodellverhalten (Stellenbelegungen, Transitionsaktivierungen etc.) formuliert werden können und auf die dann in geeigneter, definierbarer Weise reagiert wird (genaueres siehe Kapitel 9). Dies wird dadurch realisiert, daß eine Umsetzung von *Netzausdrücken* (Ausdrücken über dem Netzzustand) in die internen Zustandsvariablen des Simulators vorgenommen wird; der Simulator protokolliert dann sein Verhalten während des Laufes in geeigneter Weise. Obwohl in der Lehrveranstaltung eine Anbindung an den sequentiellen Simulator realisiert wurde, kann auch in gleicher Weise eine Abbildung der Netzausdrücke in die Zustandsvariablen des verteilten Simulators vorgenommen werden. Mit einem Algorithmus, wie er in [Rei96] implementiert wurde, kann dann auch eine Beobachtung des Netzzustandes während der verteilten Simulation erfolgen. Da damit die Beobachtung des Netzzustandes bei der verteilten Simulation kein prinzipielles Problem mehr darstellte und da andere Aufgabenstellungen (siehe Kapitel 6) für wichtiger erachtet wurden, wurde auf eine Implementierung verzichtet.

Kapitel 8

Hybride höhere Netze

Ausgehend von den Erfahrungen mit THOR-Netzen bei der Modellierung hybrider Systeme wurde im zweiten Projektabschnitt die Netzklasse der hybriden höheren Netze – kurz HYNETZE – definiert. Die grundlegende Entwurfsentscheidung für die neue Netzklasse bestand darin, eine bewährte Methode zur Modellierung diskreter Systeme in geeigneter Form zu erweitern, so daß auch kontinuierliche Zustandsänderungen angemessen beschrieben werden können. Diese Entscheidung ist im Vergleich zu anderen hybriden Modellierungssprachen eher ungewöhnlich, da diese meist auf einem kontinuierlichen Ansatz basieren, der um diskrete Elemente erweitert wurde.

HYNETZE integrieren drei etablierte Modellierungsmethoden in einer Sprache: *High-level Petri Netze* (THOR-Netze) repräsentieren die diskrete Grundlage zur Beschreibung diskreter Systemteile, *Differentialgleichungen und Zustandsgleichungen* dienen zur Spezifikation von kontinuierlichem Systemverhalten und *objektorientierte Konzepte* verbessern die Ausdruckskraft und Kompaktheit der Modelle.

In diesem Kapitel wird der neue Ansatz zur Modellierung und Simulation hybrider Systeme zunächst informell beschrieben und an einem Beispiel illustriert. Anschließend werden die Werkzeuge vorgestellt, die zur Erstellung und Untersuchung von HYNETZ-Modellen entwickelt und implementiert wurden.

8.1 Informelle Beschreibung

HYNETZE sind höhere zeitbeschriftete Netze, deren diskreter Netzteil auf dem Formalismus der THOR-Netze basiert. Als Ausgangsbasis wurden flache THOR-Netze gewählt, um die komplexen Vorgänge in hybriden Modellen zunächst an einer möglichst einfachen Klasse höherer Netze zu untersuchen. Die Konzepte zur hierarchischen Strukturierung lassen sich später wieder in den Modellierungsansatz integrieren.

Die Erweiterung des diskreten Teils von THOR-Netzen um kontinuierliche Elemente wurde maßgeblich von den Konzepten hybrider Petrinetze [DA92] beeinflußt. Die Ausdrucksmöglichkeiten gehen aber weit über die hybrider Petrinetze hinaus, da HYNETZE einerseits die Modellierung komplexer Objekte erlauben und andererseits kontinuierliche Zustandsänderungen mit Hilfe von gewöhnlichen Differentialgleichungen und Zustandsgleichungen be-

schrieben werden können. Im folgenden werden diese Gleichungen als *Differential-Zustands-Gleichungen* (DZGL) bezeichnet.

Um die Modellierung kontinuierlicher Systemteile mit DZGL möglichst homogen in den THORN-Formalismus zu integrieren, wurde ein neuer Typ von *kontinuierlichen Transitionen* eingeführt, bei dem die Schaltaktion mit DZGL beschriftet werden kann. Ähnlich wie bei hybriden Petrinetzen können durch das Schalten kontinuierlicher Transitionen Objekte bzw. deren Attribute auf den umliegenden Stellen kontinuierlich verändert werden.

Im Gegensatz zu hybriden Petrinetzen wurde für HYNETZE kein neuer Typ kontinuierlicher Stellen definiert, auf dem Objekte kontinuierlich geändert werden. Statt dessen wurde unter Berücksichtigung des Objektkonzepts ein neuer *elementarer Typ* `Real` eingeführt. Objekte und Objektattribute dieses Typs können durch kontinuierliche Transitionen und deren Gleichungen kontinuierlich verändert werden. Objekte, die mindestens ein Attribut vom Typ `Real` haben, werden als *kontinuierliche Objekte* bezeichnet. Diese Definition kommt ohne eine zusätzlichen Stellentyp aus und bietet wesentlich mehr Modellierungsmöglichkeiten.

Erste Ideen zur Erweiterung höherer Petrinetze um kontinuierliche Elemente wurden in [WS95] veröffentlicht. Diese Definition sah allerdings nur konstante Änderungsraten für kontinuierliche Zustandsvariablen vor und war damit in Bezug auf Beschreibung kontinuierlicher Systeme ähnlich ausdrucksstark wie hybride Petrinetze. Weiterentwicklungen dieser Ideen führten schließlich zur aktuellen Definition von HYNETZEN, wie sie in diesem Abschnitt präsentiert wird (vgl. dazu auch [Wie96b] und [Wie96c]).

Ein HYNETZ besteht aus einer Netzstruktur, der Beschriftung von Netzelementen mit verschiedenen Attributen und einer Beschreibung von selbstdefinierten komplexen Objekttypen. Bevor die einzelnen Netzelemente und ihre Beschriftungen näher beschrieben werden, wird zunächst die neu entwickelte Beschriftungssprache für HYNETZE vorgestellt. Anschließend folgt eine Beschreibung des Schaltverhaltens von HYNETZEN, durch das das Zusammenspiel zwischen diskreten und kontinuierlichen Netzteilen festgelegt wird.

8.1.1 Die Beschriftungssprache

Die Definition einer neuen Beschriftungssprache für HYNETZE wurde in erster Linie erforderlich, weil C++ direkt keine Möglichkeiten bietet, Differentialgleichungen oder Zustandsgleichungen zu spezifizieren. Da mit der neuen objektorientierten Beschriftungssprache sowohl diskrete als auch kontinuierliche Zustandsänderungen ausgedrückt werden können, erhielt sie den Namen **H**ybrid **O**bject-Oriented Labeling **La**nguage (HOLA).

HOLA ist eine einfache objektorientierte Sprache, die sich in ihrer Syntax und Semantik sehr stark an C++ [Str92] orientiert. Dadurch ist eine Umstellung für THORN-Modellierer relativ problemlos und auch eine Übersetzung von HOLA nach C++ läßt sich ohne größeren Aufwand realisieren. Letzteres ist deshalb von Bedeutung, damit die bestehenden Werkzeuge ohne gravierende Anpassungen an eine neue Beschriftungssprache weiter verwendet werden können.

HOLA ist verglichen mit C++ eine wesentliche einfachere Sprache. Beispielsweise werden zahlreiche Konzepte von C++ wie Templates, Zeiger und Mehrfachvererbung in HOLA

nicht mehr angeboten, da sie zur Beschriftung von HyNetzen nicht erforderlich sind. Statt dessen verfügt Hola über eine Referenzsemantik (alle Variablen sind Referenzen auf Objekte), einen Algorithmus zur Garbage Collection, eine Klassenbibliothek für komplexe Datenstrukturen und ein Laufzeitinformationssystem, mit dem die Zugehörigkeit von Objekten zu Klassen festgestellt werden kann.

Ein wesentliches Ziel beim Entwurf von Hola war die Integration von Sprachelementen, mit denen Differential-Zustands-Gleichungen (DZGL) bestehend aus gewöhnlichen Differentialgleichungen erster Ordnung in expliziter Form (8.1), Hilfsgleichungen (8.2) und Zustandsgleichungen (8.3) möglichst einfach und intuitiv verständlich spezifiziert werden können.

$$\mathbf{x}' = \mathbf{f}(\mathbf{x}, \mathbf{z}, \mathbf{u}, t) \tag{8.1}$$

$$\mathbf{z} = \mathbf{g}(\mathbf{x}, \mathbf{z}, \mathbf{u}, t) \tag{8.2}$$

$$\mathbf{x} = \mathbf{F}(\mathbf{x}, \mathbf{z}, \mathbf{u}, t) \tag{8.3}$$

In dieser für die Modellierung kontinuierlicher Systeme üblichen mathematischen Schreibweise repräsentiert $\mathbf{x}$ den Vektor der kontinuierlich veränderten Zustandsvariablen, $\mathbf{z}$ einen Vektor von Hilfsvariablen, $\mathbf{u}$ einen Vektor von Eingangsvariablen bzw. Parameter und t die Zeit. $\mathbf{f}$, $\mathbf{g}$ und $\mathbf{F}$ sind Funktionen, mit denen die erste Ableitung der Zustandsvariablen nach der Zeit $\mathbf{x}' = d\mathbf{x}/dt$, die Hilfsvariablen bzw. die Zustandsvariablen direkt berechnet werden. Mathematisch gesehen sind Zustandsvariablen stetige Funktionen der Zeit, häufig wird ihre Abhängigkeit von der Zeit allerdings bei der Modellierung dynamischer Systeme implizit vorausgesetzt, so daß die Gleichungen in obiger Form notiert werden.

Zur Spezifikation solcher Gleichungen verfügt Hola über

- einen zusätzlichen elementaren Typ `Real`, mit dem kontinuierliche Zustandsvariablen definiert werden können (`Real` ist eine Unterklasse von `Double`),
- einer globalen Variablen `Time`, durch die die Zeit in einem Modell repräsentiert wird,
- und dem Symbol `'`, mit dem die erste Ableitung einer Zustandsvariablen gekennzeichnet werden kann.

Mit diesen Sprachelementen lassen sich die gewünschten Gleichungen in Hola ähnlich wie oben formulieren. Beispielweise wird durch

```
x' = 2 * x;
```

eine Differentialgleichung spezifiziert, mit der ein exponentielles Wachstum einer Zustandsvariablen beschrieben wird. Und durch die Zustandsgleichung

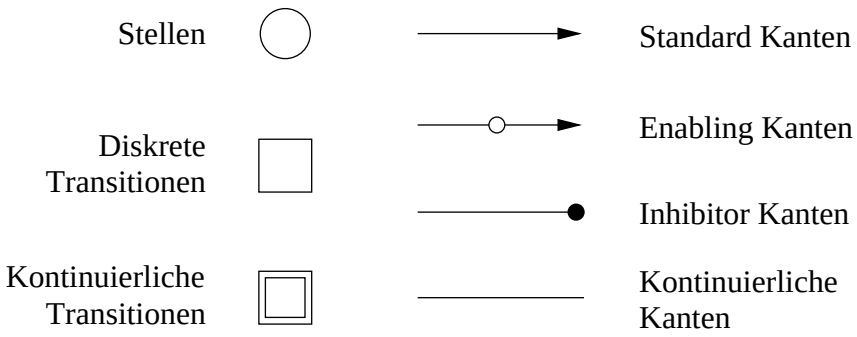
```
x = sin( Time );
```

kann ein sinusförmiger Verlauf einer Zustandsvariablen beschrieben werden.

Insgesamt ist HOLA eine Sprache, die speziell auf die Anforderungen der Modellierung mit hybriden höheren Netzen angepaßt ist, syntaktisch ähnlich zu C++ und konzeptionell mit Java vergleichbar ist. Weiterführende Informationen zum Entwurf und den Konzepten von HOLA sowie eine formelle Sprachdefinition und Angaben zur Implementierung eines Compilers sind in [Maj96] dokumentiert.

8.1.2 Die Netzelemente und ihre Beschriftung

In diesem Abschnitt werden die Netzelemente von HYNETZEN und ihre Beschriftungen beschrieben. Da HYNETZE auf der Basis von THOR-Netzen entwickelt wurden, lassen sich viele Ähnlichkeiten mit dieser Netzklasse feststellen. Dies gilt insbesondere für den diskreten Teil. Die kontinuierlichen Erweiterungen in HYNETZEN beschränken sich auf einen neuen Typ von Transitionen, einen neuen Kantentyp und einen zusätzlichen elementaren Objekttyp. Im Gegensatz zu anderen hybriden Petrinetzen konnte durch die Einführung des kontinuierlichen Objekttyps auf kontinuierliche Stellen verzichtet werden. Abbildung 8.1.2 zeigt alle Netzelemente zusammen mit ihrer graphischen Darstellung.



Im folgenden werden die einzelnen Komponenten und ihre Beschriftungen näher betrachtet.

8.1.2.1 Objekte

Objekte sind Instanzen von Klassen der objektorientierten Beschriftungssprache HOLA. Die Klassen dieser Sprache werden als *Objekttypen* bezeichnet. Die Spezifikation von Objekttypen erfolgt in ähnlicher Form wie bei THOR-Netzen. Allerdings braucht sich der Modellierer hier nicht mehr selbst um die Ein- und Ausgabe bzw. die textuelle Repräsentation von Objekten kümmern.

In HYNETZEN gibt es zusätzlich zu den elementaren Objekttypen `Int`, `Long`, `Float`, `Double`, `Char`, `String`, `Bool` und `Token` den speziellen Typ `Real`, mit dem kontinuierlich veränderbare Objekte bzw. Objektattribute von anderen Objekten unterschieden werden. Technisch gesehen ist `Real` eine Unterklasse von `Double`. Nur Objekte oder Objektattribute dieses Typs können durch DZGL kontinuierlich verändert werden. Objekte, die

mindestens ein Attribut vom Typ `Real` besitzen, werden daher als **kontinuierliche Objekte** bezeichnet.

8.1.2.2 Stellen

Stellen werden in HyNetzen ähnlich beschriftet wie in Thor-Netzen. Sie haben einen *Namen*, einen *Typ* und eine *Kapazität*. Der Name einer Stelle kann beliebig gewählt werden – er ist für das Verhalten eines Netzes unbedeutend. Der Typ legt fest, welche Objekte auf der Stelle abgelegt werden dürfen. Er wird durch den Namen eines elementaren oder selbstdefinierten Objekttyps spezifiziert. Auf einer Stelle sind nur solche Objekte zugelassen, deren Typ mit dem Stellentyp übereinstimmt oder einer davon abgeleiteten Klasse entspricht. Die Kapazität definiert eine obere Grenze für die Anzahl der Objekte, die sich auf einer Stelle befinden dürfen. Sie kann den Wert einer positiven ganzen Zahl oder den unbeschränkten Wert `Omega` annehmen.

Alle Stellen verwalten ihre Objekte als Mengen. Sie entsprechen damit den Multiset-Stellen der Thor-Netze. Spezielle Stellenstrukturen sind in der ersten Version der Definition von HyNetzen nicht vorgesehen. Im Vergleich zu einfachen hybriden Petrinetzen [DA92] werden die Stellen in HyNetzen nicht in diskrete und kontinuierlichen Stellen unterteilt. Diese Unterscheidung erfolgt hier bereits auf Objektebene und bietet dadurch wesentlich flexiblere Modellierungsmöglichkeiten.

8.1.2.3 Kanten

Standard-, Enabling- und Inhibitor-Kanten werden ohne Änderungen von Thor-Netzen übernommen (vgl. Abschnitt 3.1.3). Consuming-Kanten gibt es in HyNetzen nicht mehr, da sie kaum benötigt werden und in ähnlicher Form durch andere Konstruktionen zu ersetzen sind.

Zusätzlich zu diesen Kantentypen wird eine neue **kontinuierliche Kante** eingeführt, die mit einem *Namen* und einem *Variablennamen* beschriftet wird. Kontinuierliche Kanten sind ungerichtet und können nur zwischen Stellen und kontinuierlichen Transitionen auftreten (vgl. Abbildungen 8.1.2 und 8.1). Durch eine kontinuierliche Kante wird angezeigt, daß die Objekte auf der Stelle durch die kontinuierliche Transition verändert werden. Dabei können die Objekte auf den Stellen wie bei Standard- und Enabling-Kanten über den Variablennamen der Kante in der Transitionsbeschriftung referenziert werden.

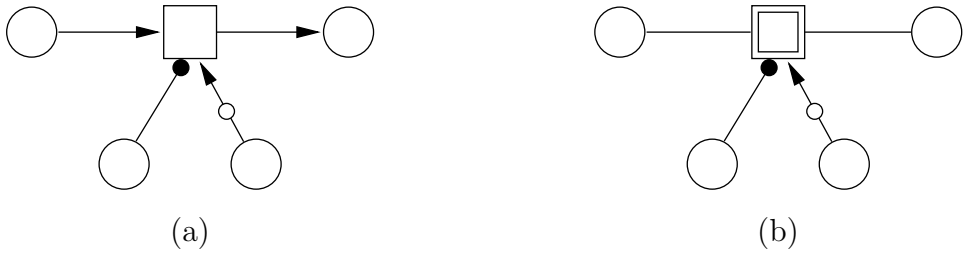


Abbildung 8.1: Erlaubte Kantenverbindungen zwischen Stellen und diskreten (a) und kontinuierlichen (b) Transitionen

Kontinuierliche Kanten passen sich auf diese Weise homogen in die bestehenden Konzepte ein. Sie sind ungerichtet, da sie im Gegensatz zu Standard-Kanten keine Objekte transportieren, sondern Objektwerte kontinuierlich verändern ohne die Objekte dabei von den Stellen zu entfernen (s. a. Abschnitt 8.1.2.5 und 8.1.3.1).

8.1.2.4 Diskrete Transitionen

Die diskreten Transitionen in HyNetzen entsprechen den Transitionen der Thor-Netze. Sie werden mit einem *Namen* einer *Schaltbedingung*, einer *Schaltaktion*, einer *Verzögerungszeit*, einer *Schaltdauer* und einer *Schaltkapazität* beschriftet. Für die Beschriftungen gelten im wesentlichen dieselben Vereinbarungen wie für Thorn-Transitionen. Allerdings erfolgt ihre Spezifikation hier in Form von gültigem Hola-Code.

In den Beschriftungen können die Kantenvariablen inzidenter Kanten in derselben Weise verwendet werden wie bei Thorn-Transitionen. Neu ist, daß nun auch auf die globale Variable für die aktuelle Modellzeit `Time` lesend in den Transitionsbeschriftungen zugegriffen werden kann. Nähere Informationen hierzu werden im Abschnitt 8.1.3.2 über das Schaltverhalten diskreter Transitionen angegeben.

8.1.2.5 Kontinuierliche Transitionen

Die Modellierung von kontinuierlichen Zustandsänderungen in einem System erfolgt in HyNetzen mit Hilfe von kontinuierlichen Transitionen. Die grundlegende Idee bei dieser neuen Transitionsart ist, daß sie reellwertige Objekte bzw. Objektattribute durch ihr Schalten kontinuierlich verändert, ohne die Objekte von ihren Stellen zu entfernen. Dieses Verhalten ist eine Erweiterung des Schaltverhaltens von kontinuierlichen Transitionen in einfachen hybriden Netzen [DA92, TK93] auf höhere Netze.

Kontinuierliche Transitionen werden mit einem *Namen* einer *Schaltbedingung* und einer *Schaltaktion* beschriftet. Der Name kann wie bei allen Netzelementen beliebig gewählt werden – er ist für das Verhalten eines Netzes unbedeutend. Die Schaltbedingung einer kontinuierlichen Transition wird wie bei diskreten Transitionen als seiteneffektfreier, boolescher Hola-Ausdruck über den Variablen eingehender Kanten und der Variablen `Time` formuliert – kontinuierliche Kanten werden in diesem Zusammenhang sowohl als eingehende als auch als ausgehende Kanten betrachtet. Mit der Schaltbedingung können wie üblich zusätzliche Bedingungen an die Aktiviertheit einer Transition gestellt werden.

Entscheidend für das Verhalten einer kontinuierlichen Transition ist ihre Schaltaktion. Die Schaltaktion kann dazu mit gewöhnlichen Differentialgleichungen erster Ordnung, Hilfsgleichungen und Zustandsgleichungen in der in Abschnitt 8.1.1 beschriebenen Form beschriftet werden. In den Gleichungen dürfen die Variablen der inzidenten kontinuierlichen Kanten lesend und schreibend und die Variablen der inzidenten Enabling-Kanten sowie die globale Variable `Time` nur lesend verwendet werden; zusätzlich können lokale Hilfsvariablen definiert werden. Bei der Spezifikation der Gleichungen ist zu beachten, daß durch die Variable auf der linken Seite einer Differential- oder Zustandsgleichung immer ein Objekt oder Objektattribut vom Typ `Real` referenziert wird, da nur solche Objekte kontinuierlich verändert werden können. Aus Vereinfachungsgründen wird außerdem vorausgesetzt, daß

sich die Gleichungen in einer ausführbaren Reihenfolge befinden und keine algebraischen Schleifen enthalten. HOLA-Anweisungen wie z. B. die `if`- oder `for`-Anweisung dürfen in der Schaltaktion einer kontinuierlichen Transition nicht vorkommen.

Kontinuierliche Transitionen werden nicht mit einer Zeitbeschriftung oder Schaltkapazität versehen. Eine Zeitbeschriftung macht für diesen Transitionstyp keinen Sinn, da diese Transitionen kontinuierlich schalten und dabei automatisch Zeit verbrauchen. Und die Schaltkapazität wird für alle kontinuierlichen Transitionen als unbeschränkt angenommen. Durch diese Definition schaltet jede kontinuierliche Transition mit allen aktivierten Schaltelementen parallel. Weitere Einzelheiten zum Schaltverhalten kontinuierlicher Transitionen werden im folgenden Abschnitt beschrieben.

8.1.3 Das Schaltverhalten

In diesem Abschnitt wird das Schaltverhalten von HYNETZEN näher erläutert. Dazu werden zunächst die Aktiviertheitsbedingungen und das Schalten von diskreten und kontinuierlichen Transitionen betrachtet. Danach wird das Zusammenspiel zwischen den verschiedenen Transitionen durch die Schaltregel für HYNETZE definiert und anhand von Beispielen illustriert. Zunächst werden allerdings einige grundlegende Begriffe geklärt, die in diesem Zusammenhang von Bedeutung sind.

Das Ausführen bzw. das Schalten eines HYNETZES bedeutet die Veränderung des Netzzustands. Die Art und Weise, wie die Zustandsänderungen durchgeführt werden, wird durch die Schaltregel festgelegt. Sie bestimmt, unter welchen Bedingungen Transitionen aktiviert sind und welche Auswirkungen das Schalten von Transitionen auf den Zustand hat. In höheren Petrinetzen wird die Schaltregel üblicherweise nicht für Transitionen sondern für Schaltereignisse formuliert. Ein Schaltereignis besteht dabei aus einer Transition und einer Belegung der Variablen eingehender Kanten mit Objekten der entsprechenden Vorbereichsstellen (vgl. Abschnitt 3.1.4).

Da in HYNETZEN das Schalten kontinuierlicher Transitionen nicht in zwei Ereignisse für den Schaltbeginn und das Schaltende unterteilt werden kann, ist der Begriff des Schaltereignisses hier nicht mehr angemessen. Aus diesem Grund wird die Kombination einer Transition mit einer Variablenbelegung eingehender Kanten im folgenden als *Schaltelement* bezeichnet. Schaltelemente werden in der Form $[\, t :< v_1 = o_1,\ \ldots,\ v_n = o_n >]$ notiert, wobei t eine Transition ist, v_i Variablen eingehender Kanten sind und o_i Objekte der jeweiligen Vorbereichsstellen repräsentieren, die an die Variablen gebunden sind.

Der *Zustand* eines HYNETZES wird durch den aktuellen Zeitpunkt, die Markierung der Stellen und ihre reservierten Plätze und durch die aktuellen Restverzögerungs- und Restschaltzeiten diskreter Schaltelemente eindeutig bestimmt.

Im Gegensatz zu THOR-Netzen ist der aktuelle Stand der globalen Uhr (`Time`) Bestandteil des Netzzustands. Dieser Zeitpunkt kann aufgrund der kontinuierlichen Zeitskala, die HYNETZEN zugrundeliegt, beliebige nichtnegative Werte annehmen. Die Markierung der Stellen besteht aus einer Menge individueller Objekte. Dabei werden auch Objekte mit gleichen Attributwerten (beispielsweise durch eine eindeutige Identifikationsnummer) voneinander unterschieden. Deshalb werden die Objekte auf den Stellen nicht als Multimengen verwaltet, wie es z. B. in CP-Netzen der Fall ist.

8.1.3.1 Kontinuierliche Transitionen

Beim Schalten kontinuierlicher Transitionen werden im Gegensatz zu dem üblichen Verhalten von (diskreten) Transitionen keine Objekte konsumiert oder produziert. Statt dessen werden die Objekte bzw. deren Attribute direkt auf den adjazenten Stellen kontinuierlich verändert. In diesem Abschnitt werden Einzelheiten zur Aktiviertheit und zum Schalten kontinuierlicher Transitionen erläutert und an Beispielen illustriert.

Ein Schaltelement einer kontinuierlichen Transition ist in einem Zustand *aktiviert*, wenn

- alle inhibitorisch wirkenden Stellen im Vorbereich leer sind,
- an alle Variablen der eingehenden Kanten ein Objekt gebunden werden kann und
- die Schaltbedingung der Transition durch die Variablenbelegung der eingehenden Kanten erfüllt wird.

Die zweite Bedingung ist eine kürzere Formulierung dafür, daß die Vorbereichsstellen, die über kontinuierliche Kanten mit der Transition verbunden sind, nicht leer sind und daß Stellen, die über aktivierende Kanten mit der Transition verbunden sind, mindestens soviele Objekte enthalten wie durch das Gewicht der verbindenden Kante spezifiziert wird. Dabei wird vorausgesetzt, daß alle Kanten mit einem Variablennamen beschriftet sind und die Anzahl der Variablen einer Kante durch ihr Gewicht bestimmt wird (s. o.). Bei dem dritten Punkt ist zu beachten, daß in der Schaltbedingung der Transition auch Bedingungen an die globale Uhr gestellt werden können, welche ebenfalls erfüllt sein müssen, um ein Schaltelement zu aktivieren.

Beim *Schalten* von aktivierten Schaltelementen werden die Objekte, die an die Variablen kontinuierlicher Kanten gebunden sind, entsprechend den Gleichungen der Schaltaktion (vgl. Abschnitt 8.1.2.5) kontinuierlich verändert, ohne die Objekte von den Stellen entfernen. Da eine kontinuierliche Transition eine unbegrenzte Schaltkapazität hat, schaltet sie mit allen aktivierten Schaltelementen parallel.

Jedes kontinuierliche Schaltelement schaltet so lange wie es aktiviert ist. Beendet wird die Aktiviertheit nur dann,

- wenn durch das Schalten einer diskreten Transition eine inhibitorisch wirkende Stelle mit einem Objekt belegt oder ein an dem Schaltelement beteiligtes Objekt entfernt wird oder
- wenn durch das Schalten kontinuierlicher Transitionen bzw. das Fortschreiten der globalen Zeit die Schaltbedingung für das Schaltelement nicht mehr erfüllt ist.

8.1.3.2 Diskrete Transitionen

Diskrete Transitionen haben im Grunde dasselbe Schaltverhalten wie die Transitionen in Thor-Netzen. Kleinere Unterschiede ergeben sich aufgrund der fehlenden Stellenstrukturen und Consuming-Kanten. Etwas größere Auswirkungen auf das Schaltverhalten haben die Verwendung der globalen Variablen `Time` in den Transitionsbeschriftungen und

der Einfluß kontinuierlicher Transitionen. Daher werden hier die wesentlichen Aktiviertheitsbedingungen und Zustandsänderungen diskreter Schaltelemente noch einmal unter besonderer Berücksichtigung dieser Neuerungen zusammengefaßt.

Ein Schaltelement einer diskreten Transition erfüllt in einem Zustand die *Vorbereichsbedingung*, wenn

- alle inhibitorisch wirkenden Stellen im Vorbereich leer sind,
- an alle Variablen der eingehenden Standard- und Enabling-Kanten ein Objekt gebunden werden kann und
- durch die Variablenbelegung dieser Kanten die Schaltbedingung der Transition erfüllt wird – man beachte, daß in der Schaltbedingung auch Bedingungen an die globale Zeit gestellt werden können.

Alle Schaltelemente, die in einem Zustand die Vorbereichsbedingung erfüllen, werden *verzögert*. Die Verzögerungszeit für ein Schaltelement wird zu Beginn der Verzögerung durch Auswerten des entsprechenden Ausdrucks der Transition ermittelt. Falls die Variable `Time` in dem Ausdruck enthalten ist, so wird der aktuelle Wert der Variablen bei der Auswertung verwendet.

Schaltelemente, die über den Zeitraum der Verzögerung (oder länger) ununterbrochen die Vorbereichsbedingung erfüllen, heißen *vollständig verzögert*. Gründe für den Abbruch der Verzögerung eines Schaltelements sind dann gegeben,

- wenn durch das Schalten einer diskreten Transition eine inhibitorisch wirkende Stelle mit einem Objekt belegt oder ein an dem Schaltelement beteiligtes Objekt entfernt wird oder
- wenn durch das Schalten kontinuierlicher Transitionen bzw. das Fortschreiten der globalen Zeit die Schaltbedingung für das Schaltelement nicht mehr erfüllt ist.

Die Verzögerung eines Schaltelements wird hiermit unter den selben Umständen abgebrochen wie die Aktiviertheit eines kontinuierlichen Schaltelements (s. o.). Man beachte, daß die Verzögerung eines Schaltelements auch durch das Fortschreiten der globalen Zeit abgebrochen werden kann.

Wenn ein Schaltelement irgendwann nach dem Abbruch der Verzögerung die Vorbereichsbedingung wieder erfüllt, so startet die Verzögerungszeit von vorn. Dabei wird auch der entsprechende Ausdruck noch einmal ausgewertet, d. h. die bisherige Verzögerung eines Schaltelements bleibt unberücksichtigt!

Ein diskretes Schaltelement ist in einem Zustand *aktiviert*, falls

- es vollständig verzögert ist,
- die Transition über freie Schaltkapazität verfügt und

- auf den Stellen im Nachbereich entsprechend den Gewichten ausgehender Kanten ausreichend Platz zur Verfügung steht.

Ein aktiviertes diskretes Schaltelement kann mit dem Schalten beginnen. Das Schalten erfolgt wie bei THOR-Netzen in zwei Teilen. Beim *Schaltbeginn* werden folgende Aktionen durchgeführt:

1. Die an dem Schaltelement beteiligten Objekte werden aus dem Vorbereich abgezogen bzw. für Enabling-Kanten kopiert und die ausgehenden Objekte werden mit ihrem Standard-Konstruktor erzeugt – Typ und Anzahl der Objekte ergeben sich aus dem Typ der jeweiligen Nachbereichsstelle und dem Gewicht der verbindenden Kante.
2. Die Schaltdauer für das Schaltelement wird durch Auswerten der Schaltdauerbeschriftung der Transition berechnet – bei dieser Berechnung können Werte der an dem Schaltelement beteiligten Vorbereichsobjekte und die globale Zeit berücksichtigt werden.
3. Die Schaltaktion der Transition wird mit den in Punkt 1 bereitgestellten Objekten ausgeführt. In der Schaltaktion selbst können weitere (Hilfs-) Objekte definiert werden, die unter Beachtung der Typkompatibilität auch an Variablen ausgehender Kanten gebunden werden dürfen.
4. Auf den Nachbereichsstellen wird entsprechend den Kantengewichten ausgehender Kanten ausreichend Platz reserviert.
5. Die Schaltkapazität wird dekrementiert.

Beim Auswerten der Schaltdauerbeschriftung und Ausführen der Schaltaktion wird der zum Schaltbeginn aktuelle Wert der Variablen `Time` verwendet, falls diese Variable in den Beschriftungen vorkommt.

Zum Zeitpunkt des *Schaltendes* (Zeitpunkt des Schaltbeginns + Schaltdauer) wird der Schaltvorgang für das Schaltelement abgeschlossen:

1. Die Schaltkapazität wird inkrementiert.
2. Die reservierten Plätze auf den Nachbereichsstellen werden wieder freigegeben.
3. Die beim Schaltbeginn berechneten Ausgangsobjekte werden im Nachbereich abgelegt.

Für Schaltelemente, deren Schaltdauer Null beträgt, werden die Aktionen des Schaltendes direkt durchgeführt, d. h. die Punkte 4 und 5 vom Schaltbeginn und die Punkte 1 und 2 von Schaltende fallen weg.

8.1.3.3 Schaltregel

Nachdem in den vorherigen Abschnitten das Schaltverhalten einzelner Transitionen beschrieben wurde, wird hier nun das Zusammenspiel zwischen diskreten und kontinuierlichen Transitionen erläutert und in Form der Schaltregel für HyNetze definiert. Durch die Schaltregel wird die Semantik von HyNetzen festgelegt. Sie basiert wie die Schaltregel für Thor-Netze auf der Bearbeitung von Schaltelementen und den daraus resultierenden Änderungen des Zustands. Die Schaltregel ist wie folgt definiert.

Zu jedem Zeitpunkt der Ausführung eines HyNetzes wird aus der Menge der diskreten Schaltelemente, die zu dem Zeitpunkt aktiviert sind oder ihren Schaltvorgang beenden, sukzessive ein Schaltelement nichtdeterministisch ausgewählt und die entsprechenden Schaltbeginn- bzw. Schaltende-Aktionen durchgeführt (s. o.). Dieser Vorgang wird so lange wiederholt, bis die Menge der zu bearbeitenden diskreten Schaltelemente leer ist. Es ist zu beachten, daß durch das Ausführen von Schaltbeginn- bzw. Schaltende-Aktionen andere Schaltelemente aktiviert oder deaktiviert werden können und daß dabei keine Zeit verbraucht wird. Die Bearbeitung diskreter Schaltelemente entspricht damit im wesentlichen der Maximum-Sofort-Schaltregel, wie sie bereits für Thor-Netze definiert wurde (vgl. Abschnitt 3.2.1).

Existieren zu dem Zeitpunkt aktivierte kontinuierliche Schaltelemente, so schalten sie alle gemeinsam bis ein diskretes Schaltelement aktiviert wird oder das Schalten beendet. Dabei werden die Werte der in den aktivierten Schaltelementen gebundenen Objekte entsprechend den Schaltaktionen der Transitionen kontinuierlich verändert (vgl. Abschnitt 8.1.3.1). Es ist zu beachten, daß sich die Menge der aktivierten kontinuierlichen Schaltelemente beim Ausführen der Schaltelemente verändern kann und daß dabei Zeit verbraucht wird – die globale Modellzeit `Time` schreitet dementsprechend voran. Falls zu dem Zeitpunkt keine kontinuierlichen Schaltelemente aktiviert sind, wird nur die Modellzeit bis zum nächsten diskreten Ereignis kontinuierlich verändert.

Bei der Ausführung eines HyNetzes wechseln sich diskrete Phasen, in denen keine Zeit verbraucht wird, mit kontinuierlichen Phasen, in denen die Zeit voranschreitet, ab (siehe Abb. 8.2). Entsprechend der oben beschriebenen Schaltregel werden kontinuierliche Phasen sofort beendet, sobald ein diskretes Schaltelement zu bearbeiten ist. Die Priorität diskreter Schaltelemente ist dabei unbedingt erforderlich, damit bei der Ausführung keine diskreten Ereignisse verpaßt werden.

Ein HyNetz ist tot, wenn in einem Zustand

- kein Schaltelement mehr aktiviert ist und kein diskretes Schaltelement mehr schaltet und
- kein Schaltelement zu einem späteren Zeitpunkt mehr aktiviert werden kann – sei es durch den Ablauf einer Verzögerungszeit oder der Abhängigkeit von der globalen Modellzeit.

Die hier beschriebene Schaltregel ist eine maximale Schaltregel, da sowohl in den diskreten als auch in den kontinuierlichen Phasen maximale Mengen von Schaltelementen bearbeitet werden.

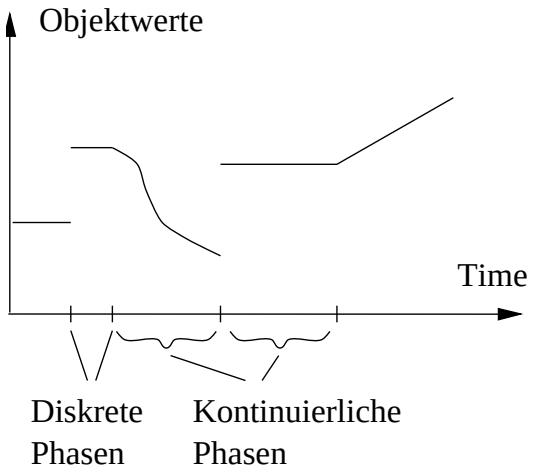


Abbildung 8.2: Phasen der Ausführung eines HyNetzes

8.1.4 Beispiel

In diesem Abschnitt wird das Schaltverhalten von HyNetz-Modellen an einem kleinen Beispiel einer Heizungssteuerung illustriert.

Heizungsanlagen, die über einen Thermostat gesteuert werden, werden häufig zur Illustration der Fähigkeiten hybrider Modellierungsansätze verwenden (vgl. [ACHH93]). In diesen Systemen wird in der Regel die Temperatur in einem Gebäude durch einen Sensor kontinuierlich erfaßt und in Abhängigkeit der Werte eine Heizungsanlage ein- bzw. ausgeschaltet. Charakteristisch für solche Systeme ist, daß die diskreten Zeitpunkte, an denen die Temperatur einen bestimmten Schwellenwert erreicht, nicht im voraus feststehen, sondern direkt von der kontinuierlichen Zustandsvariablen für die Temperatur abhängen. Umgekehrt wird der Verlauf dieser Variablen durch das An- und Abschalten der Heizung maßgeblich beeinflußt.

Hier soll als Beispiel ein System mit einem nichtlinearen Temperaturverlauf betrachtet werden, in dem die Temperatur in einem Gebäude auch von der Außentemperatur beeinflußt wird. Ein Thermostat sorgt dafür, daß die Innentemperatur immer zwischen 18 und 20 °C gehalten wird. Der Wärmeverlust des Gebäudes wird durch die Differentialgleichung $i' = -c_1\,(i-a)$ beschrieben (i und a repräsentieren dabei die Innen- bzw. Außentemperatur und c_1 ist ein konstanter Faktor, der im wesentlichen von der Isolierung des Gebäudes abhängt). Der Erwärmungsprozeß kann ebenfalls mit einer einfachen Differentialgleichung $i' = c_2$ modelliert werden, wobei c_2 die Leistung der Heizung repräsentiert. In Abbildung 8.3 wird ein HyNetz-Modell für das hier beschriebene System dargestellt (`SB` steht für die Schaltbedingung, `SA` für die Schaltaktion, `SD` für die Schaltdauer, `VZ` für die Verzögerungszeit und `SK` für die Schaltkapazität einer Transition). Die Innentemperatur beträgt in der Ausgangssituation 15 °C und die Außentemperatur 5 °C.

In dem dargestellten Zustand sind zunächst nur die beiden kontinuierlichen Schaltelemente $[\mathit{Heizung}: < i = 15.0 >]$ und $[\mathit{Wärmeverlust}: < a = 5.0,\ i = 15.0 >]$ aktiviert. Entsprechend dem in Abschnitt 8.1.3.1 und 8.1.3.3 beschriebenen Schaltverhalten für

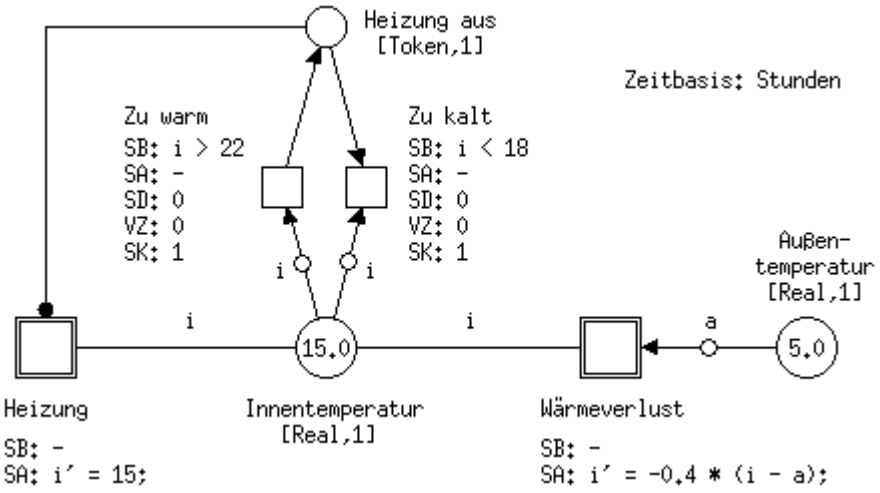


Abbildung 8.3: HyNetz-Modell einer Temperaturregelung

kontinuierliche Transitionen schalten beide Schaltelemente gemeinsam bis ein diskretes Schaltelement aktiviert wird oder die Schaltbedingung für die Schaltelemente nicht mehr erfüllt ist. Dabei verändern sie das Objekt für die Innentemperatur gemeinsam – wird ein kontinuierliches Objekt gleichzeitig durch mehrere Transitionen verändert, so werden die einzelnen Änderungen summiert. Das Temperatur-Objekt ändert sich dementsprechend in der ersten kontinuierlichen Phase gemäß der Differentialgleichung $i' = 15 - 0.4\,(i - a)$.

Nach ca. 44 Minuten hat die Innentemperatur etwas mehr als 22 °C erreicht und die diskrete Transition *Zu warm* ist aktiviert. Durch das Schalten dieser Transition wird zum selben Zeitpunkt ein Token auf der Stelle *Heizung aus* produziert und die Transition *Heizung* damit deaktiviert. Da keine weiteren diskreten Schaltelemente zu diesem Zeitpunkt zu bearbeiten sind, wird mit der zweiten kontinuierlichen Phase fortgefahren. In dieser Phase schaltet nur die Transition *Wärmeverlust*. Nach weiteren 40 Minuten ist die Innentemperatur auf unter 18 °C abgefallen und die Heizung wird durch das Schalten der diskreten Transition `Zu kalt` wieder eingeschaltet, usw. Abbildung 8.4 zeigt den für eine geregelte Heizungsanlage typischen Temperaturverlauf.

Dieses Modell läßt sich an verschiedenen Stellen erweitern. Beispielsweise kann die Außentemperatur in Abhängigkeit der Tageszeit verändert werden. Dazu ist lediglich eine weitere kontinuierliche Transition erforderlich, deren Schaltaktion mit einer entsprechenden Zustandsgleichung beschriftet ist – oder diese Gleichung wird direkt in die Transition *Wärmeverlust* integriert.

8.2 Werkzeuge

Zur Erstellung, Bearbeitung und Auswertung von HyNetz-Modellen ist ein integriertes Werkzeug entwickelt worden, das aus einem graphischen Netzeditor und einem Netzinterpreter besteht. Abweichend von der ursprünglichen Planung, einen effizienten, compilativen Simulator zu implementieren, wurde es vorgezogen, einen interpretativen Simulator

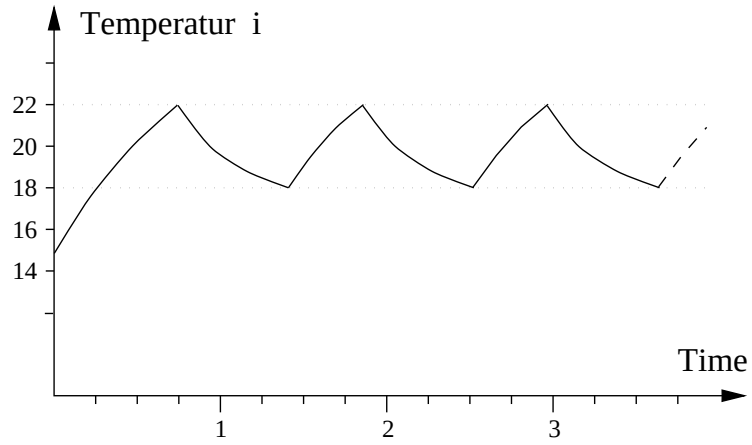


Abbildung 8.4: Temperaturverlauf in dem Gebäude

zu entwickeln, um mehr Möglichkeiten bezüglich der Visualisierung von Simulationen zu haben (vgl. Abschnitt 8.2.2).

8.2.1 Der Netzeditor

Der interaktive, graphische Netzeditor für HyNetze basiert im wesentlichen auf dem Netzeditor für Thor-Netze (vgl. Abschnitt 4.2). Er wurde in bezug auf die neuen Netzelemente (kontinuierliche Kanten und kontinuierliche Transitionen) angepaßt und in bezug auf die Eingabe und Darstellung von Objekten und Objekttypen verbessert (s. a. [End96]).

Der Modellierer wird durch den HyNetz-Editor bei der Eingabe komplexer Objekttypen durch einen speziellen graphischen Editor unterstützt. Komplexe Objekttypen brauchen dadurch nicht mehr komplett in textueller Form eingegeben werden. Außerdem ist die Spezifikation von Methoden zur Ein- und Ausgabe von Objekten durch den Modellierer nicht mehr erforderlich. Insgesamt ist der Editor in dieser Beziehung wesentlich benutzerfeundlicher geworden und es wurde eine Fehlerquelle ausgeschlossen. Abbildung 8.5 zeigt den Editor zur Eingabe und Darstellung von komplexen Objekttypen.

In ähnlicher Form wird die Eingabe und Darstellung komplexer Objekte durch einen speziellen Editor unterstützt. Dadurch kann der Benutzer leichter erkennen, welche Werte zu welchen Attributen eines komlexen Objekts gehören.

Ansonsten unterscheidet sich der HyNetz-Editor von dem Thorn-Editor nur bezüglich der Animation eines Netzes (s. u.).

8.2.2 Der Netzinterpreter

Zur Auswertung von HyNetz-Modellen wurde wie bei Thor-Netzen ein Simulator entwickelt und prototypisch implementiert. Basierend auf den Erfahrungen mit den relativ langen Übersetzungszeiten bei einer Thor-Netzcompilation, die im wesentlichen auf den

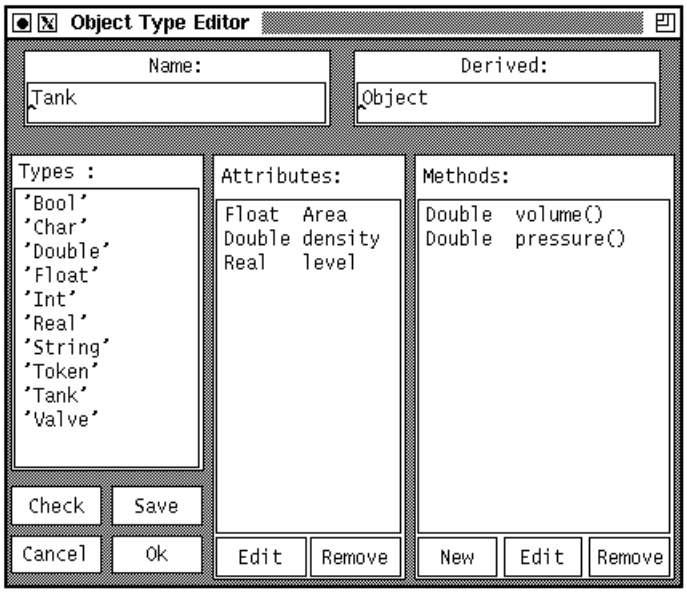


Abbildung 8.5: Editor für Objekttypen

GNU C++-Compiler zurückzuführen sind, und den fehlenden Animationsmöglichkeiten bei einer Netzsimulation wurde es vorgezogen, zunächst einen Interpreter zu entwickeln. Die Vorteile dieser Simulationsart liegen vor allem in einer einfacheren Validierung von Netzmodellen und besseren Möglichkeiten bei Visualisierung der Ausführung eines Netzes. Die Vorgänge in einem Modell können mit einem Interpreter wesentlich transparenter dargestellt werden. Außerdem können Fehler in der Netzbeschriftung dadurch direkt erkannt und mit sinnvollen Fehlermeldungen angezeigt werden. Der wesentliche Nachteil einer interpretativen Simulation ist die schlechtere Ausführungsgeschwindigkeit.

Der Netzinterpreter für HyNetze arbeitet ähnlich wie der sequentielle Simulator für Thor-Netze. Anstelle einer globalen Liste verwaltet er jedoch mehrere Listen für schaltende, verzögerte und aktivierte Transitionen (vgl. Abbildung 8.6), die er entsprechend der in Abschnitt 8.1.3.3 beschriebenen Schaltregel abarbeitet.

Unterschiede zum sequentiellen Thorn-Simulator bestehen in den folgenden Punkten:

- Der Interpreter verwaltet alle Schaltelemente zu einer Transition. Dadurch kann er dem Benutzer zu jedem Zeitpunkt der Ausführung Auskunft über verzögerte oder aktivierte Schaltelemente geben (s. u.). Allerdings benötigt er für Transitionen mit vielen Belegungsmöglichkeiten wesentlich mehr Speicher.
- Für den Interpreter werden keine Methoden zur Bestimmung der Aktiviertheit und zum Schalten einer Transition erzeugt. Hier wird das Schaltverhalten einschließlich der Hola-Beschriftungen interpretiert. Bei der Überprüfung der Aktiviertheit von Schaltelementen wird wie beim sequentiellen Simulator der Einflußbereich einer Transition berücksichtigt.

Zur Ausführung kontinuierlicher Phasen einer HyNetz-Simulation werden numerische

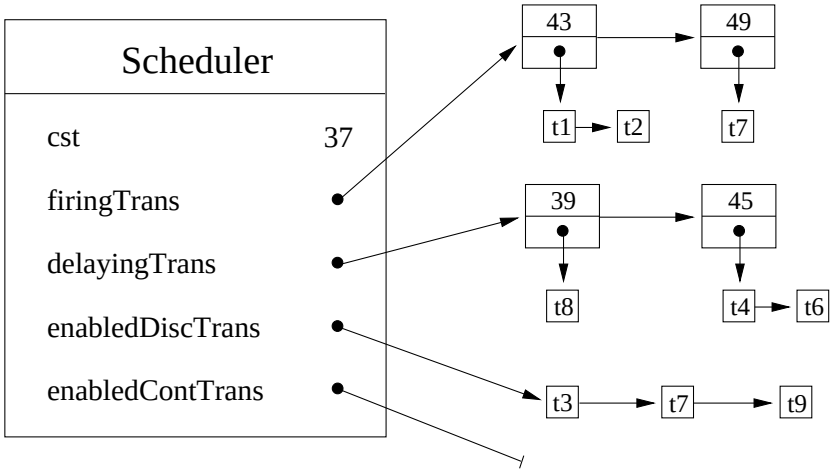


Abbildung 8.6: Aufbau des Netzinterpreters

Integrationsverfahren eingesetzt, mit denen das Netzverhalten approximiert wird. Der Modellierer kann dabei derzeit zwischen drei Verfahren mit unterschiedlicher Genauigkeit wählen: Euler-Cuchy, Runge-Kutta 2 (3) und Runge-Kutta 4 (5). Des weiteren kann er durch weitere Parameter, wie z. B. den relativen Fehler, die Ausführungsgenauigkeit kontinuierlicher Phasen steuern.

Da der Netzinterpreter direkt in das Editorsystem integriert ist, bestehen verschiedene Möglichkeiten zur interaktiven Steuerung und zur Animation von Simulationsläufen (vgl. Abbildung 8.7):

- Über ein Steuerungsfenster (Interpreter Control) kann die Ausführung eines Netzes interaktiv gesteuert werden. Unter anderem können einzelne Schritte durchgeführt, kontinuierliche oder diskrete Phasen beendet oder die Simulation einfach nur gestartet oder gestoppt werden.
- Während der Ausführung eines Netzes wird die Anzahl der Objekte auf einer Stelle durch schwarze Punkte (Token) direkt in der Netzdarstellung des Netzeditors angezeigt. Zusätzlich besteht die Möglichkeit, einzelne Objektwerte beispielsweise in Form eines Balkendiagramms in einem speziellen Fenster (Place Observer) darzustellen.
- Zu jeder Transitionen können zu jedem Zeitpunkt der Ausführung die Schaltelemente (Occurence Elements) angezeigt werden, mit denen die Transition aktiviert ist oder schaltet. Grapisch wird der Zustand einer Transition (Verzögerung, Schaltbeginn, Schalten, Schaltende) zusätzlich durch kleine Dreiecke in der Transition dargestellt.

Durch diese Steuerungs- und Visualisierungsmöglichkeiten der HyNetz-Werkzeuge lassen sich HyNetz-Modelle leicht validieren und demonstrieren. Detaillierte Informationen zur Ausführung und Animation von HyNetzen sind in [Tho96] und [End96] beschrieben.

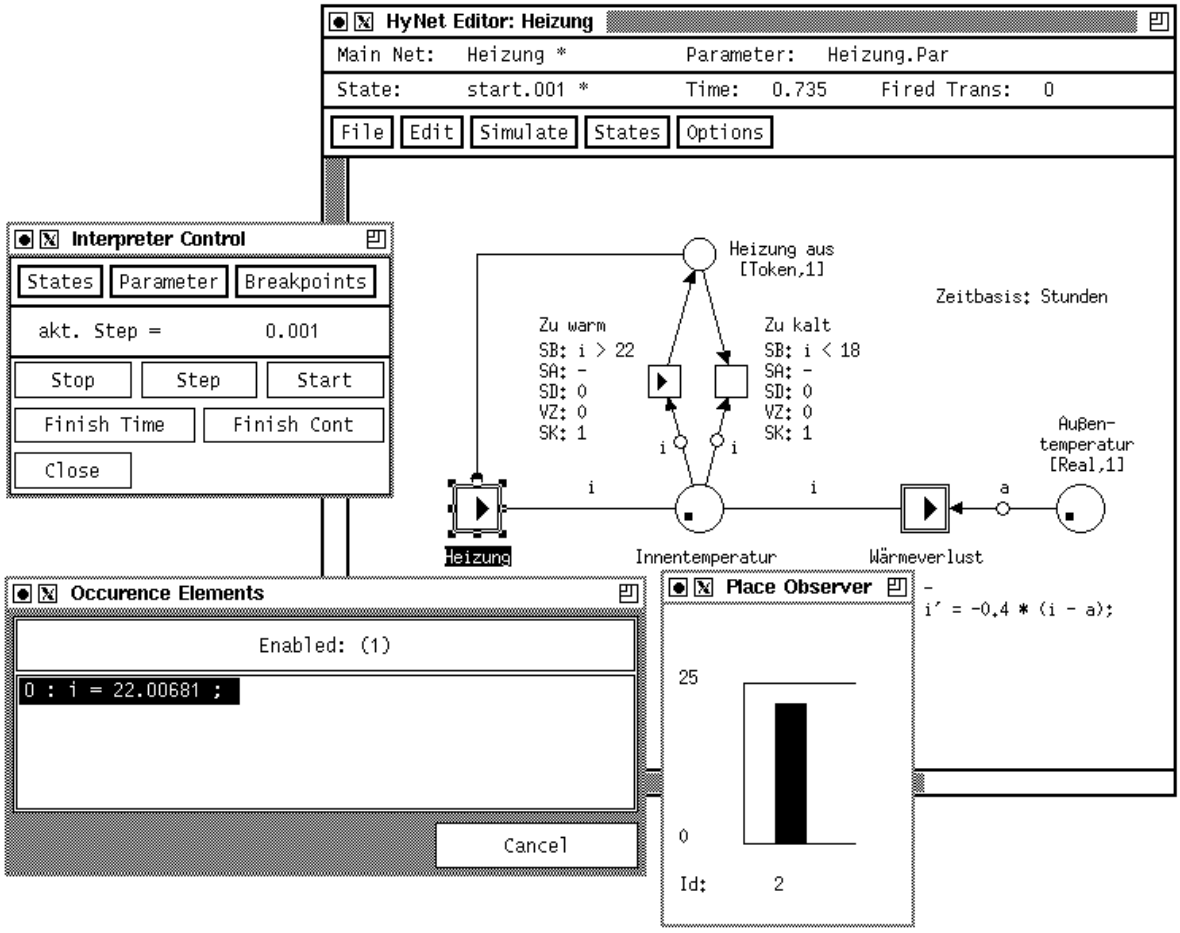


Abbildung 8.7: HyNetz-Werkzeuge

Die Ausführung oder Überprüfung eines Netzes oder eines Netzteils durch den Netzinterpreter kann direkt ohne Übersetzung des Netzes erfolgen. Allerdings erfolgt sie wesentlich langsamer als bei einem compilativen Simulator. Beispielsweise führt der Interpreter für sehr einfache Netze auf einer Sun SPARCstation 10/40 durchschnittlich 1.800 diskrete Schaltungen oder 2.200 kontinuierliche Schritte pro Minute durch. Im Vergleich mit dem sequentiellen Thorn-Simulator ist er damit erwartungsgemäß deutlich langsamer. Für umfangreiche Simulationen eines Modells wäre es daher zu empfehlen, auch einen effizienten Simulator für HyNetze zu entwickeln.

Eine umfassende Beschreibung von HyNetzen und ihrer Simulation sowie ein Vergleich mit anderen hybriden Modellierungsansätzen ist in [Wie97] zu finden.

Kapitel 9

Erweiterte Simulationsumgebung

In einer Projektgruppe [SSW+96], die über ein Jahr begleitend zum Projekt DNS als Lehrveranstaltung an der Universität Oldenburg durchgeführt wurde, waren zehn Studenten daran beteiligt, die Entwicklungs- und Ausführungsumgebung für die THORN-Simulation zu erweitern. Dabei lagen die Schwerpunkte auf der interaktiven Steuerung der Simulation sowie der Visualisierung von Simulationsläufen. An dieser Stelle sollen nicht alle Ergebnisse der Projektgruppe vorgestellt werden, sondern lediglich auf diese beiden Schwerpunkte eingegangen werden.

Die Simulation eines Netzmodells wird über ein Kontrollfenster gestartet und gestoppt

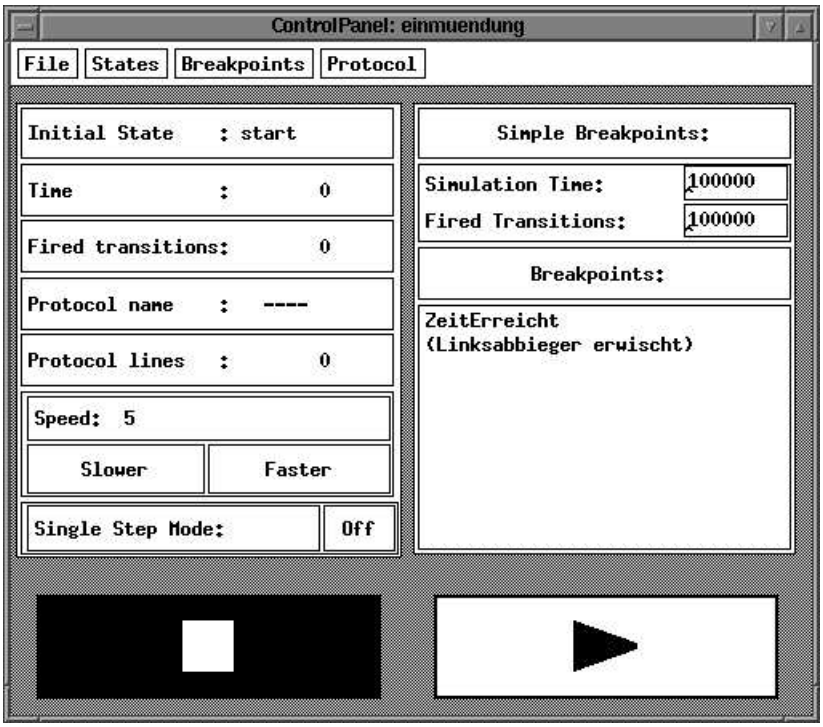


Abbildung 9.1: Kontrollfenster zur Simulationssteuerung

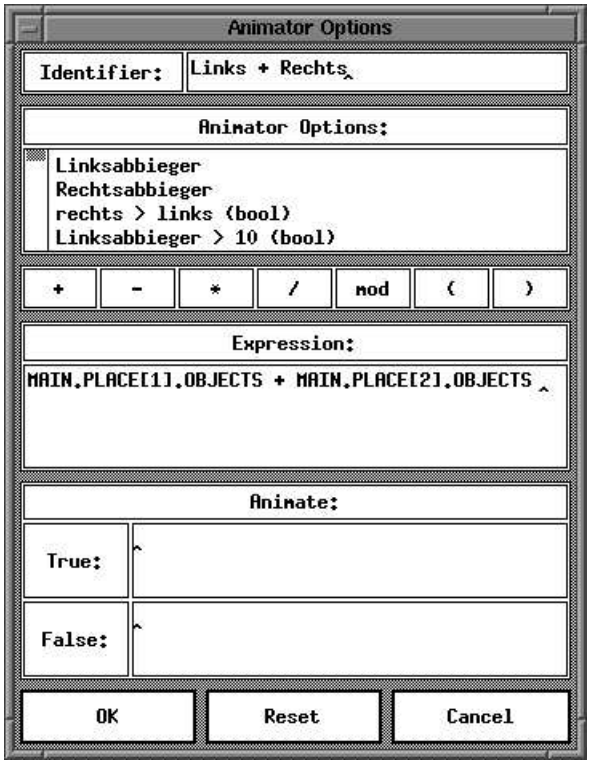


Abbildung 9.2: Steuerung des Animators

(siehe Abb. 9.1). Hierfür sind zwei Schaltfelder vorgesehen. Außerdem können Haltepunkte angegeben werden. Neben einfachen Haltebedingungen, wie der zu erreichenden Simulationszeit oder der Anzahl der zu feuernden Transitionen, können auch komplexere Haltebedingungen angegeben werden (im Beispiel `ZeitErreicht` und `(Linksabbieger erwischt)`). Die verwendeten Namen sind Identifikatoren für komplexere Netzausdrücke, die aus bestimmten atomaren Ausdrücken – etwa über die Anzahl der Marken auf einer Stelle oder die Zahl der gefeuerten Transitionen – zusammengesetzt werden können.

Ähnlich wie Breakpoints können auch Tracepoints festgelegt werden, die für eine Visualisierung des Simulationslaufes benutzt werden können. Sowohl Break- als auch Tracepoints werden durch Netzausdrücke angegeben; einen Eindruck vermittelt Abb. 9.2. Hier ist etwa zu sehen, wie ein komplexer Ausdruck, der mit `Links + Rechts` benannt ist, aus elementaren Netzausdrücken gebildet wird (im Beispiel wird etwa mit `Links + Rechts` vermittels des Ausdrucks `MAIN.PLACE[1].OBJECTS + MAIN.PLACE[2].OBJECTS` die Summe aus der Anzahl der Objekte auf den Stellen mit Nummer 1 und 2 im Hauptnetz (`MAIN`) identifiziert).

Die Netzausdrücke müssen nicht komplett von Hand eingegeben werden, vielmehr ist es möglich, etwa durch Anwählen einer Stelle im Netzeditor den entsprechenden Netzausdruck für die Stelle (etwa `MAIN.PLACE[1]`) automatisch zu generieren. Ebenso können alle bereits verwendeten Ausdrücke in einer Liste angewählt und in einem komplexeren Ausdruck wiederverwendet werden (in Abb. 9.2 sind dies etwa `Linksabbieger`, `Rechtsabbieger` etc.).

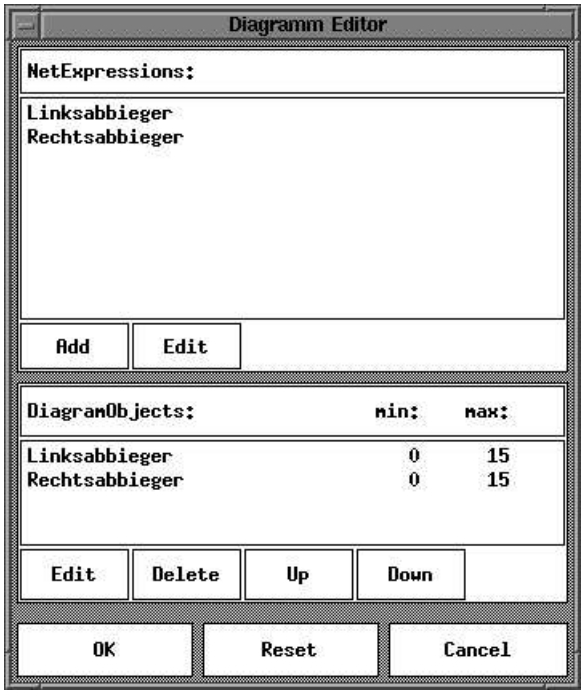


Abbildung 9.3: Steuerungsfenster für Balkendiagramme

Folgende Aktionen können bei der Änderung des Wertes eines Netzausdrucks ausgelöst werden:

- Das Anhalten der Simulation. Hierfür muß der Netzausdruck einen booleschen Wert zurückliefern, der im Moment des Anhaltens von `false` auf `true` wechselt. Nach dem Anhalten der Simulation kann der Netzzustand inspiziert und geändert werden und danach fortgesetzt werden. Auch das Ab- oder Anschalten bzw. Ändern von Break- und Tracepoints ist vor dem Weitersimulieren möglich.
- Das Aktualisieren eines Diagrammfensters, das einen Netzteilzustand wiedergibt. Je nach verwendetem Diagrammtyp (s. u.) muß der Netzausdruck einen booleschen oder einen numerischen Wert zurückliefern.
- Das Aktualisieren eines Animationsfensters.

Zur Visualisierung sind drei verschiedene Diagrammtypen vorgesehen. Balken- und Kurvendiagramme geben Werte numerischer Netzausdrücke wieder. Hierbei können mehrere Netzteilzustände einander in übersichtlicher Form gegenübergestellt werden.

Balkendiagramme erlauben es, jeweils den Momentanzustand mehrerer durch Netzausdrücke festgelegter Werte zu visualisieren. In einem speziellen Steuerungsfenster (siehe Abb. 9.3) wird festgelegt, welche Ausdrücke einander gegenübergestellt werden sollen. Hierbei können alle bereits definierten Netzausdrücke über ihre Identifikatoren verwendet werden. Um die Skalierung der Balkenhöhe festzulegen, können für jeden einzelnen Netzausdruck Minimal- und Maximalwerte angegeben werden. Im Beispiel werden so etwa bei

einer Verkehrssimulation die Anzahl der Links- und Rechtsabbieger einander gegenübergestellt.

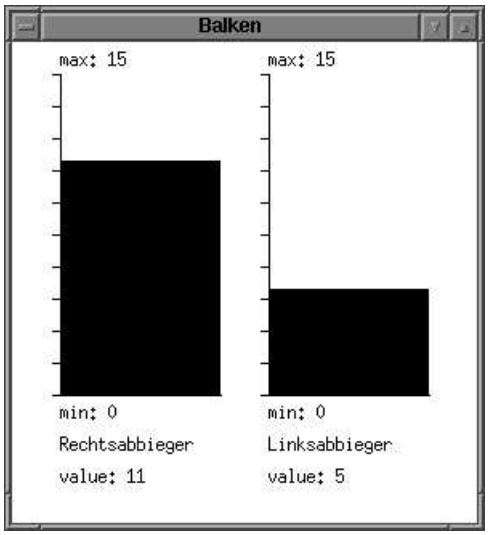


Abbildung 9.4: Darstellungsfenster für Balkendiagramme

Die durch die in Abb. 9.3 festgelegte Darstellung der Netzausdrücke ist in Abb. 9.4 wiedergegeben. Hierbei werden die Balken entsprechend angepaßt, sobald sich der Wert eines der Netzausdrücke ändert. Damit der Simulationslauf auch durch den Anwender in geeigneter Weise verfolgt werden kann, ist es möglich, im Kontrollfenster (siehe Abb. 9.1) die Geschwindigkeit des Simulationslaufes anzupassen. Auch ein Einzelschrittsimulationsverfahren kann dort gewählt werden.

Mit Balkendiagrammen läßt sich zwar sehr gut der momentane Netzzustand bzw. ein interessierender Teil hiervon darstellen, ein Verlauf bestimmter Netzzustände – etwa der Füllungsgrad einer Stelle – über einen Zeitraum ist jedoch nicht ablesbar. Hierzu dienen Kurvendiagramme. In Abb. 9.5 ist etwa dieselbe Information wie in Abb. 9.4 dargestellt, wobei allerdings nun auch der Verlauf der Werte sichtbar ist. Die Skalierung der x-Achse kann dabei im (hier nicht dargestellten) Kontrollfenster, das ansonsten wie das in Abb. 9.3 aufgebaut ist, festgelegt werden. Falls der aktuelle Simulationszeitpunkt über den gerade dargestellten rechten Rand der x-Achse hinausgeht, wird diese automatisch nachgeführt. Es ist aber auch anhand der Schalter am oberen Rand des Darstellungsfensters eine gezielte Bewegung nach links oder rechts möglich, so daß z. B. nach Ablauf oder beim Anhalten der Simulation der gesamte Simulationsverlauf nochmals untersucht werden kann.

Mit Hilfe von Ganttdiagrammen kann das Erfüllt- bzw. Nichterfülltsein bestimmter boolescher Bedingungen visualisiert werden. In einem ähnlichen Kontrollfenster wie für Balken- und Kurvendiagramme (siehe Abb. 9.3) werden die darzustellenden booleschen Netzausdrücke angegeben (die Angabe von Minimal- und Maximalwerten ist allerdings natürlich nicht erforderlich). In Abb. 9.6 ist dargestellt, wie etwa zwei Bedingungen gleichzeitig visualisiert werden. Die Bedingung, daß die Anzahl der Linksabbieger größer als 10 ist, ist in der unteren Zeile dargestellt. In den Simulationszeitintervallen $[22, 24)$ und $[26, 30]$ war dabei diese Bedingung erfüllt, während sie in den Intervallen $[24, 26)$ und $[30, 34)$ nicht

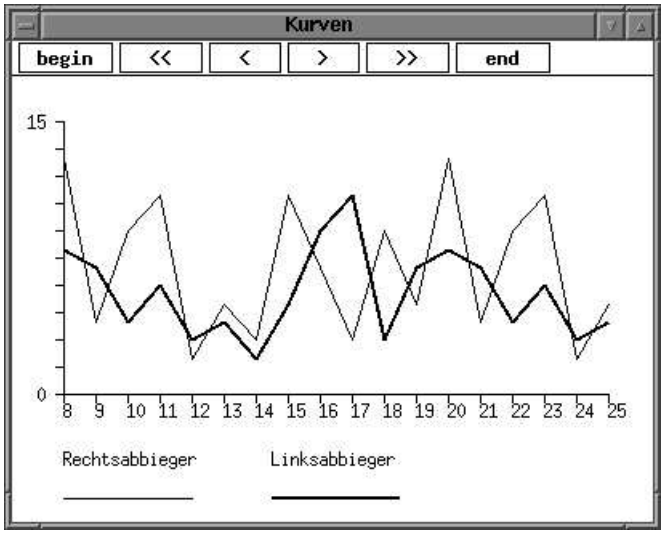


Abbildung 9.5: Darstellungsfenster für Kurvendiagramme

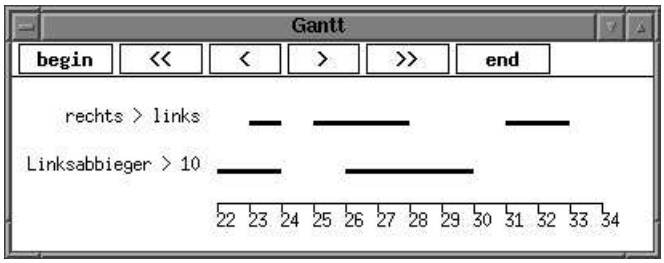


Abbildung 9.6: Darstellungsfenster für Ganttdiagramme

zutraf. Informationen über die gerade nicht dargestellten Zeitabschnitte können wiederum mit Hilfe von speziellen Schaltern am oberen Fensterrand erlangt werden.

Neben der Visualisierung von Daten ist allerdings auch eine Animation des Simulationslaufes möglich. Mit booleschen Netzausdrücken können dabei bestimmte Zeichenoperationen in einem Animationsfenster verknüpft werden. So kann etwa bei einer Verkehrssimulation das Schalten einer durch ein THORN dargestellten Ampel durch eine Darstellung der Ampel mit ihren gerade aktuellen Schaltfarben oder das Verkehrsaufkommen an einer Kreuzung durch eine entsprechende, sich verändernde Graphik der Kreuzung und der auf ihr vorhandenen Fahrzeuge visualisiert werden (siehe Abb. 9.7).

Weitere im Rahmen der Projektgruppe vorgenomme Verbesserungen, die hier jedoch nicht detaillierter dargestellt werden sollen, betrafen die Bereitstellung eines Klasseneditors für die bei THORNs notwendigen Klassenbeschreibungen von Token und die Möglichkeit, den Zustand des Editorsystems, d. h. die Größe und Anzahl geöffneter Fenster mit den zugehörigen Unternetzen, abspeichern und wieder laden zu können.

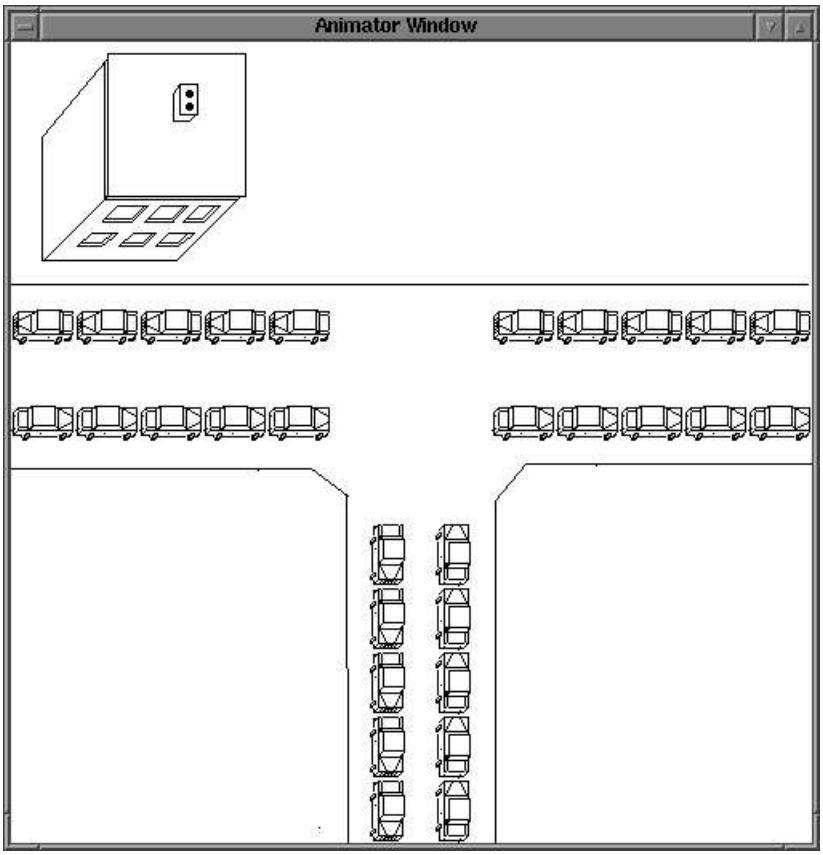


Abbildung 9.7: Animation eines Simulationslaufes

Literaturverzeichnis

[Aal93] W. M. P. v. d. Aalst. Interval Timed Coloured Petri Nets and their Analysis. In [Ajm93], Seiten 453–472.

[ACHH93] R. Alur, C. Courcoubetis, T. A. Henzinger und P.-H. Ho. Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems. In R. L. Gross, A. Nerode, A. P. Ravn und H. Rischel (Hrsg.), *Workshop on Theory of Hybrid Systems*, Band 736 der *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1993.

[AH91] R. Alur und T. A. Henzinger. Logics and models of real time: A survey. In J. W. de Bakker, C. Huizing, W. P. de Roever und G. Rozenberg (Hrsg.), *Real-Time: Theory in Practice*, Band 600 der *Lecture Notes in Computer Science*, Mook, The Netherlands, Juni 1991. REX Workshop, Springer-Verlag, Berlin.

[Ajm89] M. Ajmone Marsan. Stochastic Petri Nets: an elementary introduction. In [Roz89], Seiten 1–50.

[Ajm93] M. Ajmone Marsan (Hrsg.). *Proceedings of the* 14^{th} *International Conference on Application and Theory of Petri Nets 1993*, Band 691 der *Lecture Notes in Computer Science*, Chicago, Illinois, USA, Juni 1993. Springer-Verlag, Berlin.

[Amd67] G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, Band 30, Seiten 483–485, Atlantic City, New Jersey, Apr. 1967. AFIPS Press, Reston.

[AR92] M. Abrams und P. F. Reynolds Jr. (Hrsg.). *Proceedings of the* 6^{th} *Workshop on Parallel and Distributed Simulation*, Band 24 der *Simulation Series*, Newport Beach, California, Jan. 20–22, 1992. Society for Computer Simulation International.

[Bar94] U. Bartels. Fallstudie zur Modellierung der Beschaffungsabläufe in der Zentralen Einrichtung für wiss.-techn. Ausstattung (ZEfA). Bericht der Arbeitsgruppe Informatik-Systeme AIS–15, Fachbereich Informatik, Universität Oldenburg, Jan. 1994.

[Bau90] B. Baumgarten. *Petri-Netze: Grundlagen und Anwendungen.* BI–Wissenschaftsverlag, Mannheim/Wien/Zürich, 1990.

[BD91] B. Berthomieu und M. Diaz. Modeling and Verification of Time Dependent Systems Using Time Petri Nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, März 1991.

[BDM96] E. Battiston, F. De Cindio und G. Mauri. Modular Algebraic Nets to Specify Concurrent Systems. *IEEE Transactions on Software Engineering*, 22(10):689–705, Okt. 1996.

[Bel90] S. Bellenot. Global virtual time algorithms. In D. Nicol (Hrsg.), *Proceedings of the SCS Multiconference on Distributed Simulation*, Band 22 der *Simulation Series*, Seiten 122–127, San Diego, California, Jan. 1990. Society for Computer Simulation International.

[BGV90] W. Brauer, R. Gold und W. Vogler. A Survey of Behaviour and Equivalence Preserving Refinements of Petri Nets. In [Roz90], Seiten 1–46.

[Bra84] W. Brauer. How to Play the Token Game? or Difficulties in Interpreting Place/Transition Nets. *Petri Net Newsletter*, 16:3–13, Feb. 1984.

[CFL+89] V. Claus, H. Fleischhack, U. Lichtblau u. a. Abschlußbericht der Projektgruppe Nessi (Nets Simulation), Band I. Interne Berichte TI 4, Fachbereich Informatik, Universität Oldenburg, Juli 1989.

[CH93] S. Christensen und N. D. Hansen. Coloured Petri Nets Extended with Place Capacities, Test Arcs and Inhibitor Arcs. In [Ajm93], Seiten 186–205.

[CK81] L. A. Cherkasova und V. L. Kotov. Structured nets. In W. Brauer (Hrsg.), *Mathematical Foundations of Computer Science*, Band 118 der *Lecture Notes in Computer Science*, Seiten 242–251. Springer-Verlag, Berlin, 1981.

[CL85] K. M. Chandy und L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb. 1985.

[CM93] R. Cooper und K. Marzullo. Consistent detection of global predicates. In B. P. Miller und C. McDowell (Hrsg.), *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, Band 26 der *ACM SIGPLAN Notices*, Seiten 167–174, Santa Cruz, California, Dez. 1993. ACM/ONR, ACM Press.

[CT93] S. Christensen und J. Toksvig. DesignBeta V2.0.1 — BETA code-segments in CP-nets. Lecture Notes OO&CPN No 5, Computer Science Department, Aarhus University, Denmark, 1993.

[DA92] R. David und H. Alla. *Petri Nets & Grafcet – Tools for modelling discrete event systems.* Prentice Hall, New York, 1992.

[DD95] G. De Michelis und M. Diaz (Hrsg.). *Proceedings of the* 16^{th} *International Conference on Application and Theory of Petri Nets 1995*, Band 935 der *Lecture Notes in Computer Science*, Turin, Italy, Juni 1995. Springer-Verlag, Berlin.

[ELP+92] M. Ebling, M. D. Loreto, M. Presley, F. Wieland und D. R. Jefferson. An ant foraging model implemented on the Time Warp operating system. In [AR92], Seiten 21–26.

[End96] N. Enderlein. Steuerung und Visualisierung von HyNet-Simulationen. Diplomarbeit, Fachbereich Informatik, Universität Oldenburg, Nov. 1996.

[Feh91] R. Fehling. A Concept of Hierarchical Petri Nets with Building Blocks. In *Proceedings of the 12th International Conference on Applications and Theory of Petri Nets*, Seiten 370–389, Gjern, Denmark, Juni 1991.

[Feh92] R. Fehling. *Hierarchische Petrinetze*. Verlag Dr. Kovač, Hamburg, 1992. Dissertation.

[FL93] H. Fleischhack und U. Lichtblau. $\mathcal{MOBY}$ — A Tool for High Level Petri Nets with Objects. In *IEEE International Conference on Systems, Man and Cybernetics*, Band 4, Seiten 644–649, Le Touquet, France, Okt. 1993.

[FLSW93a] H. Fleischhack, U. Lichtblau, M. Sonnenschein und R. Wieting. Abstraktion und Zeitbegriff in höheren Netzen. In G. Scheschonk und W. Reisig (Hrsg.), *Petri-Netze im Einsatz für Entwurf und Entwicklung von Informationssystemen*, Informatik aktuell, Seiten 59–71. Springer-Verlag, Berlin, Sept. 1993.

[FLSW93b] H. Fleischhack, U. Lichtblau, M. Sonnenschein und R. Wieting. Generische Definition {hierarchischer} {zeitbeschrifteter} {höherer} Petrinetze. Bericht der Arbeitsgruppe Informatik-Systeme AIS–13, Fachbereich Informatik, Universität Oldenburg, Dez. 1993.

[Fuj90] R. M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):31–53, 1990.

[FW95] J. Fleischmann und P. A. Wilsey. Comparative analysis of periodic state saving techniques in Time Warp simulators. In *Proceedings of the* 9^{th} *Workshop on Parallel and Distributed Simulation*, Seiten 50–58, Lake Placid, New York, Juni 1995. IEEE Computer Society Press, Los Alamitos, CA.

[Gar70] M. Gardner. Mathematical games. *Scientific American*, Seiten 120–123, Okt. 1970.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra u. a. *PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, 1994.

[Gen87] H. J. Genrich. Predicate/Transition Nets. In W. Brauer, W. Reisig und G. Rozenberg (Hrsg.), *Petri Nets: Central Models and Their Properties*, Band 254 der *Lecture Notes in Computer Science*, Seiten 207–247. Springer-Verlag, Berlin, 1987. Auch erschienen als [Gen91].

[Gen91] H. J. Genrich. Predicate/Transition Nets. In [JR91], Seiten 3–43.

[GLT80] H. J. Genrich, K. Lautenbach und P. S. Thiagarajan. Elements of General Net Theory. In W. Brauer (Hrsg.), *Net Theory and Applications*, Band 84 der *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

[GMMP91] C. Ghezzi, D. Mandioli, S. Morasca und M. Pezzé. A Unified High-level Petri Net Formalism for Time-Critical Systems. *IEEE Transactions on Software Engineering*, 17(2):160–172, Feb. 1991.

[GS94] A. Gronewold und M. Sonnenschein. Diskrete Petrinetze für individuenbasierte Modelle. In *Tagungsband zum Workshop Ökosysteme — Modellierung und Simulation*, Umweltwissenschaften, Cottbus, Juni 1994. Eberhard Blottner Verlag.

[GW94] V. K. Garg und B. Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Trans. Par. Distr. Syst.*, 5(3):299–307, März 1994.

[HJS90] P. Huber, K. Jensen und R. M. Shapiro. Hierarchies in Coloured Petri Nets. In [Roz90], Seiten 313–341.

[HV87] M. A. Holliday und M. K. Vernon. A Generalized Timed Petri Net Model for Peformance Analysis. *IEEE Transactions on Software Engineering*, 13(12):1297–1310, Dez. 1987.

[Jef85] D. R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, Juli 1985.

[Jen92] K. Jensen. *Coloured Petri Nets — Basic Concepts, Analysis Methods and Practical Use Volume 1.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1992.

[Jen95] K. Jensen. *Coloured Petri Nets — Basic Concepts, Analysis Methods and Practical Use Volume 2.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1995.

[Jon86] D. W. Jones. An empirical comparison of priority-queue and event-set implementations. *Commun. ACM*, 29(4):300–311, Apr. 1986.

[JR91] K. Jensen und G. Rozenberg (Hrsg.). *High-level Petri Nets — Theory and Application.* Springer-Verlag, Berlin, 1991.

[Kie89a] A. Kiehn. A Structuring Mechanism for Petri Nets. Dissertation, Technische Universität München, 1989.

[Kie89b] A. Kiehn. Petri Net Systems and their Closure Properties. In [Roz89], Seiten 306–328.

[KKT96] B. Kleinjohann, E. Kleinjohann und J. Tacken. The SEA Language for System Engineering and Animation. In M. Silva, R. Valette und K. Takahashi (Hrsg.), *Proceedings of the 17th International Conference on Application and Theory of Petri Nets 1996*, Band 1091 der *Lecture Notes in Computer Science*, Osaka, Japan, Juni 1996. Springer-Verlag, Berlin.

[Kos94] R. Koschel. Modellierung von Kommunikationsprotokollen mit THOR-Netzen. Studienarbeit, Fachbereich Informatik, Universität Oldenburg, März 1994.

[Kös96] F. Köster. Bewertung hierarchischer Petrinetze als Grundlage für Mapping-Verfahren bei der verteilten Simulation. Diplomarbeit, Universität Oldenburg, Mai 1996.

[Kru95] J. Kruse. Modellierung, Implementierung und praktische Erprobung einer Simulationsumgebung zum Test verteilter Algorithmen zur GVT-Approximation, school = Universität Oldenburg. Diplomarbeit, Juli 1995.

[KTWZ93] F. Köster, L. Twele, R. Wieting und W. Ziegler. Fallbeispiele zur Modellierung mit THOR-Netzen. Bericht der Arbeitsgruppe Informatik-Systeme AIS–14, Fachbereich Informatik, Universität Oldenburg, Dez. 1993.

[Lak95] C. A. Lakos. From Coloured Petri Nets to Object Petri Nets. In [DD95], Seiten 278–297.

[LL90] Y.-B. Lin und E. D. Lazowska. Determining the global virtual time in a distributed simulation. In P.-C. Yew (Hrsg.), *Proceedings of the 1990 International Conference on Parallel Processing*, Band III, Seiten 201–209, Urbana-Champaign, IL, Aug. 1990. Pennsylvania State University Press.

[Maj96] R. Majchszak. Eine objektorientierte Beschriftungssprache höherer Petrinetze für hybride Modelle. Diplomarbeit, Fachbereich Informatik, Universität Oldenburg, Apr. 1996.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard, Y. Robert, P. Quinton und M. Raynal (Hrsg.), *Parallel and Distributed Algorithms*, Seiten 215–226, Gers, France, 1989. North-Holland.

[Mat93] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *J. Paral. Distr. Comp.*, 18:423–434, 1993.

[Mau95] S. Maurer. Entwurf und Implementierung einer graphischen Oberfläche zur Entwicklung hierarchischer Petrinetzmodelle. Diplomarbeit, Fachbereich Informatik, Universität Oldenburg, Jan. 1995.

[Mer74] P. Merlin. A Study of the Recoverability of Communication Protocols. Ph. D. Thesis, Computer Science Department, University of California, Irvine, 1974. Zitiert in [Pop89].

[Mie96] B. Mielenhausen. Sequentielle Simulation von THOR-Netzen. Interner Bericht im Rahmen des Projekts DNS, OFFIS, Oldenburg, Juli 1996.

[Mie97] B. Mielenhausen. Entwurf und Implementierung eines Editors für THOR-Netze unter MS Windows. Diplomarbeit, Fachbereich Informatik, Universität Oldenburg, Apr. 1997.

[Mol82] M. K. Molloy. Performance Analysis Using Stochastic Petri Nets. *IEEE Transactions on Computers*, C-31(9):913–917, Sept. 1982.

[PEWJ92] M. Presley, M. Ebling, F. Wieland und D. R. Jefferson. Benchmarking the Time Warp operating system with a computer network simulation. In [AR92], Seiten 8–13.

[PML92] B. R. Preiss, D. Macintyre und W. M. Loucks. On the trade-off between time and space in optimistic parallel discrete-event simulation. In [AR92], Seiten 33–42.

[Poh96] H.-P. Pohle. Konservative verteilte Simulation von THOR-Netzen. Diplomarbeit, Universität Oldenburg, Nov. 1996.

[Pop89] L. G. Popova-Zeugmann. Zeit-Petri-Netze. Dissertation, Humboldt-Universität, Berlin, 1989.

[Ram74] C. Ramchandani. Analysis of Asynchronous Concurrent Systems by Petri Nets. Interim Scientific Report MAC TR–120, Massachusetts Institute of Technology, Cambridge, Massachusetts, Feb. 1974. Ph. D. Thesis.

[Ram93] F. J. Rammig. Modelling Aspects of System Level Design. In *Proceedings of the EURO-DAC '93*, Seiten 534–539, Hamburg, Sept. 1993. IEEE Computer Society Press, Los Alamitos, CA.

[Ree96] G. Reents. Effizienzsteigerung bei der optimistischen verteilten THOR-Netzsimulation. Diplomarbeit, Universität Oldenburg, Nov. 1996.

[Rei85] W. Reisig. *Petri Nets: An Introduction*, Band 4 der *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1985.

[Rei86] W. Reisig. *Petrinetze: Eine Einführung*. Studienreihe Informatik. Springer-Verlag, Berlin, 1986. Zweite, überarbeitete und erweiterte Auflage. Auch erschienen als [Rei85].

[Rei91a] W. Reisig. Petri Nets and Algebraic Specifications. *Theoretical Computer Science*, 80:1–34, 1991. Auch erschienen als [Rei91b].

[Rei91b] W. Reisig. Petri Nets and Algebraic Specifications. In [JR91], Seiten 137–170.

[Rei94] H. Reineke. Die Analyse höherer Petrinetze – ein Überblick. Bericht der Arbeitsgruppe Informatik-Systeme AIS–16, Fachbereich Informatik, Universität Oldenburg, Juni 1994.

[Rei96] K. Reich. Beobachtung globaler Zustände bei der verteilten optimistischen Simulation. Diplomarbeit, Universität Oldenburg, Feb. 1996.

[Roz89] G. Rozenberg (Hrsg.). *Advances in Petri Nets*, Band 424 der *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1989.

[Roz90] G. Rozenberg (Hrsg.). *Advances in Petri Nets*, Band 483 der *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1990.

[SB94] C. Sibertain-Blanc. Cooperative Nets. In R. Valette (Hrsg.), *Proceedings of the 15th International Conference on Application and Theory of Petri Nets 1994*, Band 815 der *Lecture Notes in Computer Science*, Seiten 471–490, Zaragoza, Spain, Juni 1994. Springer-Verlag, Berlin.

[Sch96a] M. Schaal. Verkehrsmodellierung. Diplomarbeit Nr. 1373, Fakultät Informatik, Universität Stuttgart, Juli 1996.

[Sch96b] S. Schöf. Time Warp calendar queues. In K. Yetongnon und S. Hariri (Hrsg.), *Proceedings of ISCA 9th International Conference on Parallel and Distributed Computing Systems*, Band II, Seiten 815–816. ISCA, International Society for Computers and Their Applications, Sept. 1996.

[Sch97] S. Schöf. Efficient data structures for Time Warp simulation queues. *Euromicro J. Syst. Archit.*, 1997. Special Issue on Parallel and Distributed Simulation. Erscheint voraussichtlich Juni 1997.

[SE96] H. M. Soliman und A. S. Elmaghraby. An improved chandy-lamport snapshot algorithm for GVT approximation in distributed simulations. *ISCA*, 3(1):1–7, Apr. 1996.

[Sei93] A. Seidel. Entwurf und Implementierung eines Editors für elementare Petriboxen. Diplomarbeit, Fachbereich Informatik, Universität Oldenburg, Nov. 1993.

[SSW94a] S. Schöf, M. Sonnenschein und R. Wieting. Sequentielle und verteilte Simulation von THOR-Netzen. In J. Desel, A. Oberweis und W. Reisig (Hrsg.), *Algorithmen und Werkzeuge für Petrinetze*, Workshop der GI-Fachgruppe 0.0.1 "Petrinetze und verwandte Systemmodelle", Seiten 61–66, Berlin, Okt. 1994. Erschienen als Forschungsbericht 309, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, Universität Karlsruhe (TH).

[SSW94b] S. Schöf, M. Sonnenschein und R. Wieting. Verteilte Simulation von THOR-Netzen. In S. Schöf, M. Sonnenschein und R. Wieting (Hrsg.), *Parallele und verteilte Simulation*, Bericht der Arbeitsgruppe Informatik-Systeme AIS–21, Seiten 20–25. Fachbereich Informatik, Universität Oldenburg, Nov. 1994.

[SSW95a] S. Schöf, M. Sonnenschein und R. Wieting. Efficient Simulation of THOR Nets. In [DD95], Seiten 412–431.

[SSW95b] S. Schöf, M. Sonnenschein und R. Wieting. High-level Modeling with THORNs. In *Proceedings of the 14th International Congress on Cybernetics*, Seiten 453–458, Namur, Belgium, Aug. 1995.

[SSW+96] M. Sonnenschein, S. Schöf, R. Wieting u. a. Abschlußbericht der Projektgruppe IDEFIX – Umgebung zur Simulation mittels THOR-Netzen. Interne Berichte, Fachbereich Informatik PSS 96/1, Universität Oldenburg, März 1996.

[Sta90] P. H. Starke. *Analyse von Petri-Netz-Modellen*. Leitfäden und Monographien der Informatik. Teubner Verlag, Stuttgart, 1990.

[Str92] B. Stroustrup. *Die C++ Programmiersprache*. Addison-Wesley (Deutschland), Bonn, München, Paris, 2. Auflage, 1992.

[Stu95] M. Stulle. Die ereignisdiskrete Beobachtungsaufgabe im Kontext des Koordinationssteuerungsentwurfs für flexible Fertigungssysteme. In E. Schnieder (Hrsg.), *Entwurf komplexer Automatisierungssysteme*, Methoden, Anwendungen und Tools auf der Basis von Petrinetzen und anderer formaler Beschreibungsmittel, 4. Fachtagung, Seiten 585–596, Braunschweig, Juni 1995.

[SWK+94] S. Schöf, R. Wieting, F. Köster, B. Mielenhausen, R. Radtke, G. Reents, L. Twele und W. Ziegler. Konzepte zur sequentiellen und verteilten Simulation von THOR-Netzen. Bericht der Arbeitsgruppe Informatik-Systeme AIS–17, Fachbereich Informatik, Universität Oldenburg, Juni 1994.

[TG93] A. I. Tomlinson und V. K. Garg. Detecting relational global predicates in distributed systems. In B. P. Miller und C. McDowell (Hrsg.), *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, Band 28 der *ACM SIGPLAN Notices*, Seiten 21–31, San Diego, California, Dez. 1993. ACM/ONR, ACM Press.

[Tho96] W. Tholen. Entwurf und Implementierung eines Interpreters für hybride High-Level Netze. Diplomarbeit, Fachbereich Informatik, Universität Oldenburg, Nov. 1996.

[TK93] K. S. Trivedi und V. G. Kulkarni. FSPNs: Fluid Stochastic Petri Nets. In [Ajm93], Seiten 24–31.

[Twe95a] L. Twele. Beschreibung des ANR-File-Formats für THOR-Netze. Interner Bericht im Rahmen des Projekts DNS, OFFIS, Oldenburg, Aug. 1995.

[Twe95b] L. Twele. Beschreibung des THOR-Netz-Compilers. Interner Bericht im Rahmen des Projekts DNS, OFFIS, Oldenburg, Nov. 1995.

[Val96] R. Valk. On the process of Object Petri Nets. Bericht Nr. 185 (FBI-HH-B-185), Universität Hamburg, Fachbereich Informatik, Vogt-Kölln-Straße 30, D-22527 Hamburg, Juni 1996.

[WHF+92] F. Wieland, L. Hawley, A. Feinberg u. a. Distributed combat simulation and Time Warp: The model and its performance. In [AR92], Seiten 14–20.

[Wie95] R. Wieting. Verkehrsmodellierung mit THOR-Netzen. In J. Desel, H. Fleischhack, A. Oberweis und M. Sonnenschein (Hrsg.), *2. Workshop Algorithmen und Werkzeuge für Petrinetze*, Bericht der Arbeitsgruppe Informatik-Systeme AIS–22, Seiten 67–73, Oldenburg, Okt. 1995. Fachbereich Informatik, Universität Oldenburg.

[Wie96a] R. Wieting. Handbuch zur THORN Entwicklungsumgebung. Bericht der Arbeitsgruppe Informatik-Systeme AIS–25, Fachbereich Informatik, Universität Oldenburg, Aug. 1996.

[Wie96b] R. Wieting. Hybrid High-level Nets. In J. M. Charnes, D. J. Morrice, D. T. Brunner und J. J. Swain (Hrsg.), *Proceedings of the 1996 Winter Simulation Conference*, Seiten 848–855, Hotel del Coronado, Coronado, California, USA, Dez. 1996.

[Wie96c] R. Wieting. Modeling and Simulation of Hybrid Systems Using Hybrid High-level Nets. In A. G. Bruzzone und E. J. H. Kerckhoffs (Hrsg.), *Proceedings of the 8th European Simulation Symposium (ESS'96)*, Band II, Seiten 158–162, Genoa, Italy, Okt. 1996. SCS Publications.

[Wie97] R. Wieting. Modellbildung und Simulation mit hybriden höheren Netzen. Dissertation, Universität Oldenburg, Oldenburg, 1997. In Vorbereitung.

[WS95] R. Wieting und M. Sonnenschein. Extending High-level Petri Nets for Modeling Hybrid Systems. In A. Sydow (Hrsg.), *Proceedings of the IMACS Symposium on Systems Analysis and Simulation*, Band 18–19 der *Systems Analysis Modelling Simulation*, Seiten 259–262, Berlin, Juni 1995. Gordon and Breach Publishers.

[Zie95] W. Ziegler. Erweiterung und Optimierung von Simulatoren für THOR-Netz-Modelle. Diplomarbeit, Fachbereich Informatik, Universität Oldenburg, Feb. 1995.

Anhang A

Erstellte Arbeiten

A.1 Veröffentlichungen

1. Hans Fleischhack, Ulrike Lichtblau, Michael Sonnenschein, Ralf Wieting. *Abstraktion und Zeitbegriff in höheren Netzen.* In G. Scheschonk, W. Reisig, editors, Petri-Netze im Einsatz für Entwurf und Entwicklung von Informationssystemen, Informatik aktuell, pp. 59-71. Springer-Verlag, Berlin, Germany, September 1993.

2. Stefan Schöf, Michael Sonnenschein, Ralf Wieting. *Sequentielle und verteilte Simulation von* THOR*-Netzen.* In J. Desel, Andreas Oberweis, W. Reisig, editors, Workshop „Algorithmen und Werkzeuge für Petrinetze“, pp. 61-66, Berlin, Germany, Oct. 1994. Published as Forschungsbericht 309, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, Universität Karlsruhe (TH).

3. Michael Sonnenschein, Anja Gronewold. *Diskrete Petrinetze für individuenbasierte Modelle.* Ökosysteme: Modellierung und Simulation, Cottbus, Germany, pp. 109-130, Eberhard Blottner Verlag, Taunusstein, Reihe Umweltwissenschaften, Band 6, 1995.

4. Stefan Schöf, Michael Sonnenschein, Ralf Wieting. *Efficient Simulation of* THOR *Nets.* Proceedings of the 16th International Conference on Application and Theory of Petri Nets, Torino, Italy, June 26-30, 1995, pp. 412-431. LNCS 935, Springer-Verlag, Berlin, Germany.

5. Ralf Wieting, Michael Sonnenschein. *Extending High-level Petri Nets for Modeling Hybrid Systems.* In A. Sydow, editor, Proceedings of the IMACS Symposium on Systems Analysis and Simulation, Bände 18-19 der Systems Analysis Modelling Simulation, pp. 259-262, Berlin, Germany, June 26-30, 1995. Gordon and Breach Publishers.

6. Stefan Schöf. *Concepts for the Optimistic Simulation of* THORNS. Proceedings of the Annual Meeting of the International Institute for Advanced Studies in Systems Research and Cybernetics (InterSymp '95), Baden-Baden, Germany, August 16-20, 1995, Vol. 3, pp. 39-43.

7. Stefan Schöf, Michael Sonnenschein, Ralf Wieting. *High-level Modeling with* THORNs. Proceedings of the 14th International Congress on Cybernetics, Namur, Belgium, August 21-25, 1995, pp. 453-458.

8. Stefan Schöf. *Quality Considerations for Mapping Strategies of Dynamic Models in Time Warp Simulation.* Proceedings of the 7th European Simulation Symposium, Erlangen, Germany, 26-28 October 1995, pp. 445-449. Society for Computer Simulation International.

9. Stefan Schöf. *A Distributed Simulation Engine for Hierarchical Petri Nets.* Tagungsband zum 10. ASIM-Workshop „Simulation verteilter Systeme und paralleler Prozesse“, Dresden, Germany, October 23-24, 1995, pp. 153-159.

10. Stefan Schöf. *Time Warp Calendar Queues.* Proceedings of the ISCA 9th International Conference on Parallel and Distributed Computing Systems, Dijon, France, September 25-27, 1996, pp. 815-816.

11. Stefan Schöf and Michael Sampels. *Fairness and Instant Reactions in Distributed Simulation.* In A. G. Bruzzone and E. J. H. Kerckhoffs, editors, Proceedings of the 8th European Simulation Symposium (ESS '96), Volume I, Genoa, Italy, October 24-26, 1996, pp. 96-100.

12. Ralf Wieting. *Modeling and Simulation of Hybrid Systems Using Hybrid High-level Nets.* In A. G. Bruzzone and E. J. H. Kerckhoffs, editors, Proceedings of the 8th European Simulation Symposium (ESS'96), Volume II, Genoa, Italy, October 24-26, 1996, pp. 158-162.

13. Michael Sampels and Stefan Schöf. *Massively Parallel Architectures for Parallel Discrete Event Simulation.* In A. G. Bruzzone and E. J. H. Kerckhoffs, editors, Proceedings of the 8th European Simulation Symposium (ESS '96), Volume II, Genoa, Italy, October 24-26, 1996, pp. 374-378.

14. Ralf Wieting. *Hybrid High-level Nets.* In J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain, editors, Proceedings of the 1996 Winter Simulation Conference, Coronado, California, USA, December 8-11, 1996, pp. 848-855.

15. Stefan Schöf. *Lastverteilung bei der Time-Warp-Simulation höherer Petrinetze.* Tagungsband des Treffen der ASIM-Arbeitskreise „Simulation Technischer Systeme“ und „Simulationsmethoden für verteilte Systeme und parallele Prozesse“, Rostock, März 1997, ASIM-Mitteilungen, to appear.

16. Stefan Schöf. *Efficient Data Structures for Time Warp Simulation Queues.* Euromicro J. Syst. Archit., Special Issue on Parallel and Distributed Simulation, to appear.

17. Anja Gronewold, Michael Sonnenschein. *Event-based Modelling of Ecological Systems with Asynchronous Cellular Automata.* In Ecological Modelling, to appear.

A.2 Workshops und Technische Berichte

1. Karin Rössig. *Zum Stand der Forschung auf dem Gebiet der parallelen Logiksimulation.* Bericht der Arbeitsgruppe Informatik-Systeme Nr. AIS-9, Fachbereich Informatik der Universität Oldenburg, April 1993.

2. Karin Rössig. *Eine Übersicht über aktuelle Ansätze zur verteilten optimistischen Simulation.* Bericht der Arbeitsgruppe Informatik-Systeme Nr. AIS-11, Fachbereich Informatik der Universität Oldenburg, August 1993.

3. Hans Fleischhack, Ulrike Lichtblau, Michael Sonnenschein, Ralf Wieting. *Generische Definition hierarchischer zeitbeschrifteter höherer Petrinetze.* Bericht der Arbeitsgruppe Informatik-Systeme Nr. AIS-13, Fachbereich Informatik der Universität Oldenburg, Dezember 1993.

4. Frank Köster, Lutz Twele, Ralf Wieting, Werner Ziegler. *Fallbeispiele zur Modellierung mit* THOR*-Netzen.* Bericht der Arbeitsgruppe Informatik-Systeme Nr. AIS-14, Fachbereich Informatik der Universität Oldenburg, Dezember 1993.

5. Stefan Schöf, Ralf Wieting, Frank Köster, Björn Mielenhausen, Roland Radtke, Gerriet Reents, Lutz Twele, Werner Ziegler. *Konzepte zur sequentiellen und verteilten Simulation von* THOR*-Netzen.* Bericht der Arbeitsgruppe Informatik-Systeme Nr. AIS-17, Fachbereich Informatik der Universität Oldenburg, Juni 1994.

6. Stefan Schöf, Michael Sonnenschein, Ralf Wieting (Hrsg.) *Workshop über parallele und verteilte Simulation.* Berichte der Arbeitsgruppe Informatik-Systeme Nr. AIS-21, Fachbereich Informatik der Universität Oldenburg, Dezember 1994.
(enthält u.a. einen Beitrag von Stefan Schöf, Michael Sonnenschein und Ralf Wieting zum Thema *Verteilte Simulation von* THOR*-Netzen.*)

7. Michael Sonnenschein, Ulrich Bartels, Stefan Schöf, Ralf Wieting et al. *Zwischenbericht der Projektgruppe Idefix (Umgebung zur Simulation mittels* THOR*-Netzen).* Interner Bericht PSS 95/2, Fachbereich Informatik der Universität Oldenburg, Mai 1995.

8. Jörg Desel, Hans Fleischhack, Andreas Oberweis, Michael Sonnenschein (Hrsg.) *2. Workshop „Algorithmen und Werkzeuge für Petrinetze“ der GI-Fachgruppe 0.0.1.* Berichte der Arbeitsgruppe Informatik-Systeme Nr. AIS-22, Fachbereich Informatik der Universität Oldenburg, Oktober 1995.
(enthält u.a. einen Beitrag von Ralf Wieting zum Thema *Verkehrsmodellierung mit* THOR*-Netzen.*)

9. Michael Sonnenschein, Stefan Schöf, Ralf Wieting et al. *Abschlußbericht der Projektgruppe Idefix (Umgebung zur Simulation mittels* THOR*-Netzen).* Interner Bericht PSS 96/1, Fachbereich Informatik der Universität Oldenburg, März 1996.

10. Ralf Wieting. *Handbuch zur* THORN *Entwicklungsumgebung.* Bericht der Arbeitsgruppe Informatik-Systeme Nr. AIS-25, Fachbereich Informatik der Universität Oldenburg, August 1996.

A.3 Diplomarbeiten

1. Ziegler, Werner: Erweiterungen und Optimierungen von THOR-Netz-Simulatoren. Februar 1995
2. Kruse, Jörg: Modellierung, Implementierung und praktische Erprobung einer Simulationsumgebung zum Test verteilter Algorithmen. Juli 1995
3. Reich, Karsten: Beobachtung globaler Zustände bei der verteilten optimistischen Simulation Februar 1996
4. Majchszak, Roland: Eine objektorientierte Beschriftungssprache höher Petrinetze für hybride Modelle. April 1996
5. Köster, Frank: Bewertung hierarchischer Petrinetze als Grundlage für Mapping-Verfahren in der verteilten Simulation. Mai 1996
6. Pohle, Hans-Peter: Konservative verteilte Simulation von THOR-Netzen. November 1996
7. Enderlein, Nils: Steuerung und Visualisierung von HYNET Simulationen. November 1996
8. Tholen, Wolfgang: Entwurf und Implementierung eines Interpreters für hybride High-Level Netze. November 1996
9. Reents, Gerriet: Effizienzsteigerung bei der optimistischen verteilten THOR-Netz-Simulation. November 1996
10. Mielenhausen, Björn: Entwurf und Implementierung eines Editors für THOR-Netze unter MS-Windows. April 1997

A.4 Studienarbeiten

1. Kürsten, Petra: Effiziente Auswahl aktiver Transitionen zu interpretierten Petri-Netzen. Oktober 1992
2. Koschel, Ralf: Modellierung von Kommunikationsprotokollen mit THOR-Netzen. März 1994
3. Maurer, Stephan: Entwurf eines Objektnetz-Editors. Juli 1994
4. Kruse, Jörg: Erweiterung einer Simulations-Testumgebung zur Beurteilung des Zeitverhaltens verteilter Algorithmen für die Deadlockerkennung. November 1994
5. Arndt, Nils: Einbindung eines Deadlockerkennungs-Algorithmus in verteilte Programme. April 1995
6. Roth, Katja: Ein Präprozessor zur Integration eines Deadlockerkennungs-Algorithmus in verteilte Programme. April 1995

7. Tangemann, Bernd: Konvertierung von der Thornetzeditor-Beschreibungssprache in das Postscript-Format. Juli 1995