Beschleunigung von Diskret-Ereignisorientierten Simulationsstudien unter Verwendung des DEVS-Formalismus auf HPC-Systemen

Dissertation
zur
Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)
der Fakultät für Informatik und Elektrotechnik
der Universität Rostock

vorgelegt von:
David Jammer, geboren am 27.12.1990 in Hagenow
aus Wismar

Rostock, 29. Oktober 2025

Kurzfassung

Zur Planung von Systemen in ingenieurtechnischen Aufgabenstellungen kommt vermehrt die Methode der Modellbildung und Simulation (M&S) zum Einsatz. Diese ist heute so vielseitig und komplex, dass praktisch alle Systemklassen damit abgedeckt werden können. Im Laufe von mehr als einem halben Jahrhundert haben sich die Methoden dabei auf bestimmte Anwendungsbereiche konzentriert und sich dafür spezialisiert. In der Anfangsgeschichte der M&S war das Ziel mit der Modellbildung und einzelnen Simulationsläufen bereits erreicht. Für anspruchsvolle M&S-Anwendungen wurde die jeweils leistungsstärkste verfügbare Rechentechnik eingesetzt. Daraus entstand eine eigene Community, die sich mit *High Performance Computing* (HPC) in Kombination mit Simulationstechnik beschäftigt. Heutige M&S-Anwendungen gehen über einfache Simulationen weit hinaus und stellen in der Regel komplexe simulationsbasierte Experimente mit vielen tausenden Simulationsläufen dar.

Für die Modellierung aktueller Problemstellungen werden hohe Ansprüche an die Methoden gestellt. So sollen nicht nur Aspekte einer Domäne umsetzbar sein, sondern Aspekte vieler Domänen. Dies ist zum Beispiel in der Automatisierungstechnik der Fall. Hier haben sich zwei Welten entwickelt, die sich auf der einen Seite mit der Regelungstechnik und auf der anderen Seite mit der Steuerungstechnik beschäftigen. Diese Welten sind aus Sicht der M&S in die Systemklassen kontinuierlich und diskret-ereignisorientiert unterteilt. Viele praktische Probleme beinhalten aber sowohl die Steuerungs- als auch die Regelungstechnik. Für solche kombinierten Problemstellungen werden hybride M&S-Techniken benötigt. Eine dafür inzwischen geeignete Methodik ist der *Discrete Event System Specification* (DEVS)-Formalismus von 1976, der ursprünglich nur für die diskret-ereignisorientierte Systemklasse entwickelt wurde. Im Laufe der Zeit erfuhr dieser Formalismus zahlreiche Weiterentwicklungen und kann heute für viele weitere Systemklassen verwendet werden. Die letzten Weiterentwicklungen dieses Formalismus zielten darauf ab, Komponenten mit Mealy-Verhalten besser abbilden zu können. Diese Forschungen brachten den *Revised Parallel Discrete Event System Specification* (RPDEVS)-Formalismus hervor. Im Rahmen einer kritischen Analyse wies P. Junglas allerdings nach, dass auch RPDEVS noch ernsthafte Defizite aufweist. Infolge stellt er die Theorie der *Non-Standard Analysis* (NSA) als Lösungsvorschlag zur Diskussion.

In dieser Arbeit werden zwei Problemstellungen bearbeitet. (i) Der Vorschlag von Junglas wird aufgegriffen und es wird ein vollständiger NSA-DEVS-Formalismus entwickelt. Zur Verifikation wird ein komplexes Praxisproblem aus dem Bereich der Produktionstechnik mit NSA-DEVS modelliert. (ii) Darüber hinaus werden mit diesem Modell Simulationsstudien auf einem HPC-System ausgeführt und analysiert, um zu prüfen, ob mit NSA-DEVS komplexe und hochkomplexe Simulationsexperimente auf Systemen der Klasse bis 100.000 Euro in akzeptabler Zeit durchführbar sind.

Abstract

Modeling and simulation (M&S) is a method that's being used more and more to design systems for engineering tasks. Today, it's so diverse and complex that it can cover pretty much all system classes. Over more than half a century, the methods have focused on specific areas of application and become specialized for them. In the early days of M&S, the goal was already reached with modeling and some simulation runs. For advanced M&S applications, the most powerful computing technology was used. From this, a separate community was created that deals with high-performance computing (HPC) in combination with simulation technology. Today's M&S applications go far beyond simple simulations and usually represent complex simulation-based experiments with many thousands of simulation runs.

The methods used to model current problems must meet high demands. Not only the aspects of one domain should be covered, but also those of many domains. This is the case, for example, in automation technology. Two worlds have developed here, one dealing with continuous control engineering and the other with discrete event-oriented control engineering. From the perspective of M&S, these two worlds are continuously integrated into the two system classes. However, many practical problems involve both discrete event-oriented control and continuous control approaches. Hybrid M&S methods are needed for such combined problems. One suitable methodology for this purpose is the Discrete Event System Specification (DEVS) formalism from 1976, which was originally developed only for the discrete event-oriented system class. Over time, this formalism has been developed further in numerous ways and can now be used for many other system classes. The latest developments in this formalism have been aimed at improving the representation of components with Mealy behavior. This research led to the Revised Parallel Discrete Event System Specification (RPDEVS) formalism. However, in a critical analysis, P. Junglas demonstrated that RPDEVS still has serious problems. As a result, he proposes the theory of non-standard analysis (NSA) as a possible solution.

This paper considers two topics. (i) Junglas' approach is adopted in this work, and a complete NSA-DEVS formalism is developed. For verification purposes, a complex practical problem from the field of production engineering is modeled using NSA-DEVS. (ii) In addition, simulation studies are performed and analyzed on an HPC system using this model to verify whether NSA-DEVS can be used to perform complex and highly complex simulation experiments on systems costing up to €100,000 within an acceptable time frame.

Inhaltsverzeichnis

1 Einleitung

Die Methodik der *Modellbildung und Simulation* (M&S) ist heute von zentraler Bedeutung bei sehr vielen ingenieurtechnischen Aufgabenstellungen. Das Fachgebiet blickt inzwischen auf eine lange, in den 1960er Jahren beginnende Entwicklungsgeschichte zurück. Anfänglich ging es darum, das dynamische Verhalten von technischen Systemen mittels einzelner Simulationsläufe auf digitalen Computern mit ausreichender Genauigkeit nachbilden zu können. Mit der wachsenden Leistungsfähigkeit der Computersysteme ist diese Problemstellung für viele Anwendungsbereiche inzwischen gelöst.

In heutigen Anwendungsbereichen wie der Digitaltechnik, den Telekommunikationsnetzen, der Fertigungstechnik, Logistik und den Verkehrsnetzen geht es inzwischen aber längst nicht mehr nur um die Ausführung einzelner Simulationsläufe, sondern um komplexe Simulationsstudien, in denen eine sehr große Anzahl von Simulationsläufen auszuführen ist. Im Kontext der modernen M&S-Methodik werden solche Studien auch als *simulationsbasierte Experimente* bezeichnet. Vorgehensmodelle der M&S sehen beispielsweise vor, bei Entwurfsproblemen mit einer explorativen Analyse zu beginnen, die meist nur einige Dutzend Simulationsläufe benötigt. Anschließende Parameter-Untersuchungen wie *Screening* oder *Sensitivitätsanalysen* erfordern dagegen häufig bereits tausende bis hunderttausende Simulationsläufe. Solche Experimente werden deshalb auch als *komplex* bezeichnet. Der abschließende und zugleich anspruchvollste Schritt bei vielen Entwurfsproblemen besteht im Auffinden einer besten Lösung, was folglich ein Optimierungsproblem darstellt. Erschwerend kommt hinzu, dass es hierbei oftmals nicht nur um die Bestimmung optimaler Systemparameter geht, sondern auch um optimale Systemstrukturen. Aufgrund der sehr großen Zahl der dafür erforderlichen Simulationsläufe werden derartige Experimente auch als *hochkomplex* bezeichnet.

Bis zur Jahrtausendwende wurden diese Optimierungsprobleme in vielen Bereichen meist nur aus betriebswirtschaftlicher Sicht betrachtet, also im Sinne der Effizienz. Aufgrund der inzwischen deutlich sichtbaren Klimaproblematik sind bei der Formulierung von Zielfunktionalen neben der Effizienz aber auch der Energieeinsatz und die damit verbundenen CO_2-Emissionen ins Blickfeld gerückt. Mathematisch gesehen führt dies in das Feld der *multikriteriellen Optimierung* (Pareto-Optimierung).

Einfache simulationsbasierte Experimente sind auf der aktuellen Computertechnik in der Regel seit langem mit sequentiellen Ausführungstechniken gut beherrschbar. Völlig anders sieht es bei komplexen und hochkomplexen Studien aus. In diesem Bereich sind akzeptable Bearbeitungszeiten nur noch unter massivem Einsatz der heute durch *High Performance Computing* (HPC)-Systeme verfügbaren Parallelverarbeitungstechnologien erreichbar. In der Praxis besteht aber häufig noch das Problem,

dass viele M&S-Anwender bisher über wenig Know-How bezüglich paralleler Techniken verfügen.

Aus mathematischer Sicht umfasst die M&S ein breites Spektrum an Systemklassen. Traditionell ist die Unterteilung in *kontinuierliche* und *diskret-ereignisorientierte* Problemstellungen üblich. Dadurch entwickelten sich zunächst relativ getrennte Teilgebiete innerhalb der M&S. Insbesondere im Anwendungsgebiet der Automatisierungstechnik zeigte sich bei komplexen Problemstellungen der Regelungs- und Steuerungstechnik jedoch recht bald, dass diese scheinbar getrennten *Weltsichten* möglichst durch einen umfassenden *hybriden* Formalismus modellierbar und simulierbar sein sollten.

Aus dieser Perspektive ist der von B. P. Zeigler 1976 vorgeschlagene Formalismus *Discrete Event System Specification* (DEVS) besonders interessant. Wie bereits die Namensgebung anzeigt, hat dieser Formalismus seinen Ursprung zunächst im Bereich der diskret-ereignisorientierten Systemklasse. Er schließt gut nachvollziehbar an frühere Arbeiten von Turing und Moore an, aber weniger an die von Mealy. Turing, Moore und Mealy beschäftigten sich in den 1940er beziehungsweise 1950er Jahren mit der formalisierten Spezifikation von Schaltnetzwerken, was heute als Automatentheorie bezeichnet wird.

In den 1980er und 1990er Jahren kann insbesondere H. Prähhofer aufzeigen, dass Zeiglers DEVS-Formalismus sehr gut auch auf kontinuierliche Systeme und weitere Systemklassen erweiterbar ist. Um die Jahrtausendwende herum entwickelte E. Kofmann dann den *Quantized State System* (QSS) Ansatz zur M&S hybrider Systeme auf Basis von DEVS. In diesen Jahrzehnten wurden auch Arbeiten durchgeführt, um Zeiglers Formalismus hinsichtlich der inzwischen allgemein verfügbaren Parallelrechentechnik weiterzuentwickeln. Das meistbeachtete Ergebnis dieses Entwicklungsstrangs war die *Parallel Discrete Event System Specification* (PDEVS). Wegen der vielen Weiterentwicklungen wurde in dieser Zeit zwecks besserer Unterscheidung für Zeiglers ursprünglichen Formalismus die Bezeichnung Classic DEVS eingeführt.

Spätestens ab 2007 widmeten sich mehrere Autoren dem Problem, dass diskret-ereignisorientierte Systeme mit Mealy-Charakteristik in Classic DEVS und den Weiterentwicklungen wie PDEVS zuvor nur wenig Beachtung fanden. Im Ergebnis entstand unter anderem der von F. J. Preyser ausgearbeitete Formalismus *Revised Parallel Discrete Event System Specification* (RPDEVS).

2020 wies Junglas in direkter Diskussion mit Preyser und weiteren DEVS-Experten auf dem 25. ASIM Symposium Simulationstechnik darauf hin, dass auch RPDEVS bezüglich der Mealy-Unterstützung noch ungenügend sei und schlug einen Versuch zur Lösung der noch vorhandenen Probleme über die Theorie der *Non-Standard Analysis* (NSA) unter Verwendung des hyperreellen Zahlenraumes ${}^*\mathbb{R}$ vor.

Diese Arbeit widmet sich zwei Fragestellungen: (i) Zunächst soll geprüft werden, ob auf Basis des NSA-Vorschlags von Junglas ein vollständiger DEVS-Formalismus entwickelt werden kann. Dies umfasst eine Systemspezifikation inklusive Abarbeitungsalgorithmus sowie die Implementierung eines entsprechenden Simulators. (ii) Unter der Annahme, dass die Beantwortung der ersten Fragestellung positiv ausfällt, soll

im Anschluss untersucht werden, ob mit einem NSA-DEVS-Simulator unter Verwendung eines HPC-Systems mittlerer Größe (max. 100.000 Euro) auch praxisrelevante komplexe und hochkomplexe Simulationsstudien in akzeptabler Zeit durchgeführt werden können.

M&S-Anwender sind oftmals keine Experten auf den Gebieten der Computerarchitekturen sowie der verteilten und parallelen Programmierung. Deshalb wird im zweiten Kapitel dieser Arbeit zunächst auf wichtige Grundlagen der Parallelverarbeitung auf Hard- und Softwareebene sowie auf die Leistungsbewertung paralleler Systeme eingegangen.

Das dritte Kapitel befasst sich mit der Beschleunigung von *Discrete Event System* (DES)-Studien mittels Verteilung und Parallelverarbeitung auf den Ebenen: einzelner Simulationslauf, mehrere Simulationsläufe bis hin zur Ebene komplexer beziehungsweise hochkomplexer Simulationsstudien.

Im vierten Kapitel wird in die für diese Arbeit wichtigsten DEVS-Formalismen eingeführt. Dabei werden ausgehend von Classic DEVS Weiterentwicklungen bis zum aktuellen Stand der Forschung berücksichtigt.

Das fünfte Kapitel stellt die Ergebnisse der Bearbeitung der ersten Forschungsfrage dar, ob ein vollständiger Formalismus auf Basis der NSA-Idee entwickelt werden kann. Eine erste prototypische Simulator-Implementierung zur Verifikation wird in diesem Kapitel ebenfalls vorgestellt.

Da die Untersuchung der ersten Forschungsfrage positiv ausfiel, wurde im Anschluss die zweite Fragestellung geprüft, ob mit einem NSA-DEVS-Simulator unter Verwendung eines mittelgroßen HPC-Systems auch komplexe und hochkomplexe Simulationsstudien in akzeptabler Zeit durchgeführt werden können. Die Ergebnisse dieser Untersuchung werden im sechsten Kapitel dargestellt.

Im abschließenden siebten Kapitel erfolgt eine Zusammenfassung der Arbeit und es werden Anregungen für weitere Untersuchungen des Themas gegeben.

2 Grundlagen der Parallelverarbeitung

Aus Anwendersicht wird der Begriff Parallelverarbeitung zunächst meist mit der beschleunigten Lösung eines gegebenen Problems assoziert. In der ingenieurtechnischen Praxis geht es aber auch sehr häufig darum, mit der aktuell verfügbaren Computertechnik ein gegebenes Problem in der verfügbaren Zeit mit der erforderlichen Genauigkeit oder allgemeiner gesprochen: mit der gewünschten Komplexität zu lösen.

Aus Sicht der technischen Informatik ist Parallelverarbeitung eine Technik, die bei Computerarchitekturen eingesetzt wird, um die Leistungsfähigkeit von Computersystemen zu erhöhen.

Der Schwerpunkt dieser Arbeit liegt auf dem Einsatz der Parallelverarbeitung zur Lösung einer simulationstechnischen Problemklasse (DES-Studien) unter Nutzung aktueller, allgemein verfügbarer Computertechnik.

In diesem Kapitel werden die Grundlagen der Parallelverarbeitung deshalb nur in einem Maße besprochen, wie es für das Verständnis des restlichen Teils der Arbeit erforderlich ist. Zunächst werden die Relationen zwischen sequentieller, paralleler und verteilter Verarbeitung näher betrachtet. Anschließend wird ein Überblick zur Entwicklung der Hardwareebene, also zu wesentlichen Aspekten der Computerarchitekturen gegeben. Danach folgt die Softwareebene: Auf dieser Ebene werden sowohl relevante Programmiersprachen und Middleware-Ansätze als auch das *Scientific Computing Environment* (SCE) Matlab charakterisiert und eingeordnet. Abschließend wird auf die wesentlichen Leistungsmaße im Zusammenhang mit der Parallelverarbeitung aus Anwenderperspektive eingegangen.

2.1 Parallele versus verteilte Verarbeitung

In der Frühzeit der Computertechnik waren die Parallelverarbeitung und verteilte Systeme noch völlig getrennte Fachgebiete. Insbesondere im Bereich der Simulationstechnik kommt es in den 1980er Jahren aber zu einer starken Annäherung und teilweisen Verschmelzung beider Disziplinen. In dieser Zeit werden Begrifflichkeiten geprägt, die sich später bei der Erarbeitung klar verständlicher Klassifikationen als sehr problematisch erweisen. Konkret geht es um die Begriffe *parallele Simulation* und *verteilte Simulation*.

Die Abgrenzung beider Begriffe erfolgt zunächst nicht auf der Anwendungsebene, also aus simulationstechnischer Sicht, sondern allein auf Basis der eingesetzten Computerarchitektur. Kommt eine Struktur mit gemeinsamen Speicher zum Einsatz,

wird von paralleler Simulation gesprochen. Wird eine Struktur mit verteiltem Speicher benutzt, wird von verteilter Simulation gesprochen. Im letzteren Fall kann es sich sowohl um eine tatsächlich verteilte Anwendung wie *Simulatorkopplung*, *Hardware in the Loop*, *Software in the Loop* oder Ähnliches handeln, aber auch um eine gewöhnliche Simulation, die zwecks Beschleunigung (also Parallelverarbeitung) auf einer lose gekoppelten Plattform ausgeführt wird.

Gemäß Fujimoto [32] wird eine verteilte Simulation definiert als eine Simulation, die auf einem Verbund von lose gekoppelten Prozessoren ausgeführt wird, die über ein *Wide Area Network* (WAN) oder ein *Local Area Network* (LAN) miteinander verbunden sind. Eine parallele Simulation definiert Fujimoto als eine Simulation, die auf einem eng gekoppelten Verbund von Prozessoren, die über ein Hochgeschwindigkeitsnetzwerk kommunizieren, ausgeführt wird. Diese Definitionen lösen das Problem der Unschärfe des Begriffs verteilte Simulation aber in keiner Weise. Bereits in den 1990er Jahren waren LAN-Technologien so leistungsfähig, dass darauf aufbauende Plattformen zum Zwecke der Beschleunigung von Simulationen eingesetzt wurden und noch heute werden.

Pawletta [69] setzt sich mit der problematischen Begriffsbildung im Bereich der Simulationstechnik bereits 1998 auseinander. In seiner Arbeit verweist er auf die Klassifikation von Carriero und Gelernter [10]. Diese unterscheiden zunächst zwischen der sequentiellen Lösung eines gegebenen Problems und der koordinierten Lösung eines in Teilaufgaben gegliederten Problems. Bei der Zerlegung eines Problems in Teilaufgaben unterscheiden sie zwischen zeitlicher und örtlicher Separierung. Über diesen Ansatz kommen sie zu einer Klassifikation, bei der die parallele Verarbeitung ein Teilgebiet der verteilten Verarbeitung ist.

In dieser Arbeit wird der Begriff *parallele Simulation* im Kontext einer angestrebten Beschleunigung verwendet. Lösungsansätze aus dem Bereich der *verteilten Simulation* können auch in diesen Bereich fallen, wenn diese das Ziel einer Beschleunigung haben.

2.2 Hardwareebene

Eine sehr gute Darstellung der frühen Entwicklung der Computerarchitekturen findet sich in [58]. In diesem Werk beschreiben Märtin et al. wie 1944 erstmals die Architektur eines sogenannten *Universalrechners* durch eine Gruppe bestehend aus von Neumann, Eckert, Mauchly und Goldstine entwickelt und zunächst nur in einem internen Papier dokumentiert wurde. Inzwischen ist dieser Entwurf allgemein unter dem Namen *Von-Neumann-Rechner* bekannt [9, 66]. Praktisch alle heutigen Universalrechner sind das Ergebnis eines bereits 8 Dekaden andauernden Entwicklungsprozesses, der bei diesem Entwurf seinen Anfang nahm. Auch wenn die Komplexität heutiger Computersysteme gegenüber dem Von-Neumann-Rechner enorm angestiegen ist, basieren sie immer noch auf den in [66] für notwendig erachteten fünf Grundelementen: *Central Arithmetical* (CA), *Central Control* (CC), *Memory* (M), *Input* (I) und *Output* (O).

Der Von-Neumann-Rechner ist zunächst ein rein sequentieller Datenverarbeitungsansatz. Bereits in den 1960er Jahren wird in den tatsächlich gebauten Computern jedoch ein erheblicher Grad an Parallelverarbeitung realisiert. Diesen Stand der Technik analysiert Flynn in [26]. Als Ergebnis seiner Analyse schlägt er eine sehr einfache Klassifikation der Rechnerarchitekturen seiner Zeit vor, die auch zum zuvor entwickelten Von-Neumann-Modell anschlussfähig ist. In den Mittelpunkt stellt er die *Execution Unit*. Diese korrespondiert mit dem Zusammenschluss von CA und CC (vgl. [66]). In der sekundären Literatur wird dafür später auch häufig der Begriff Verarbeitungseinheit (*Processing Element*, PE) benutzt. M, I und O spielen bei Flynn keine primäre Rolle. Vielmehr entwickelt er die Abstraktion, dass mittels einer PE genau ein Datenstrom (*Single Data Stream*) gemäß eines Befehlsstroms (*Single Instruction Stream*) bearbeitet wird. In seiner Analyse stellt er fest, dass die Computer in der Mitte der 1960er Jahre in diesem Sinne offensichtlich bereits mehrere PEs enthalten, weil sie teilweise in der Lage sind, auch multiple Datenströme und/oder Befehlsströme zu verarbeiten. An diesem Fakt orientiert, stellt er eine Taxonomie auf, die aus nur vier Klassen besteht:

Single Instruction Single Data (SISD)

In diese Kategorie ordnet sich der Von-Neumann-Rechner mit seiner rein sequentiellen Verarbeitungsfähigkeit ein. In heutiger Zeit trifft man solche Systeme nur noch bei sehr einfachen Mikrocontrollern an.

Single Instruction Multiple Data (SIMD)

In dieser Kategorie gibt es seit den 1960er Jahren eine Vielzahl von realisierten Systemen, die einen hohen Grad an Parallelverarbeitung ermöglichen. Beispiele in heutiger Zeit sind Grafikkarten, Tensor-Chips, Signalprozessoren und vieles mehr. Häufig verwendete synonyme Begriffe zu SIMD sind *synchrone Parallelverarbeitung* und *Datenparallelität*.

Multiple Instruction Single Data (MISD)

Dieses Prinzip spricht Flynn in [26] nur kurz an und führt aus, dass es bis 1966 nur wenig Aufmerksamkeit erfahren hat. Trotzdem führt er zumindest ein theoretisches Beispiel auf. In den folgenden fast 60 Jahren äußern sich viele Autoren zu dieser Klasse. Einige meinen, dass es überhaupt keine Realisierungen dieses Prinzips gibt, einige bringen die über bereits lange Zeit in vielfältiger Form realisierten Pipelinetechniken auf der Prozessorebene mit diesem Prinzip in Verbindung. Für eine weitere Exploration zu Sinn oder Unsinn dieser Klasse soll an dieser Stelle nur auf die weiteren originären Publikationen von Flynn [25, 27] sowie auf sekundäre Publikationen wie beispielsweise von Hoffmann [41] und Wijtvliet et. al [102] verwiesen werden.

Multiple Instruction Multiple Data (MIMD)

In dieser Kategorie gibt es bis heute ebenfalls eine Vielzahl realisierter Systeme mit einem hohen Parallelitätsgrad. Dabei fiel und fällt die Leistungsfähigkeit der einzelnen PEs durchaus sehr unterschiedlich aus. Charakteristisch für diese Realisierungen

ist aber, dass die in kleinerer oder auch sehr großer Zahl vorhanden PEs grundsätzlich nicht im Gleichtakt arbeiten müssen. Ein häufig verwendetes Synonym zu MIMD ist deshalb auch der Begriff *asynchrone Parallelverarbeitung*. Beispiele in heutiger Zeit sind äußerst vielfältig und reichen aktuell von Multicoreprozessoren über Multiprozessorsystemen bis hin zu Clustern von Computersystemen verbunden über das Internet.

Die Klassifikation nach Flynn ist heute in gewisser Weise historisch, weil sich die modernen Systeme selten eindeutig einer der obigen Klassen zuordnen lassen. Selbst auf der aktuellen Prozessor- beziehungsweise Chipebene werden inzwischen viele Techniken der Parallelverarbeitung in Kombination realisiert. In diesem Sinne stellen heutige Computerarchitekturen bis auf immer seltenere Ausnahmen *hybride Systeme* dar. Trotzdem sind insbesondere die Klassenbezeichnungen SIMD und MIMD immer noch stark im Gebrauch, um gerade die teilweise sehr komplexe hybride Struktur der aktuellen Computersysteme verständlich zu machen.

Ein Grund für die vergleichsweise lang anhaltende Relevanz der Klassifikation von Flynn ist ganz sicher ihr hoher Abstraktionsgrad. Einige Autoren verwendeten die Kernideen dieser Hardwareklassifikation sogar, um langzeitgültige Klassifikationen für die Softwareebene zu schaffen [69, 23].

Ein weiterer Grund die Klassifikation von Flynn auch in dieser Arbeit erneut aufzugreifen ist der Titel der originalen Veröffentlichung von 1966: *Very High-Speed Computing Systems*. Dieser Begriff war bereits damals unscharf und ist heute nicht mehr im Gebrauch. Er wurde inzwischen ersetzt durch den ebenfalls unscharfen Begriff: Hochleistungsrechnen (*High Performance Computing*, HPC).

Vor und insbesondere nach [26] wurden natürlich noch eine sehr große Zahl an Arbeiten zu Hardware-Klassifikationssystemen veröffentlicht. Beispielhaft sollen hier nur zwei erwähnt werden: An erster Stelle das *Erlangen Classification System* (ECS) [4]. Bei der ECS-Klassifikation wird quantitativ die Nebenläufigkeit und das Pipelining auf der Leitwerks-, Rechenwerk- und Wortebene erfasst. Dabei entsteht ein ECS-Tripel, welches den jeweiligen Rechner beschreibt. An zweiter Stelle sei hier auf die Hardwareklassifikation von Tanenbaum und Austin [89] hingewiesen, die in Abbildung 1 dargestellt ist. Ihre Veröffentlichung liegt inzwischen ebenfalls eine Dekade zurück, bietet aber immer noch eine sehr einfache und gut verständliche Brücke zwischen der klassischen Flynn-Klassifikation und den derzeit realen Computerarchitekturen.

In den folgenden Abschnitten werden drei Bereiche der Entwicklung der Computerarchitekturen vertieft besprochen, die gegenwärtig allgemein beziehungsweise insbesondere im Kontext dieser Arbeit besonders relevant sind. Dies ist zunächst das Konzept der Coprozessoren, gefolgt vom Ansatz der Multiprozessor- und Multicorearchitekturen. Abschließend wird auf Multicomputer, die heute meist als Computercluster bezeichnet werden, eingegangen.

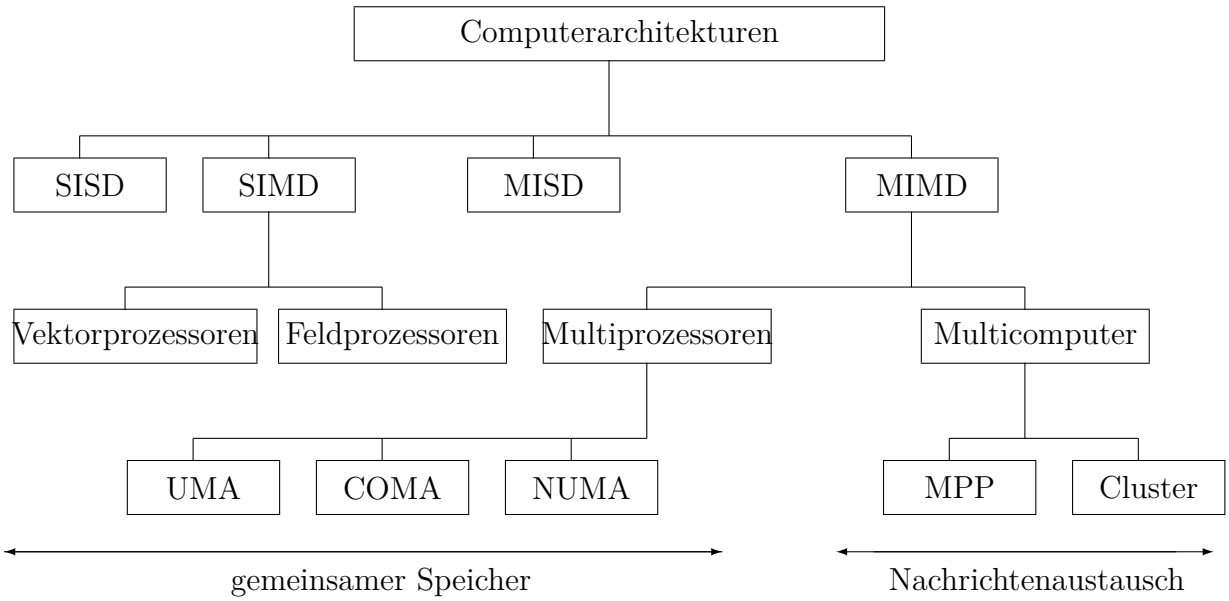


Abbildung 1: Computerarchitekturen nach [89].

2.2.1 Coprozessoren

Coprozessoren werden bereits sehr lange zur Leistungssteigerung von Computersystemen eingesetzt. Tanenbaum und Austin führen in [89] als frühe Beispiele die I/O-Coprozessoren der Mainframes auf. In den ersten zwei Jahrzehnten der Mikroprozessor-Ära werden mathematische Coprozessoren als Beschleuniger für Gleitkommaoperationen eingesetzt, wie beispielsweise die *Floating Point Unit* (FPU) der x87-Coprozessoren von Intel.

Seit Mitte der 1990er Jahren spielen *Graphics Processing Units* (GPUs) als Coprozessoren eine immer wichtigere Rolle. Anfänglich waren diese ausschließlich für Grafikoperationen zuständig. Mit der Entwicklung von 3D-Anwendungen werden die Ansprüche an Grafikkarten aber immer größer. Infolge werden Grafikkarten mit stark erhöhter Leistungsfähigkeit und Funktionalität entwickelt. Dieses Marktsegment wird bis heute durch die Hersteller Nvidia und AMD dominiert.

In den 2000er Jahren werden erstmals programmierbare GPUs auf den Markt gebracht, die über Grafikanwendungen hinaus auch für andere rechenintensive, insbesondere hochgradig datenparallele Problemstellungen eingesetzt werden können. Im Zuge dieser Entwicklung kommt zusätzlich zu GPU der neue Begriff *General Purpose Computation on Graphics Processing Unit* (GPGPU) auf. Inzwischen wurden mehrere Programmierschnittstellen entwickelt. Ein herstellerübergreifender Zugang ist *Open Computing Language* (OpenCL), der neben GPUs auch Central Processing Units (CPUs) und weitere Coprozessoren wie zum Beispiel *Digital Signal Processor* (DSP) mit einbezieht. OpenCL wird sowohl von Nvidia als auch von AMD unterstützt. Neben OpenCL hat auch die *Compute Unified Device Architecture* (CUDA)

große Bedeutung. Allerdings ist CUDA ein propritärer Ansatz von Nvidia, der nur für GPUs dieses Herstellers verwendbar ist.

Das enorme Potential für datenparallele Problemstellungen von GPUs machen Tanenbaum und Austin bereits 2014 in [89] sehr anschaulich. Sie vergleichen die Geforce GTX-580 von 2010 mit dem Supercomputer Cray-2 von 1990 hinsichtlich der Gleitkommarechenleistung und der erforderlichen elektrischen Leistung. Die Cray-2 erreicht 0,002 Teraflops und benötigt dazu 150.000 Watt. Die GTX-580 erreicht 1,5 Teraflops und benötigt dafür nur 250 Watt.

Die im Vergleich betrachtete GTX-580 ist eine Realisierung der Fermi-Architektur von Nvidia. Diese Architektur kann primär der Klasse SIMD nach Flynn zugeordnet werden. 32 CUDA-Kerne bilden in der Fermi-Architektur einen sogenannten *Streaming Multiprocessor* (SM). Die Fermi-Architektur besitzt 16 SMs, die über einen gemeinsamen Speicher verbunden sind. Insgesamt sind in der Fermi-Architektur damit 512 PEs enthalten.

Die ebenfalls bedeutsamen GPUs des Herstellers AMD werden wie bereits besprochen über OpenCL programmiert. Sie sind primär auch der Klasse SIMD nach Flynn zuzuordnen. Die Komponentenbezeichnungen bei AMD folgen der OpenCL-Terminologie. Demzufolge wird die kleinste Verarbeitungseinheit als *Processing Element* (PE) bezeichnet. Eine Gruppe von PEs bildet eine *Compute Unit* (CU). Da OpenCL neben GPUs auch CPUs und andere Recheneinheiten mit einbezieht, gibt es hier noch den Begriff des *Compute Device* (CD), mit dem sich eine bestimmte Recheneinheit spezifizieren lässt.

2.2.2 Multiprozessor- und Multicorearchitekturen

Das Konzept der Multiprozessoren wird bereits seit den 1960er Jahren zur Leistungssteigerung von Computersystemen eingesetzt [26, 58, 89]. Im Unterschied zum Von-Neumann-Rechner besitzen solche Systeme mehrfache CPUs (CC,CA). Nach Flynn können diese Systeme zunächst eindeutig der MIMD-Klasse zugeordnet werden. Wenn die CPUs über einen gemeinsamen Speicher zusammenarbeiten können, spricht man von eng gekoppelten MIMD-Systemen. Bei diesem Prinzip kann die Anzahl der CPUs allerdings nicht beliebig skaliert werden. Erfolgt die Zusammenarbeit der CPUs über Kommunikationslinks, dann handelt es sich um ein lose gekoppeltes MIMD-System. Solche Systeme können technisch gesehen beliebig skaliert werden. Heutige Multiprozessorsysteme sind in der Regel hochgradig hybrid, dass heißt, sie arbeiten sowohl mit enger Kopplung auf unterer Ebene als auch loser Kopplung auf höherer Ebene. Darüber hinaus werden CPUs inzwischen mit GPUs und anderen spezialisierten Coprozessoren in Kombination eingesetzt.

Die in Multiprozessorsystemen eingesetzten CPUs waren nach [89] bis zur Jahrtausendwende im Wesentlichen um einige SIMD-Fähigkeiten erweiterte Ein-Chip-Prozessoren vom SISD-Typ. Als bedeutender CPU-Hersteller führte Intel in den 2000er-Jahren zwei neue Technologien ein, mit denen MIMD-Parallelität auf der Chip-Ebene Einzug hielt. Da der Begriff Chip missverständlich sein kann, wird im

Folgenden bevorzugt der eindeutigere Begriff *Die* benutzt. Im ersten Schritt führte Intel das On-Chip-Multithreading ein. Intel selbst verwendet den Begriff Hyperthreading. Darunter versteht man die Einführung mehrerer Piplines, wodurch die Auslastung der restlichen CPU-Hardware gesteigert werden kann. Tanenbaum und Austin führen in [89] aus, dass Intel 2002 erstmals beim Xeon-Prozessor mittels einer Dual-Pipeline bei nur 5%iger Vergrößerung der Die-Fläche bei vielen Anwendungen einen 25%igen Leistungsgewinn erzielen konnte. Wenig später wurde der zweite Schritt mit der Einführung der Ein-Chip-Multiprozessor-Technologie vollzogen [89]. Hierbei werden auf einem Die mehrere vollständige CPUs integriert, die dann als Kerne beziehungsweise Cores bezeichnet werden. Bereits mit zwei Kernen kann die Prozessorleistung verdoppelt werden. Allerdings ist dafür auch eine Verdopplung der Die-Fläche erforderlich. Die Ein-Chip-Multiprozessor-Technologie wird zumindest bei Intel-Prozessoren häufig mit dem On-Chip-Multithreading in Kombination eingesetzt. Unabhängig davon, ob On-Chip-Multithreading integriert ist oder nicht, hat sich heute der Begriff Multicore-CPUs als Bezeichnung für Ein-Chip-Multiprozessoren durchgesetzt.

Moderne Multicore-CPUs sind mit bis zu 128 Kernen ausgestattet [1]. Um die Menge an Daten vom Hauptspeicher zu den Kernen übertragen zu können, wird auf bewährte Techniken der eng gekoppelten Multiprozessor-Architekturen aus den 1990er Jahren zurückgegriffen. Beim zweiten bedeutenden Prozessorhersteller AMD handelt es sich dabei um Uniform Memory Access (UMA) und Non-Uniform Memory Access (NUMA). Während UMA und NUMA zunächst nur für sehr kostenintensive Hochleistungsrechner zur Verfügung stand, werden diese Techniken inzwischen teilweise auch in kostengünstigen Standardsystemen eingesetzt. Beispielsweise verwendet AMD für seine EPYC-Prozessoren NUMA zur Hauptspeicheranbindung. Bei diesen Prozessoren wurde nunmehr auch der Schritt von der Ein-Chip- zur Multi-Chip-Technologie vollzogen. Das heißt, ein EPYC-Prozessor kann bis zu 8 Dies mit jeweils mehreren Kernen enthalten. Die Kommunikation zwischen den Dies sowie für den Multi-Socket-Betrieb wird über Infinity Fabric (IF) realisiert. Die Multi-Chip-Prozessoren enthalten einen Memory-Controller, der 2 mal 6 Speicherkanäle besitzt. Damit können 6 TB DDR5 Hauptspeicher pro Prozessor installiert werden. Der gesamte Speicher ist auf 4 NUMA-Knoten aufgeteilt, wobei die Kerne und der Hauptspeicher innerhalb eines Knotens eine feste Zuordnung besitzen. Innerhalb eines Knotens ist die Latenz bei Speicherzugriffen am geringsten. Greift ein Kern auf einen Adressbereich eines anderen NUMA-Knoten zu, dann erhöht sich die Latenz.

Die modernen Multicore-Technologien können natürlich auch in Kombination mit der klassischen Multiprozessor-Architektur eingesetzt werden. Für erhöhte Leistungsanforderungen stehen beispielsweise Dual-Socket-Mainboards zur Verfügung, die bei einer Bestückung mit beispielsweise zwei entsprechenden EPYC-Prozessoren ein System mit 256 Kernen ergeben.

2.2.3 Cluster

In der Fachliteratur werden Computercluster zunächst als Multicomputer bezeichnet und später als Teilklasse der *Multicomputer* neben *Massively Parallel Processing Systems* (MPPs) eingeordnet [89]. Multicomputer erscheinen etwas später als Multiprozessor-Systeme und können wie diese zumindest bezüglich ihrer Oberstruktur in die MIMD-Klasse eingeordnet werden. Im Unterschied zu Multiprozessor-Systemen handelt es sich bei Multicomputern und damit auch bei Clustern praktisch immer um lose gekoppelte MIMD-Systeme. Ein Computercluster ist kein einzelner Rechner mehr, sondern ein Rechnerverbund. Kleinere Systeme bestehen aus einer einstelligen bis niedrigen zweistelligen Zahl von Rechnern, die im einfachsten Fall einfach übereinandergestapelt werden. In mittleren bis großen Systemen sind mehrere dutzend bis zehntausende Rechner installiert. In der obersten Leistungsklasse, also bei Supercomputern, überschreiten aktuelle HPC-Systeme inzwischen die Millionengrenze, wobei die Unterscheidung zwischen Clustern und klassischen MPPs praktisch obsolet geworden ist.

Da in Computerclustern auf Grund des Fehlens eines gemeinsamen Speichers die Kommunikation zwischen den Einzelrechnern über ein Verbindungsnetzwerk erfolgen muss, entscheidet dessen Latenz und Transferleistung ganz wesentlich über die Gesamtleistung des Systems. Inzwischen ist bereits die weit verbreitete und kostengünstige Ethernet-Technologie relativ leistungsfähig und wird als Verbindungsnetzwerk für Cluster der untersten Leistungsklasse benutzt. Für höhere Anforderungen müssen aber nach wie vor kostenintensivere Netzwerk-Technologien eingesetzt werden. In diesem Bereich spielen neben anderen insbesondere *InfiniBand* (Nvidia, früher Melanox), *Slingshot* (Hewlett Packard Enterprise) und *Omni-Path* (Intel) eine wichtige Rolle.

Seit vielen Jahren setzen alle Hersteller der leistungsstärksten Supercomputer ausschließlich das Open-Source-Betriebssystem Linux ein [91]. Um diese Entwicklung besser zu koordinieren und auch auf weniger leistungsstarke HPC-Systeme auszudehnen, wurde auf der Supercomputing-Konferenz 2015 unter dem Dach der *Linux Foundation* unter Beteiligung vieler Hardware- und Software-Hersteller die *OpenHPC*-Initiative gegründet. Im Rahmen dieses Projekts wurden unter dem Titel *Cluster Building Recipes* inzwischen mehrere Versionen von Leitlinien für den möglichst einheitlichen Aufbau von clusterbasierten HPC-Systemen ausgearbeitet und veröffentlicht [21]. Danach sollte der grundlegende Aufbau eines HPC-Clusters wie in Abbildung 2 dargestellt erfolgen. Die zentralen Komponenten sind die über ein Hochgeschwindigkeitsnetzwerk verbundenen Computenodes, ergänzt um einen Master, der die externe und interne Steuerung über ein Standardnetzwerk realisiert. In der Regel stellen Cluster nicht nur hohe Rechenleistungen zur Verfügung, sondern auch große sekundäre Speicherkapazitäten, die mittels eines parallelen Filesystem den Computenodes direkt zugänglich gemacht werden.

Im Sinne von OpenHPC sind für den Betrieb von Clustern insbesondere die Linux-Distributionen SUSE Linux Enterprise, OpenSUSE oder Cent-OS geeignet. Oberhalb des Betriebssystems werden als weitere Softwareinfrastruktur Warewulf oder

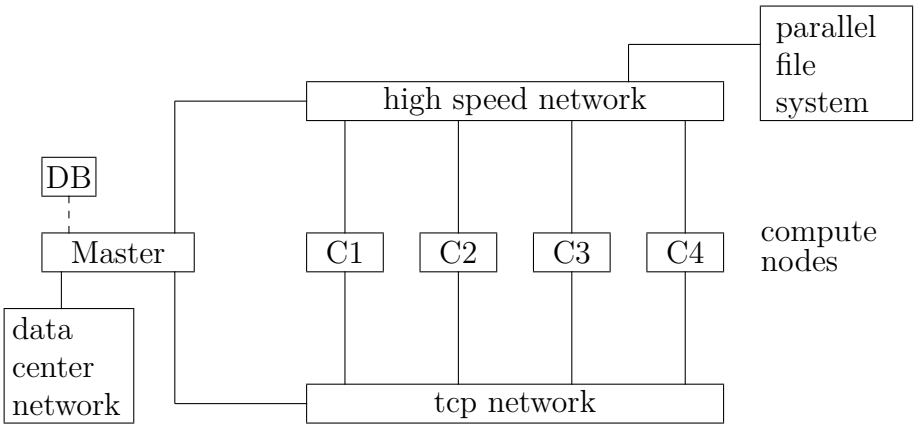


Abbildung 2: OpenHPC Cluster-Architektur nach [21].

xCAT für die Verwaltung und SLURM oder PBS Professional als Jobmanger empfohlen. Für das Monitoring können nach OpenHPC Nagios oder Ganglia eingesetzt werden. Bezüglich der Anwendungsentwicklung setzt OpenHPC auf alle populären MPI-Stacks, Bibliotheken und Compiler. Auf einige dieser Software-Komponenten geht der nachfolgende Abschnitt näher ein.

2.3 Softwareebene

Vor dem Hintergrund des Entwicklungsstandes der Computerarchitekturen in der ersten Hälfte der 1990er Jahre entwickelt Ungerer in [93, 94] ein Modell mit fünf Ebenen, auf denen aus softwaretechnischer Sicht Parallelverarbeitung in sehr unterschiedlicher Weise genutzt werden kann. Er konzentriert sich dabei auf die vorwiegend eingesetzten Rechnerinstallationen, die mit Multi-Task-Betriebssystemen in Verbindung mit Compilern für Hochsprachen betrieben werden. Dieses Modell wird auch nach der Jahrtausendwende noch von vielen Autoren verwendet, beispielsweise in [58, 23]. Im Verlauf von inzwischen 3 Dekaden wurden die Computerarchitekturen erheblich weiterentwickelt (s. Abschn. 2.2). Ungerers Annahmen zum Betrieb üblicher Rechnerinstallationen sind in etwas erweiterter Form aber auch heute noch gültig. Nach wie vor werden diese durch Betriebssysteme gesteuert, die aufgrund der erfolgreichen POSIX-Standardisierung in ihren Kerntechnologien sehr konstant geblieben sind. Im Bereich der Übersetzungssysteme sind Compiler im HPC-Bereich immer noch dominant. Daneben spielen aber Interpreter für Hochsprachen inzwischen ebenfalls eine gewisse Rolle. Nachfolgend wird in die Ebenen des Ungerer-Modells aus heutiger Perspektive, beginnend mit der höchsten Ebene, kurz eingeführt:

1. *Programm-* oder *Jobebene*: Auf dieser Ebene werden mehrere Programme eines komplexen Problems in Form separater Betriebssystemprozesse parallel ausgeführt. Diese Ebene stellt die grobkörnigste Variante der Parallelverarbeitung dar. Eine Kommunikation zwischen den parallelen Prozessen ist in der Regel nicht erforderlich. Bei geeigneter Anwendungsstruktur können auf dieser Ebene hohe Beschleunigungen, selbst bei MIMD-Strukturen mit Kommunikations-

Links via Internet, erzielt werden. In der Literatur wird diese Form der Parallelverarbeitung von einigen Autoren als *embarrassingly parallel* bezeichnet, beispielsweise in [58].

2. *Prozessebene*: Diese Ebene ist bei lokalen, also nicht über das Internet vernetzten MIMD-Strukturen relevant. Sie ist geeignet für ein *einzelnes* Programm mit inhärenter, grobkörniger Parallelität, dass sich zur Ausführung in mehrere *kooperierende* Prozesse aufspalten lässt. Im Unterschied zur Programm- oder Jobebene ist der Kommunikations- und Synchronisationsbedarf erhöht, aber relativ zum Rechenbedarf immer noch gering. Bei Multicomputern wird die notwendige Kommunikation und Synchronisation ausschließlich über ein in der Regel vorhandenes Hochgeschwindigkeitsnetzwerk realisiert. Mehrprozessor- beziehungsweise Multi-Core-Systeme verfügen heute meist über ein Gemisch aus schnellen Links und gemeinsamen Speicher. Diese Hardware-Ressourcen werden auf der Software-Ebene über die im Betriebssystem implementierten Techniken der sogenannten *Interprozesskommunikation* verfügbar gemacht. Das Konzept kooperierender Betriebssystemprozesse ist weitgehend unabhängig von der eingesetzten Programmiersprache und wird traditionell als *schwergewichtige* Prozesse bezeichnet.

3. *Block-* oder *Threadebene*: Diese Ebene der Parallelverarbeitung ist primär ebenfalls auf MIMD-Strukturen fokussiert. Bei MIMD-Strukturen mit gemeinsamen Speicher können auf dieser Ebene auch Anwendungen mittlerer Granularität mit Erfolg parallelisiert werden. Das Konzept wird traditionell als *leichtgewichtige* Prozesse bezeichnet und wurde zunächst nur durch das Laufzeitsystem einer Hochsprache innerhalb des Adressraums eines Betriebssystemprozesses realisiert. Moderne Betriebssysteme wie zum Beispiel Linux unterstützen diesen Ansatz inzwischen aber auch auf der Kernel-Ebene.

4. *Anweisungsebene*: Diese Parallelitätsebene stellt auch heute noch auf das Innere eines Prozessors ab. Dabei handelt es sich um verschiedenste Technologien wie VLIW, Superskalarität, EPIC und weitere. Im Fachgebiet besteht Konsens, dass es sich dabei nicht mehr um MIMD-Strukturen handelt. Häufig werden diese Techniken in Abgrenzung zu MIMD grob unter SIMD subsummiert. Im Sinne der von Flynn eingeführten Hardwareklassifikation ist dies bei strenger Betrachtung allerdings fragwürdig. Entscheidender ist aber, dass auf der Anweisungsebene tatsächlich eine erfolgreiche Beschleunigung von feingranulären Anwendungen ermöglicht wird, was mittels MIMD-Strukturen zuvor nicht gelang. Die Ausnutzung der Anweisungsebene im Sinne der Parallelverarbeitung ist in der Regel kein expliziter Gegenstand auf der Ebene der Programmierung und des Betriebssystems. Die effiziente Nutzung des Potenzials der Anweisungsebene der heutigen Hardware hängt entscheidend von den unterstützten Funktionalitäten des eingesetzten Compilers ab.

5. *Suboperationsebene*: Mit dieser untersten Ebene hat Ungerer bereits 1993 SIMD-Architekturen im Blick, die damals noch als eigenständige Rechnersysteme realisiert wurden. Diese waren bereits geeignet, Anwendungen mit sehr kleiner

Granularität erfolgreich zu parallelisieren. Eigenständige SIMD-Rechner werden inzwischen nicht mehr produziert. Das SIMD-Prinzip lebt heute aber in den sogenannten SIMD-Funktionseinheiten von gewöhnlichen CPUs und insbesondere in Co-Prozessoren wie GPUs (s. Abschn. 2.2.1) fort. Diese Ebene hat in der jüngeren Zeit gerade bei Anwendungen der sogenannten künstlichen Intelligenz (neuronale Netze, maschinelles Lernen u.ä.), die durch sehr feine Granularität geprägt sind, zu bemerkenswerten Durchbrüchen geführt. Wie schon auf der darüberliegenden Anweisungsebene, wird der Erfolg der Parallelisierung auch hier maßgeblich durch die Funktionalität der eingesetzten Compiler-Implementierung bestimmt.

Die Hauptkriterien der Parallelitätsebenen nach Ungerer sind offensichtlich die Hardware-Fähigkeiten bezüglich der Granularitäts-Charakteristik von Anwendungen. Über Jahrzehnte wurde innerhalb der Informatik sehr viel Arbeit investiert, Modelle und Methoden zu entwickeln, mit denen sich das Verhältnis von Kommunikations- und Synchronisationsbedarf gegenüber dem Rechenbedarf von Anwendungen vor deren Implementierung zumindest abschätzen lässt. Diese Arbeiten erbrachten wegen der sich schnell fortentwickelnden Computerarchitekturen bis heute leider keine auf längere Zeit praxistauglichen und quantitativ verwertbaren Lösungen. Fink schlägt 2007 vor diesem Hintergrund in [23] einen Bewertungsansatz vor, der bis heute zumindest eine qualitative Vorabbewertung von Anwendungen ermöglicht. Ausgehend von seinen praktischen Erfahrungen bei der Parallelisierung von komplexen ingenieurtechnischen Aufgabenstellungen sollte seiner Meinung nach eine Anwendung zunächst - wie in Abbildung 3 dargestellt - aus der Sicht des Amdahlschen Gesetzes [2] betrachtet werden. Die einfachste Ablaufstruktur einer parallelen Anwendung mit

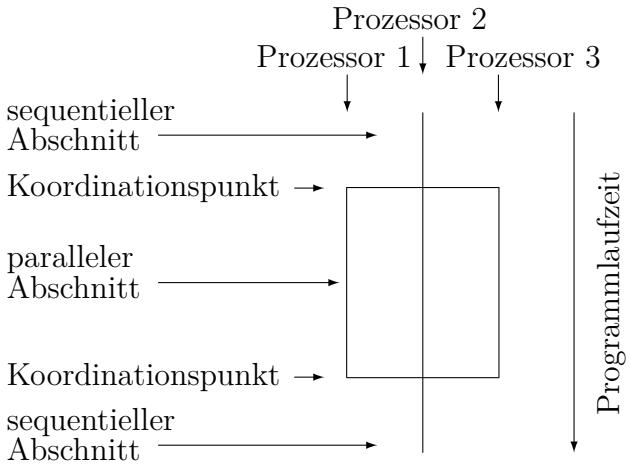


Abbildung 3: Ablaufverhalten einer einfachen parallelen Anwendung nach [23].

nur einem parallelen Abschnitt und zwei *äußeren Koordinationspunkten* bezeichnet Fink als Typ 1. Eine Abfolge mehrerer paralleler und sequentieller Abschnitte wird als Typ 2 bezeichnet. Anwendungen vom Typ 3 besitzen wie Typ 1 nur einen paralle-

len Abschnitt, dieser weist neben den äußeren aber auch *innere Koordinationspunkte* auf. Eine Abfolge mehrerer paralleler Abschnitte vom Typ3 und sequentieller Abschnitte wird als Typ 4 bezeichnet. Abbildung 4 zeigt eine schematische Darstellung der vier Typen. Nach Fink weisen Anwendungen vom Typ 1 eine grobe Granularität

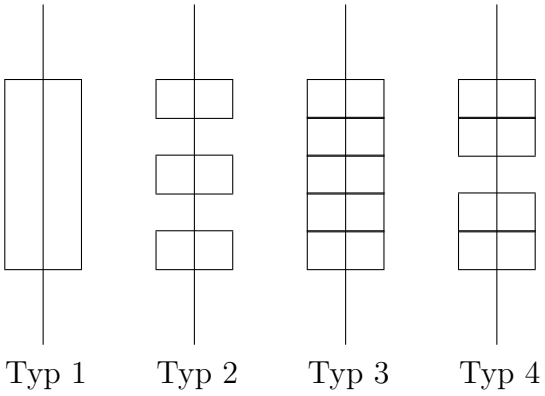


Abbildung 4: Typische Ablaufstrukturen paralleler Anwendungen nach [23].

auf. Alle weiteren Typen werden Schritt für Schritt immer feingranulärer. Über den erfolgreichen Einsatz dieses qualitativen Klassifikationssystems im Bereich simulationstechnischer Problemstellungen wird in [24] berichtet.

Unabhängig von einer konkreten Parallelisierungsebene und einem bestimmten Anwendungstyp ist der Entwurf paralleler Software grundsätzlich komplexer als bei sequentiell abzuarbeitenden Problemen. Foster [29] fasst die vier wesentlichen Schritte eines parallelen Softwareentwurfs unter dem Akronym *PCAM* zusammen. Danach besteht der erste Schritt in der *Partitionierung* des zu lösenden Problems. Durch Partitionierung wird ein Problem in Teilaufgaben zerlegt. Hierdurch wird überhaupt erst eine Parallelverarbeitung möglich. Allerdings resultiert aus der Partitionierung zwangsläufig, dass die Abarbeitung der Teilaufgaben untereinander koordiniert werden muss, was über Synchronisations- und Kommunikationsoperationen zu realisieren ist. Für diesen zweiten Entwurfsschritt steht bei Foster das C, also *Communication*. Danach folgt die *Agglomeration*. Darunter wird die Evaluierung der Partitionierung und der damit verbundenen Koordination vor dem Hintergrund der einzusetzenden Parallelisierungsebene verstanden. Im Ergebnis dieser Evaluation wird die Gesamtmenge der Teilaufgaben unter Leistungs- und Kostengesichtspunkten unter Umständen in Teilaufgaben-Cluster strukturiert. Der letzte Entwurfsschritt ist das *Mapping*. Hier werden die Teilaufgaben beziehungsweise Teilaufgaben-Cluster den Verarbeitungseinheiten eines Rechnersystems zugeordnet.

Nach dem Entwurf einer parallelen Software-Lösung muss für die Implementierung der Einsatz einer konkreten Programmiersprache beziehungsweise -technologie entschieden werden. Diesbezüglich ist das Angebot aus der Sicht von praktischen Anwendern kaum noch überschaubar und auch ordnende Klassifikationen verlieren

durch neue Entwicklungen auf diesem Gebiet relativ schnell an Gültigkeit. Ähnlich wie auf der Hardware-Ebene erweisen sich auch hier ältere, weniger detaillierte Klassifikationsansätze wie beispielsweise in [85, 69, 58, 23] veröffentlicht als hilfreich. Danach muss sich ein Anwender zunächst nur zwischen zwei grundlegenden Implementierungstechniken entscheiden: der *impliziten parallelen Programmierung* oder der *expliziten parallelen Programmierung*. Bei genauerer Betrachtung weisen einige moderne Technologien diesbezüglich aber auch einen hybriden Charakter auf. Bei einer reinen impliziten Technologie erfolgt die Programmierung für den Anwender in gleicher Art und Weise, wie er es von der sequentiellen Problemlösung gewohnt ist. Die Parallelisierung des Quellcodes muss dann idealerweise vollständig durch das Übersetzungssystem erfolgen. Dieser Ansatz funktioniert mit den heutigen Technologien nur auf den unteren Parallelisierungsebenen mit akzeptabler Effizienz. Bei reinen expliziten Technologien muss der Anwender alle Aspekte des PCAM-Entwurfs bei der Programmierung in Eigenregie realisieren. Dieser Ansatz ist gegenüber impliziten Ansätzen ungleich aufwendiger, bietet aber bei ausreichendem Know-How die Möglichkeit, das Potenzial einer konkreten Hardware optimal zu nutzen.

Für das Anwendungsgebiet dieser Arbeit, die diskret-ereignisorientierte Simulation, sind das explizite Programmiermodell und die Parallelisierung auf Programmsowie Prozessebene von besonderer Bedeutung. Deswegen werden im Folgenden zunächst die wichtigsten Middleware-Technologien betrachtet, die in Verbindung mit den klassischen compilerbasierten Programmiersprachen häufig für diesen Anwendungsbereich eingesetzt werden. Abschließend werden als Beispiel für interpreterbasierte Ansätze die Parallelverarbeitungstechniken der SCE Matlab dargestellt.

2.3.1 Middleware für die parallele Verarbeitung

Bereits seit den 1980er Jahren werden Programmiersprachen entwickelt, mit denen sich explizit Nebenläufigkeit beziehungsweise Parallelität beschreiben lässt. Bekannte frühe Beispiele sind Ada und Occam. Daneben setzten sich aber vor allem Middleware-Techniken durch, die Parallelverarbeitung unter Verwendung der etablierten sequentiellen Programmiersprachen ermöglichen. Im Folgenden werden fünf wichtige Technologien in der Reihenfolge ihrer Entwicklung kurz vorgestellt.

RPC

Das Akronym RPC steht für *Remote Procedure Call*. Die Grundidee besteht zunächst in der Übertragung der aus Programmiersprachen bekannten lokalen Prozeduraufrufe auf Computernetzwerke. James E. White beschreibt bereits 1976 ein solches Framework im Kontext des ARPANET [101]. Frühe Remote Procedure Call (RPC)-Implementierungen wie die von Xerox sowie von Apollo und Hewlett Packard sind zunächst nicht für die Parallelverarbeitung geeignet, sondern nur zum Aufbau verteilter Systeme mittels blockierender skalarer RPCs nach dem Client-Server-Interaktionsmodell. Eine weiterentwickelte RPC-Variante, die sehr schnell zum Industriestandard wurde, ist Sun-RPC. Diese Implementierung unterstützt auch nichtblockierende skalare RPCs, wodurch eine einfache Parallelisierung bestimmter Anwendungsklassen (Typ 1 und 2, vgl. Abschn. 2.3) ermöglicht wird. Im

Zusammenhang mit Fortran 90 und Matlab werden in den 1990er Jahren schließlich vektorielle RPCs realisiert [69], die ebenfalls vor allem für die Parallelverarbeitung von großem Interesse sind.

PVM

PVM (*Parallel Virtual Machine*) ist eine Message-Passing-Middleware, deren Entwicklung 1989 am Oak Ridge National Laboratory startete und dann unter weiterer Leitung von Jack Dongarra an der University of Tennessee fortgesetzt wurde [34]. Die Middleware wurde in den frühen 1990er Jahren sehr schnell zu einem Quasi-Standard der expliziten parallelen Programmierung im gesamten HPC-Bereich, also von vernetzten heterogenen Rechnern bis hin zu Supercomputer-Installationen. Im Unterschied zu RPC können mit einem Message-Passing-System wie PVM auch Anwendungsprobleme mit komplexer Struktur (innere Koordinierungspunkte, vgl. Abschn. 2.3) umgesetzt werden.

MPI

MPI (*Message Passing Interface*) ist zunächst eine Initiative von 40 Organisationen aus den USA und Europa, darunter auch die meisten Hersteller von HPC-Systemen. Das Ziel der Initiative ist die Ausarbeitung eines Standards für die Schnittstellengestaltung von Message-Passing-Systemen. Die Erfahrungen mit den bereits existierenden Systemen PVM, P4, PARMACS und anderen fließen in den Standardisierungsprozess ein. Im Unterschied zur bisherigen Praxis sollen mit einem formalen Standard parallele Anwendungen unabhängig von einer konkreten Message-Passing-Implementierung werden, also Quelltextkompatibilität nach dem Vorbild von POSIX bei Betriebssystemen. Dieser Ansatz soll auch die Bereitstellung von hardwarespezifischen Implementierungen wie zum Beispiel für Rechnersysteme mit gemeinsamen Speicher oder hybriden Strukturen erleichtern. Die Version 1.0 des Standards wurde im Mai 1994 veröffentlicht. Seitdem wird der Standard weiterentwickelt. Die letzte Version ist MPI 4.1 [62]. Populäre MPI-Implementierungen sind MPICH, MVAPICH und Open MPI.

HLA

Das Akronym HLA steht für *High Level Architecture* und ist ähnlich wie MPI keine konkrete Middleware, sondern ein Standard zur Implementierung von Middleware zur Unterstützung der Interoperabilität von Simulationen und der Realisierung von verteilten, insbesondere diskret-ereignisorientierten Simulationen, wodurch Parallelverarbeitung eingeschlossen sein kann. Die Initiative zur HLA-Entwicklung startete bereits in den frühen 1990er Jahren innerhalb des Verteidigungsministeriums der USA. 1995 wurde eine erste HLA-Definition vom Defense Modelling and Simulation Office vorgeschlagen [22]. Im weiteren Verlauf wurde 1998 der erste offizielle Standard HLA 1.3 auch für zivile Anwender veröffentlicht. HLA wird bis heute als Institute of Electrical and Electronics Engineers (IEEE)-Standard fortgeschrieben (1516-2000, 1516-2010). Die Veröffentlichung der neusten Version HLA 4 soll 2025 erfolgen (1516-2025). Die als Middleware zu implementierende Komponente wird bei HLA als *Run-Time Infrastructure* (RTI) bezeichnet. Bekannte RTI-

Implementierungen, die IEEE 1516-2010 unterstützen, sind MAK High Performance RTI, Pitch pRTI und Open RTI.

OpenMP

OpenMP (*Open Multi-Processing*) ist ähnlich wie MPI und HLA keine konkrete Implementierung, sondern eine seit 1997 betriebene herstellerübergreifende Initiative zur Unterstützung der parallelen Programmierung auf Plattformen mit gemeinsamen Speicher in Form einer informellen Standardisierung [68]. In gewisser Weise kann OpenMP als Gegenstück der zuvor genannten Technologien RPC, PVM, MPI und HLA gesehen werden. Diese starteten zunächst alle mit dem Fokus auf Hardware-Plattformen ohne gemeinsamen Speicher. Mit dem Erscheinen von hybriden Speicherstrukturen auf der Hardware-Ebene unterstützten diese dann mehr oder weniger auch die Kommunikation über gemeinsamen Speicher. Bei OpenMP lief die Entwicklung entgegengesetzt: zunächst wurde nur gemeinsamer Speicher unterstützt, später erscheinen Varianten, die auch über einer verteilten Speicherstruktur arbeiten können. OpenMP unterscheidet sich aber auch noch in zwei weiteren Aspekten von den zuvor betrachteten Technologien. Während diese eindeutig den Klassen Middleware und explizite parallele Programmierung zugeordnet werden können, ist das bei OpenMP nicht mehr so eindeutig. Bei den meisten OpenMP-konformen Lösungen ist zwar immer noch das Linken von Bibliotheken notwendig, was einen gewissen Middleware-Charakter anzeigt, der zentrale Bestandteil von OpenMP sind aber Compiler-Erweiterungen, die durch sogenannte Pragmas die Parallelisierung bei der Quellcode-Übersetzung steuern. Damit kann in der Regel ein solcher Quellcode auch immer noch durch einen Compiler übersetzt werden, der über keine OpenMP-Fähigkeit verfügt. In einem solchen Fall werden die OpenMP-Pragmas vom Compiler ignoriert und der Quellcode wird wie gewöhnlich in einen nur sequentiell ausführbaren Code übersetzt. Vor diesem Hintergrund ordnet beispielsweise Märtin [58] die OpenMP-Technologie nicht der expliziten parallelen Programmierung zu, sondern bezeichnet sie als *semi-implizit*.

2.3.2 Parallele Verarbeitung in der SCE Matlab

Wie im vorangegangenen Abschnitt dargestellt, startet die Entwicklung der auch heute noch dominanten Middleware-Technologien in der Mehrzahl in den frühen 1990er Jahren. Bis in diese Zeit galten interpreterbasierte Technologien, wie sie für SCEs typisch sind, im Bereich der Informatik zwar als komfortabel für Anwender, aber als völlig ungeeignet für das Hochleistungsrechnen. Cleve Moler führte bereits in den 1980er Jahren umfängliche Experimente aus, um die Leistungsfähigkeit der von ihm entwickelten SCE Matlab durch Parallelverarbeitung zu erhöhen. 1995 veröffentlicht er in [64] die ernüchternden Ergebnisse seiner Arbeiten und kündigte an, dass seine Firma The MathWorks sicher für mindestens eine Dekade nicht mehr in Parallelverarbeitung investieren würde. In der Rückschau ist das negative Ergebnis der frühen Experimente von Moler leicht erklärbar. Er versuchte nach Ungerer [94] Parallelverarbeitung in Matlab auf einer niedrigen Granularitätsebene einzuführen, was beim damaligen Entwicklungsstand der Computerarchitekturen noch zum

Scheitern verurteilt war. Pawletta ging ab 1992 an der Universität Rostock einen völlig anderen Weg: Er führte mit seinem *Multi-SCE-Ansatz* Parallelverarbeitung auf den viel höheren Programm- und Prozessebenen ein und konnte damit zumindest für grobgranuläre Anwendungen unter Verwendung gewöhnlicher Matlab-Versionen hohe Speedups erzielen [69].

Infolge wurde der Multi-SCE-Ansatz vielfach aufgegriffen und mit Systemen der gesamtem SCE-Klasse, wie beispielsweise Octave, Scilab, IDL und anderen, erfolgreich umgesetzt. Fink analysiert in [23] den Stand der Parallelverarbeitung in SCEs bis 2007 sehr umfänglich. Dabei bezieht er neben dem Multi-SCE-basierten Ansätzen auch weitere Techniken mit ein. Unter den analysierten Technologien findet sich bei Fink auch die *Distributed Computing Toolbox* (DCT) von The Mathworks aus dem Jahr 2006. Zu diesem Zeitpunkt beschränkte sich The MathWorks noch im Wesentlichen auf die Umsetzung des Multi-SCE-Ansatzes auf Basis von MPI. Der Autor dieser Arbeit führte 2019 eine erneute Studie zum Entwicklungsstand der SCEs Octave, Matlab und Scilab bezüglich Parallelverarbeitung durch [47]. Im Zusammenhang mit der SCE Matlab wurde dabei festgestellt, dass die Distributed Computing Toolbox (DCT) inzwischen in *Parallel Computing Toolbox* (PCT) umbenannt wurde. Moderne PCT-Versionen stellen eine ganze Sammlung verschiedener Ansätze zur Parallelverarbeitung bereit [90]. Im Folgenden werden die im Kontext dieser Arbeit wichtigsten Techniken kurz vorgestellt:

- `parfor`: Parallelisierung von for-Schleifen,
- `spmd`: einen parallelen Abschnitt eröffnen,
- `parsim`: paralleles Ausführen von Simulink-Modellen,
- `batch`: Ausführen von sequentiellen und parallelen Jobs auf Matlab-Workern,
- `CUDA`: Berechnungen auf einer Nvidia-GPU ausführen.

Die Methode `parfor` entspricht funktional OpenMP (vgl. Abschn. 2.3.1). Sie unterstützt eine semi-implizite parallele Programmierung auf der Threadebene. Die Parallelisierungsaufgaben: Zerlegung, Kommunikation, Instanziierung und Synchronisation werden automatisch durch die Laufzeitumgebung realisiert.

Mit der Methode `spmd` (*single program multiple data*) kann aus einem sequentiellen Verarbeitungsabschnitt heraus ein paralleler Abschnitt eröffnet werden. Diese Methode der PCT kann der expliziten parallelen Programmierung zugeordnet werden, weil die Zerlegung, Kommunikation und Synchronisation vom Programmierer zu spezifizieren ist.

Die Methode `parsim` entspricht funktional einem blockierenden vektoriellen RPC (vgl. Abschn. 2.3.1). In Matlab sind, wie in anderen SCEs auch, umfangreiche Funktionalitäten über sogenannte Subsysteme realisiert. Solche Subsysteme stellen eigenständige Anwendungen auf der Programmebene dar. Ein sehr populäres Subsystem im Matlab-Kontext ist Simulink, das die Erstellung und Ausführung von Simulationsmodellen ermöglicht. Mittels `parsim` kann aus einem sequentiellen Abschnitt heraus, eine Vielzahl von Simulink-Ausführungen angestoßen werden. Damit sind für

simulationstechnische Anwendungen insbesondere parallele Parameterstudien sehr einfach realisierbar.

Die Methode `batch` ist als Stapelverarbeitungssystem gedacht. Damit können Jobs erzeugt werden, die entweder sequentiell oder parallel auf einem Cluster ausgeführt werden.

Mit `CUDA` unterstützt die PCT auch die Parallelverarbeitung auf der SIMD-Ebene unter Verwendung einer Nvidia-GPU (vgl. Abschn. 2.3.1). Ein breiteres Spektrum für das GPU-Computing mittels OpenCL wird in Matlab über die *OpenCL-Toolbox* [40] bereitgestellt.

In Vorbereitung dieser Arbeit wurden unter Verwendung der in SNE [6] veröffentlichten simulationstechnischen Benchmarks Performance-Untersuchungen unter Einschluss der PCT durchgeführt [43, 42]. Im Vergleich zu den ebenfalls untersuchten Lösungen auf Basis der Programmiersprache C in Kombination mit MPI und GSL (*GNU Scientific Library*) lieferte die PCT zum Zeitpunkt der Performace-Studie relativ schlechte Ergebnisse. Dennoch wird im Kapitel 6 dieser Arbeit auf die PCT abgestellt. Für diese Entscheidung sprechen zwei Aspekte: einerseits die überragende Relevanz von Matlab als Quasi-Standard im ingeniertechnischen Bereich und andererseits die Annahme, dass The Mathworks kontinuierlich an der Optimierung ihrer Implementierungen arbeiten wird.

2.4 Leistungsbewertung paralleler Systeme

Die Leistungsbewertung paralleler Systeme kann auf der Hardware- und Softwareebene erfolgen. Für die Hardware verwenden Hersteller zur Spezifikation der Rechenleistung Kenngrößen wie *Instructions per Cycle*, *Instructions per Second*, *Floating Point Operations per Second*. Die Kommunikationsleistung wird üblicherweise in *Bits per Second* angegeben.

Parallele Rechnersysteme stellen unterschiedliche Verhältnisse von Rechen- versus Kommunikationsleistung zur Verfügung. Demgegenüber steht der Bedarf von Anwendungen, der qualitativ von *fein-* über *mittel-* bis *grobkörnig* klassifiziert werden kann. Feinkörnige Anwendungen haben in Relation zum Rechenbedarf einen hohen Kommunikationsbedarf. Grobkörnige Anwendungen sind durch ein inverses Bedarfsverhältnis charakterisiert.

Als Maß für die durch Parallelverarbeitung erzielte Beschleunigung einer Anwendung dient der *Speedup*. Er ist dimensionslos und ergibt sich als Quotient aus der Laufzeit der Anwendung auf nur einer Ausführungseinheit T_1 und der Laufzeit T_p unter Nutzung von p Ausführungseinheiten:

$$S_p = \frac{T_1}{T_p}. \tag{2.1}$$

Gewöhnlich wird zur Messung von T_1 eine sequentielle Anwendungsimplementierung benutzt, die bereits vor der Parallelisierung im Einsatz war. Existiert eine solche

nicht, kann T_1 unter Verwendung der parallelen Implementierung bestimmt werden, die dann auf nur einer Ausführungseinheit abläuft und ausgemessen wird. Diese Art der Speedup-Bestimmung wird unter anderem bei der Evaluierung alternativer paralleler Algorithmen und Implementierungen eingesetzt. In diesem Kontext wird dann meist von einem *algorithmischen* Speedup gesprochen.

Wenn $S_p = p$ ist, spricht man von einem *idealen* Speedup. Wegen des Amdahlschen Gesetzes lässt sich ein solcher Speedup in der Regel aber nicht erreichen, weil mehr oder weniger große Teile einer realen Anwendung nicht parallelisiert werden können. Damit ergibt sich eine Laufzeit von:

$$T = t_s + t_p, \tag{2.2}$$

wobei t_s für die Laufzeit des nicht parallelisierbaren Anteils und t_p für die Laufzeit des parallelen Anteils steht. Gemäß Gleichung 2.1 ergibt sich dann:

$$S_p = \frac{T}{t_s + \frac{t_p}{p}}. \tag{2.3}$$

Aus Gleichung 2.3 lässt sich erkennen, dass sich die Laufzeit einer parallelisierten Anwendung auch bei einer sehr großen Zahl von Ausführungseinheiten p nicht beliebig verkleinern lässt und der Speedup S_p damit asymptotisch auf einen maximal erreichbaren Wert zuläuft, der durch den Laufzeitaufwand t_s des nicht parallelisierbaren Anwendungsanteils bestimmt wird. In der Praxis kann man häufig beobachten, dass nach Erreichen des nach Gleichung 2.3 erwarteten maximalen Speedups, dieser bei weiterer Erhöhung von p wieder absinkt. Hierfür ist die Eigenschaft mancher Anwendungen verantwortlich, bei denen t_s nicht wie in Gleichung 2.2 angenommen konstant ist, sondern bei steigender Zahl von Ausführungseinheiten p ebenfalls ansteigt. Dieser häufig als steigender Synchronisationsaufwand bezeichnete Effekt kann als p-abhängige Größe $t_{O(p)}$ in Gleichung 2.2 als zusätzlicher Summand eingeführt werden. Dann ergibt sich:

$$S_p = \frac{T}{t_s + t_{O(p)} + \frac{t_p}{p}}. \tag{2.4}$$

Mit Gleichung 2.4 lässt sich das Absinken von S_p nach Erreichen des Maximalwertes bei weiterer Erhöhung von p erklären.

Neben dem Amdahlschen Gesetz spielt für die Planung und Bewertung von Parallelisierungen für manche Anwendungsbereiche auch das sogenannte *Gustafson Gesetz* eine Rolle. Hier steht anders als bei Amdahl nicht die Lösung eines Problems konstanter Komplexität in kürzerer Zeit im Fokus, sondern die Lösung eines Problems in einem konstanten Zeitfenster, aber mit erhöhter Genauigkeit. In der sekundären Literatur wird auf Basis von Gustafson vielfach das zum Speedup alternative Bewertungsmaß *Scaleup* SC_p eingeführt.

2.5 Zusammenfassung

In diesem Kapitel wurde in die wesentlichen Grundlagen der Parallelverarbeitung in einem Maße eingeführt, wie es für das Verständnis der weiteren Arbeit erforderlich ist.

Zunächst wurde auf die schwierige Begriffsabgrenzung zwischen paralleler und verteilter Verarbeitung eingegangen. Diese Problematik ist insbesondere im Kontext simulationstechnischer Anwendungen von Bedeutung.

Danach wurde die Entwicklung der Hardwareebene, also der Computerarchitekturen, in ihren wesentlichen Meilensteinen dargestellt. Die Betrachtungen starten mit den theoretischen Arbeiten, die zur auch heute noch wohlbekannten Von-Neumann-Architektur führen und zunächst auf eine sequentielle Verarbeitung abzielen. Im Folgenden wird aufgezeigt, dass die realen Computersysteme bereits in den 1960er Jahren über die Von-Neumann-Architektur erheblich hinausgehen, um eine Leistungssteigerung durch verschiedene Parallelverarbeitungstechniken zu ermöglichen. Die in diesem Zusammenhang von Flynn erarbeitete Klassifikation wurde vorgestellt. Auch wenn sich heutige Computerarchitekturen aufgrund der technologischen Vielfalt nicht mehr eindeutig in diese Klassifikation einordnen lassen, so bietet sie doch immer noch gute Anhaltspunkte zur Einordnung von Teilaspekten der heute oft hybriden Strukturen. Hinsichtlich der diskutierten Hardwaretechnologien sind für diese Arbeit primär HPC-Cluster mit Multicore- und Multiprozessorarchitekturen von Bedeutung. Die ebenfalls kurz betrachteten GPU-Coprozessoren spielen für die hier vorliegende Arbeit eine eher untergeordnete Rolle.

Im Softwareteil wurden die verschiedenen Ebenen zur Realisierung von Parallelverarbeitung und die Aufgaben bei der Entwicklung paralleler Programme kurz vorgestellt. Für das Anwendungsgebiet dieser Arbeit, die Beschleunigung komplexer und hochkomplexer Simulationsstudien, sind insbesondere die Programm- und die Prozessebene von Bedeutung. Dementsprechend wurden verschiedene Middleware-Lösungen für die parallele Verarbeitung auf diesen Ebenen betrachtet. Anschließend wurden die in der SCE Matlab unterstützten Konzepte zur Parallelverarbeitung analysiert, wobei nur die proprietäre *Parallel Computing Toolbox* betrachtet wurde.

Abschließend wurde ein kurzer Überblick zur Leistungsbewertung von parallelen Systemen auf der Hardware- und auf der Softwareebene gegeben.

3 Methoden zur Beschleunigung von DES-Studien

DES-Studien bestehen aus einer Reihe von Simulationsexperimenten [103, 82]. Für die Durchführung von DES-Studien wurden verschiedene Vorgehensmodelle [78, 95], Workflows [81] und unterstützende Werkzeuge [54] entwickelt, die alle auf dem Grundansatz der strikten Trennung von Modell und Experiment basieren [105, 106].

DES-Studien können unterschiedliche Ziele verfolgen. Peng [73] unterscheidet zur Einordnung der Ziele drei Kategorien: den Untersuchungsgegenstand (Modell oder Simulationsalgorithmen), die Phase im M&S-Prozess und die Experimentziele (Modellkalibrierung, Sensitivitätsanalyse, Optimierung etc.).

Leye [53] analysiert die Struktur von Simulationsexperimenten in DES-Studien und identifiziert sechs charakteristische Aufgaben von der Konfiguration eines Experiments bis zur Auswertung der Ergebnisse. Ausgehend von diesen Aufgaben entwickelt Schmidt [82] ein Konzept zur Experimentsteuerung und teilt Simulationsexperimente hinsichtlich ihrer Struktur in drei Klassen ein: *einfache, komplexe* und *hochkomplexe Experimente*. Schmidts Klassen basieren auf der Strukturierung nach Zeigler [105, 106] (*Model Under Study (MUS), Experimental Frame (EF), Experiment Control (EC)*) und den Definitionen nach Breitenecker [5] (*model, method, experiment*).

Ein simulationsbasiertes Experiment (SBE) besteht immer aus einer *Simulationsmethode (SM)*, die die Durchführung eines oder mehrerer *Simulationsläufe* steuert, und mindestens einer *Experimentmethode (EM)*. Bei *einfachen Experimenten* ist die EM eine Ablaufsteuerung. Bei *komplexen Experimenten* führt eine EM unter Verwendung der SM automatisiert Simulationsläufe aus, analysiert die Ergebnisse und generiert gegebenenfalls neue Simulationsläufe, wie zum Beispiel bei einer Parameteroptimierung. *Hochkomplexe Experimente* variieren nicht nur Einflussgrößen des Modells, sondern variieren auch Modellstrukturen, wie zum Beispiel bei der kombinierten Parameter- und Strukturoptimierung [37].

In diesem Kapitel werden verschiedene Ansätze zur Beschleunigung von DES-Studien diskutiert. Dabei geht es nicht um den Einsatz spezifischer Hard- oder Softwaretechnologien, sondern um Parallelverarbeitung auf den im Abschnitt 2.3 besprochenen Softwareebenen. Analog zu den Software-Vergleichen zur Beschleunigung von Simulationsstudien durch Parallelverarbeitung von Breitenecker et. al [7, 6] wird in dieser Arbeit zwischen Ansätzen zur Parallelisierung auf Modell- beziehungsweise Experimentebene unterschieden.

Die Parallelisierung auf Modellebene betrifft die verteilte Ausführung eines DES-Modells und die parallele Ereignisausführung. Bezogen auf die im Abschnitt 2.3 betrachteten Softwareebenen erfolgt die Parallelisierung bei der verteilten Ausführung eines DES-Modells auf der *Prozessebene*, während sie bei der parallelen Ereignisausführung auf der *Prozess-* oder *Threadebene* erfolgt. Für die verteilte Ausführung eines DES-Modells wurde bereits im Abschnitt 2.3.1 auf HLA als De-facto-Standard und Middleware für die verteilte Simulation hingewiesen. Die Ausführungen in diesem Kapitel konzentrieren sich auf die grundlegenden Konzepte nach [33, 88, 61], die unter anderem die Basis der HLA-Spezifikation bilden.

Für die Parallelisierung auf Experimentebene wird die parallele Ausführung von Simulationsläufen betrachtet. Diese ist hinsichtlich der Beschleunigung von DES-Studien für alle Experimentklassen nach Schmidt [82] von besonderer Relevanz.

3.1 Parallelisierung auf Modellebene

In diesem Abschnitt werden Methoden zur Parallelisierung auf Modellebene betrachtet. Dabei wird nach Mehl [61] zwischen der verteilten Ausführung eines DES-Modells und der parallelen Ausführung von Ereignissen unterschieden.

3.1.1 Verteiltes DES-Modell

Das DES-Modell wird in Teilmodelle zerlegt. Diese sollten einen logischen Zusammenhang besitzen und möglichst unabhängig voneinander sein. Jedem Teilmodell wird eine eigene Ausführungseinheit zugeordnet, die als Simulator bezeichnet wird. Ein Teilmodell mit zugeordnetem Simulator bildet nach Mehl [61] einen logischen Prozess (LP). Während der Ausführung interagieren die LPs miteinander, wobei eine Koordination der LPs erfolgen muss. Die Koordinationsprozesse und die einzelnen Simulatoren bilden den ereignisdiskreten Simulator. Nach Mehl [61] sollten folgende Anforderungen möglichst erfüllt sein, um eine Beschleunigung der Simulationsausführung zu erreichen: (i) Partitionierung des Modells in unabhängige Teilmodelle, (ii) Mapping der LPs auf Prozessoren und Scheduling, (iii) Synchronisation der Ereignisausführung. Diese Anforderungen ähneln den in Abschnitt 2.3 besprochenen Schritten zur Parallelisierung von Anwendungen nach Foster [29] beziehungsweise Skillicorn [85].

Nach Mehl [61] hat die Partitionierung des Modells in Teilmodelle einen entscheidenden Einfluss auf die Effizienz der Parallelisierung. Ziel der Partitionierung sollte sein, Teilmodelle zu finden, die möglichst wenig Interaktionen untereinander aufweisen. Nach Aussage von Mehl ist dieser Vorgang nur schwer automatisierbar und bleibt oftmals Aufgabe des Modellierers.

Beim Mapping werden die LPs den Prozessoren zugeordnet. Die LPs können unterschiedlich viel Rechenaufwand in Anspruch nehmen und es besteht unterschiedlich viel Kommunikations- beziehungsweise Synchronisationsbedarf. Bei Modellen mit

relativ hohem Kommunikations- und Synchronisationsbedarf der LPs kann sich eine Zerlegung in eher wenige Teilmodelle positiv auf die Laufzeit auswirken. Nach Mehl [61] soll die Anzahl der Teilmodelle von realistischen Simulationen viel geringer sein als die Anzahl der zur Verfügung stehenden Prozessoren. Ob diese Aussage heute noch zutrifft, ist fraglich, aber nicht Gegenstand dieser Arbeit.

Die Synchronisation der LPs ist ein entscheidender Aspekt der verteilten Simulation. In der Literatur werden diesbezüglich drei Ansätze unterschieden, die im Folgenden überblicksartig dargestellt werden.

Konservative Simulationsverfahren

Bei der verteilten Simulation soll die Laufzeit gegenüber einer sequentiellen Simulation dadurch verkürzt werden, dass die LPs zeitlich simultan ausgeführt werden. Durch die parallele Abarbeitung können aufgrund von Wechselwirkungen zwischen den LPs Probleme hinsichtlich der Reihenfolge der Ereignisse auftreten. Konservative Verfahren zielen darauf ab, die kausale Ordnung der Ereignisse unter allen Umständen einzuhalten. Um dies zu gewährleisten, müssen die LPs Informationen über die Zeitstempel ihrer Ereignisse austauschen. Ansätze hierzu wurden von Bryant [8] sowie von Chandy und Misra [16, 63] vorgeschlagen.

Basierend auf den Ideen zu Zeitstempeln wurde der Begriff *Garantie* eingeführt. Damit ein konservatives Verfahren LPs parallel ausführen kann, ohne die Kausalordnung der Ereignisse zu gefährden, müssen Garantien zwischen den LPs ausgetauscht werden. Mehl [61] definiert eine Garantie wie folgt:

„Eine Garantie G von LP_i an LP_j ist die Zusicherung von LP_i an LP_j, daß LP_j während der restlichen Simulation keine Ereignisse mehr von LP_i erhalten wird, deren Zeitstempel kleiner als G sind."

Die Garantien können nach Mehl [61] aus lokalem, globalem oder externem Wissen bestimmt werden und der Austausch von Garantien kann auf Basis impliziter, expliziter oder hybrider Austauschschemata erfolgen. Wesentlich ist, dass *Deadlocks* vermieden werden und möglichst wenig Kommunikationsaufwand für den Austausch von Garantien entsteht. Die Effizienz eines konservativen Verfahrens wird somit maßgeblich durch die Qualität der Garantien bestimmt.

Mit den im Kapitel 4 dieser Arbeit betrachteten DEVS-Formalismen wurden verschiedene konservative Simulatoren realisiert. Beispielhaft sei hier auf Nutaro [107, S. 356-358] verwiesen, der einen konservativen Simulator auf Basis des PDEVS-Formalismus vorstellt.

Optimistische Simulationsverfahren

Bei optimistischen Simulationsverfahren werden mögliche Reihenfolgeverletzungen bei der Ereignisausführung nicht vorab verhindert. Treten solche tatsächlich auf, werden sie aber erkannt und anschließend korrigiert. Der Ursprung optimistischer Verfahren geht auf Jefferson [48] zurück und steht im Zusammenhang mit der Entwicklung des *Time Warp Operating System* [49, 31].

In einer optimistischen Simulation werden die LPs im Time-Warp-Verfahren betrieben. Das bedeutet, dass ein LP immer das Ereignis mit dem kleinsten Zeitstempel in seiner Ereignisliste ausführt und somit in der Simulationszeit t voranschreitet. Aufgrund von Wechselwirkungen mit anderen LPs können jedoch noch Ereignisse mit Zeitstempeln $t' < t$ eintreffen. Um die kausale Ordnung wiederherzustellen, muss der LP in diesem Fall einen *Rollback* durchführen, also seinen Zustand auf einen früheren Zustand vor t' zurücksetzen. Sollte der LP zuvor zwischen t' und t Ereignisse erzeugt haben, die andere LPs betreffen, so muss er diese durch sogenannte *Anti-Ereignisse* [61] korrigieren. Anti-Ereignisse können in anderen LPs weitere Anti-Ereignisse auslösen, wodurch eine *Rollback-Kaskade* ausgelöst werden kann.

Besonders aufwendig ist die Speicherung der Zustände, um bei Bedarf einen Rollback durchführen zu können. Bis in die 1990er Jahre konnte es bei optimistischen Simulationen aufgrund von Speichermangel schnell zu Blockaden kommen. Um dieses Problem zu beheben, wurden nach [61] einerseits ressourcenschonendere Algorithmen entwickelt, aber auch spezielle Hardware (Rollback-Chip) zur Speicherung und Markierung von Zuständen sowie zur Rollback-Durchführung.

Der optimistische Ansatz kann auch für Simulationen auf Basis des DEVS-Formalismus verwendet werden. Nutaro beschreibt beispielsweise in [107, S. 358-364] einen optimistischen PDEVS-Simulator.

Hybride Simulationsverfahren

Die Klasse der hybriden Verfahren umfasst eine Kombination aus konservativen und optimistischen Verfahren. Mehl [61] unterteilt die hybriden Verfahren weiter in *vertikal-hybride* und *horizontal-hybride Verfahren*. Bei horizontal-hybriden Verfahren werden konservative und optimistische Verfahren nebeneinander betrieben. Dabei kann ein Teil der LPs eine konservative und ein anderer Teil eine optimistische Strategie verfolgen. Bei vertikal-hybriden Verfahren verfolgen alle LPs die gleiche Strategie, die sowohl konservative als auch optimistische Ansätze enthält.

Ziel dieser Verfahren ist es, die Vorteile optimistischer und konservativer Verfahren zu kombinieren. Mehl [61] führt als vertikal-hybrides Verfahren die *spekulative Simulation* ein. Bei diesem Verfahren wird die ungenutzte Rechenzeit eines LPs, die durch das Warten auf Garantien entsteht, genutzt, um die Simulation auf einer Kopie fortzusetzen. Dabei wird spekuliert, dass keine ungeplanten Ereignisse von anderen LPs den Zustand des LPs beeinflussen. Tritt ein solches ungeplantes Ereignis doch auf, werden die Ergebnisse der spekulativen Simulation verworfen.

Hybride Verfahrensentwicklungen auf Basis des DEVS-Formalismus sind dem Autor nicht bekannt.

3.1.2 Parallele Ereignisausführung

Ein weiterer Ansatz zur Beschleunigung von DES-Simulationen ist die parallele Verarbeitung von gleichzeitigen Ereignissen. Nach Mehl [61] ist die parallele Abarbeitung von gleichzeitigen Ereignissen nicht erfolgversprechend, da diese sehr selten

auftreten. Dies ist beispielsweise bei Simulationen der Fall, die mit einem 2-Phasen- oder 3-Phasen-Scheduler nach [99] durchgeführt werden.

Bei Simulationsansätzen auf Basis des DEVS-Formalismus, die im Kapitel 4 dieser Arbeit betrachtet werden, treten gleichzeitige Ereignisse dagegen häufiger auf, insbesondere bei Modellen mit Komponenten mit Mealy-Verhalten. Je nach Ausprägung des DEVS-Formalismus werden Ereignisse in verschiedene Typen unterteilt. So gibt es in Classic DEVS interne und externe Ereignisse. Die Abarbeitung der internen Ereignisse erfolgt sequentiell und wird durch eine spezielle Funktion gesteuert. Dabei kann ein internes Ereignis mehrere externe Ereignisse auslösen. Die externen Ereignisse können prinzipiell parallel abgearbeitet werden. Dies wird nach Christensen [20] als *External Event Parallelism* bezeichnet.

Ende der 1980er und Anfang der 1990er Jahre wurde verstärkt an parallelen DEVS-Formalismen geforscht. Dabei ging es nicht nur um konservative und optimistische Simulationsverfahren wie bei Christensen [20], sondern auch um *Internal Event Parallelism*. Ein erster Ansatz hierzu stammt von Wang [98, 97] mit dem *Extended Discrete Event System Specification* (E-DEVS)-Formalismus. Nach diesem Formalismus werden interne Ereignisse, die den gleichen Ereigniszeitpunkt haben, parallel ausgeführt. Dieser Ansatz wurde von Chow [19, 17, 18] zum PDEVS-Formalismus weiterentwickelt (vgl. Abschn. 4.2).

Um die Performance von DEVS-basierten Simulatoren mit paralleler Ereignisausführung zu testen, entwickelte Glinsky [35] den *DEVStone-Benchmark*. Ein DEVS-Simulator hat eine hierarchische Struktur und besteht aus Koordinatoren und Simulatoren (vgl. Kapitel 4). Der DEVStone-Benchmark testet sowohl die Leistung der Koordinatoren als auch der Simulatoren. Für den Test der Simulatoren wird der *Dhrystone-Benchmark* nach Weicker [100] verwendet. Wainer [96] testete 2011 auf dieser Basis verschiedene DEVS-Implementierungen. Eine weitergehende Studie mit einem modifizierten Benchmark wurde 2016 von Risco-Martín [79] durchgeführt.

3.2 Parallelisierung auf Experimentebene

Durch die im Abschnitt 3.1 betrachtete Parallelisierung auf Modellebene wird die Beschleunigung eines einzelnen Simulationslaufs angestrebt. DES-Studien bestehen aus einer Vielzahl von Simulationsläufen und können damit in Summe von dieser Art der Parallelisierung profitieren. Darüber hinaus können je nach Struktur der Studie unter Umständen aber auch mehrere Simulationsläufe gleichzeitig ausgeführt werden. Hierbei handelt es sich dann um eine Parallelisierung auf Experimentebene.

Nachfolgend wird dieses Vorgehen zunächst in seiner allgemeinen Form beschrieben und dann die spezielle Methode der parallelen Batch-Simulationsläufe vorgestellt.

3.2.1 Parallele Simulationsläufe

Bei der parallelen Ausführung von Simulationsläufen werden diese auf die vorhandenen Prozessoren verteilt und möglichst unabhängig voneinader ausgeführt (Abb. 5). Wenn die Simulationsläufe den gleichen Rechenaufwand erfordern, können sie gleichmäßig auf die Prozessoren verteilt werden. Eine solche Situation tritt typischerweise bei Monte-Carlo-Simulationen auf, wie beispielsweise im Simulations-Benchmark nach [6], welche vom Autor dieser Arbeit in [43] untersucht wurde. Die technische Umsetzung ist einfach und der Ansatz skaliert sehr gut, da in der Regel keine Kommunikation zwischen den parallelen Instanzen erforderlich ist.

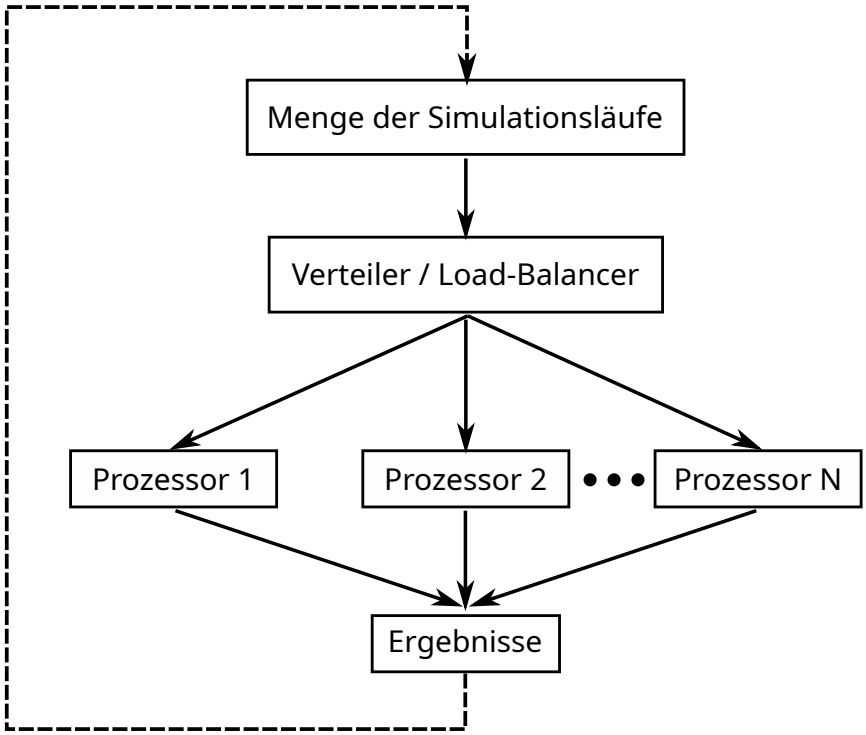


Abbildung 5: Prinzip der Ausführung paralleler Simulationsläufe.

Für Simulationsläufe mit sehr verschiedenen Rechenbedarfen bietet sich der Einsatz eines Load-Balancers an. Dieser verteilt die Simulationsläufe so, dass die Prozessoren möglichst maximal ausgelastet werden.

Generell ist zu beachten, dass die parallele Ausführung von Simulationen nur dann einen Geschwindigkeitsvorteil erbringt, wenn die Einzelläufe einen erheblichen Rechenbedarf aufweisen, der den Aufwand für deren Verteilung weit übersteigt. Dies gilt insbesondere für komplexe und hochkomplexe Experimente.

3.2.2 Parallele Batch-Simulationsläufe

Der hier als Methode der *parallelen Batch-Simulationsläufe* bezeichnete Ansatz hat zum Ziel, durch Bündelung von Aufgaben, in diesem Fall Simulationsläufen, die Auslastung einer nebenläufigen Ausführungseinheit zu erhöhen oder die Auslastung verschiedener paralleler Ausführungseinheiten anzugleichen. Die gebündelten Simulationsläufe werden gemäß Abbildung 5 auf die Prozessoren verteilt und ausgeführt. Neben einer verbesserten Auslastung der Hardware-Ressourcen kann durch Bündelung von Simulationsläufen häufig auch der Kommunikationsaufwand reduziert werden.

Dieses Verfahren wurde von Schmidt [82] erfolgreich zur Beschleunigung von DES-Studien eingesetzt. In seinem Fall umfasste die DES-Studie 2592 Simulationsläufe, die in Batches zu je 32 Simulationsläufen gebündelt ausgeführt wurden.

3.3 Zusammenfassung

Ausgehend von Arbeiten zum Aufbau und zur Durchführung von DES-Studien sowie zur Strukturierung von Simulationsexperimenten wurden in diesem Kapitel grundlegende Methoden zur Beschleunigung von DES-Studien durch Parallelverarbeitung betrachtet. Entsprechend der grundsätzlichen Trennung von Modell und Experiment wurden zunächst Methoden zur Parallelisierung auf Modellebene und anschließend zur Parallelisierung auf Experimentebene diskutiert.

DES-Studien bestehen aus einer Reihe von Simulationsexperimenten. Diese können sich in ihrer Komplexität unterscheiden und werden in der Literatur in verschiedene Klassen eingeteilt. Die Parallelisierung auf Experimentebene durch die parallele Ausführung von Simulationsläufen ist auf alle Experimentklassen gleichermaßen anwendbar und relativ einfach zu realisieren. Die Granularität, also das Verhältnis von Rechenbedarf zu Kommunikations-/Synchronisationsbedarf, ist deutlich günstiger als bei der Parallelisierung auf Modellebene. Unter dieser Voraussetzung sind auf heutigen MIMD-Plattformen (vgl. Abschn. 2.2) signifikante Beschleunigungen erzielbar. Mit dem Ansatz der *parallelen Batch-Simulationsläufe* wurde eine Methode vorgestellt, mit der durch Bündelung von Simulationsläufen die Granularität noch weiter erhöht werden kann.

In den folgenden Kapiteln wird zunächst in den DEVS-Formalismus allgemein und in die neue Ausprägung NSA-DEVS eingeführt. Anschießend werden die Ergebnisse einer konkreten DES-Studie durch Parallelisierung auf der Experimentebene unter Verwendung von NSA-DEVS präsentiert.

4 DEVS-Formalismen

Der Discrete Event System Specification (DEVS)-Formalismus wird seit der ersten Veröffentlichung von Zeigler im Jahr 1976 [105] stetig weiterentwickelt, so dass heute verschiedene DEVS-Formalismen existieren. Der Ausgangsformalismus, heute als Classic DEVS bezeichnet, sowie die für diese Arbeit wichtigen Weiterentwicklungen Parallel Discrete Event System Specification (PDEVS) und Revised Parallel Discrete Event System Specification (RPDEVS) werden in diesem Kapitel betrachtet.

Die kleinste Einheit bei DEVS wird als atomares DEVS oder atomares Modell bezeichnet, welches die Modelldynamik beschreibt. Ein atomares Modell kann als eine Erweiterung des Moore-Automaten betrachtet werden. Die Erweiterung bezieht sich auf die explizite Abhängigkeit von der Zeit, die bei Moore [65], aber auch bei Mealy [60], keine Auswirkung auf das Verhalten hat. Jedem atomaren Modell wird auf der Simulationsseite ein Simulator zugeordnet.

DEVS-Modelle definieren eine Ein-/Ausgangsschnittstelle und können durch die Spezifikation von Kopplungsrelationen zu gekoppelten DEVS aggregiert werden, die als gekoppelte Modelle oder Netzwerke bezeichnet werden. Es können auch Kopplungsrelationen zwischen atomaren und gekoppelten Modellen oder auch zwischen gekoppelten Modellen beschrieben werden. Dadurch ist es möglich, modular-hierarchische Systeme abzubilden.

Ein weiterer wichtiger Teil aller DEVS-Formalismen ist der dazugehörige Simulationsalgorithmus. Dieser beschreibt die Abarbeitung eines DEVS-Modells. Der Algorithmus wird in der Regel durch drei Module beschrieben: Root-Koordinator, Koordinator und Simulator. Der Simulator beschreibt die Abarbeitung eines atomaren Modells, der Koordinator beschreibt die Abarbeitung beziehungsweise die Koordinierung eines gekoppelten Modells und der Root-Koordinator übernimmt die Aufgabe der obersten Instanz und initialisiert, startet und überwacht die Simulation. Das heißt, die modular-hierarchische Modellstruktur wird zur Abarbeitung mit einer modular-hierarchischen Simulatorstruktur gemapped, indem jedem DEVS-Modell eine eigene Abarbeitungseinheit in Form eines Simulators oder Koordinators zugewiesen wird. Dieses Konzept ist in Abbildung 6 dargestellt und bezieht sich auf alle in dieser Arbeit vorgestellten Formalismen. Auf der obersten Ebene befindet sich der Root-Koordinator. Diesem ist als einzige Instanz kein Modell zugeordnet. Dem Root-Koordinator folgt immer genau ein Koordinator oder ein Simulator. Letzteres ist ein Sonderfall, wenn ein DEVS-Modell aus nur einem atomaren Modell besteht. Danach folgen beliebig viele weitere Koordinatoren und Simulatoren. Die Struktur ist ein Baum, wobei die Wurzel durch den Root-Koordinator, die Verzweigungen durch Koordinatoren und die Blätter durch Simulatoren repräsentiert werden. Die Kommunikation zwischen den Koordinatoren und Simulatoren erfolgt gerichtet über

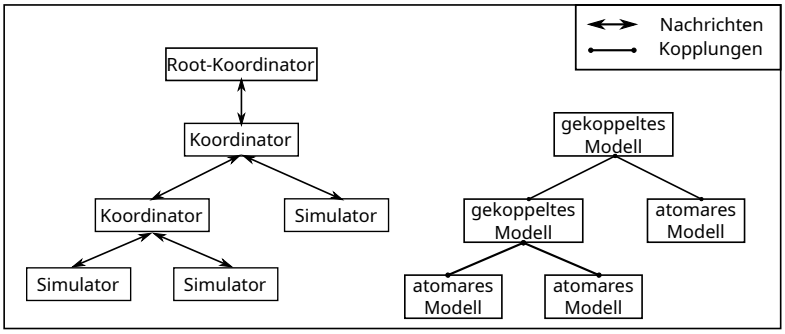


Abbildung 6: DEVS-Architektur

Nachrichten. Die Nachrichtentypen sind formalismusabhängig und werden in den nachfolgenden Abschnitten besprochen.

4.1 Classic DEVS

Der ursprüngliche DEVS-Formalismus wird heute als Classic DEVS bezeichnet. Nach Zeigler [107, S. 94-95] wird ein atomares Modell durch ein 7-Tupel beschrieben:

$$DEVS =< X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta >,$$

dabei ist

X	die Menge aller Eingangsereignisse,
S	die Menge aller Zustände,
Y	die Menge aller Ausgangsereignisse,
$\delta_{int} : S \to S$	die interne Zustandsüberführungsfunktion,
$\delta_{ext} : Q \times X \to S$	die externe Zustandsüberführungsfunktion,
$Q = \{(s, e) \mid s \in S, 0 \leq e < ta(s)\}$	die totale Zustandsmenge,
e	die vergangene Zeit seit der letzten Transition,
$\lambda : S \to Y$	die Ausgabefunktion und
$ta : S \to \mathbb{R}^{\geq 0} \cup \{\infty\}$	die Zeitfortschrittsfunktion.

Im Vergleich zum Moore-Automaten spielt bei DEVS die Abhängigkeit der Zeit eine wichtige Rolle. Jedem Zustand $s \in S$ wird eine Lebenszeit zugeordnet, die durch die Zeitfortschrittsfunktion $ta(s)$ beschrieben wird. Wenn das Ende der Lebenszeit erreicht ist, wird ein internes Ereignis ausgelöst. Ein internes Ereignis hat zur Folge, das über die λ-Funktion in Abhängigkeit vom Zustand $s \in S$ ein Ausgangsereignis generiert wird. Anschießend wird über die δ_{int} Funktion ein neuer Zustand $s \in S$

berechnet. Die Zeitfortschrittsfunktion gibt für diesen Zustand wieder die Lebensdauer an. Das Verhalten eines atomaren Modells bei Ereignissen von außen, welche als externe Ereignisse bezeichnet werden, wird durch die δ_{ext}-Funktion beschrieben. Diese Funktion berechnet aus der vergangenen Zeit e, dem aktuellen Zustand $s \in S$ und dem externen Ereignis $x \in X$ einen neuen Zustand $s \in S$. Für die Lebensdauer eines Zustandes gibt es zwei besondere Werte. Eine Lebenszeit von null Zeiteinheiten wird als *transitory State* bezeichnet und wird verwendet, um Mealy-Verhalten abzubilden. Im Gegensatz dazu gibt eine Lebenszeit von unendlich an, dass ein atomares Modell auf ein externes Ereignis wartet.

Ein gekoppeltes Modell wird in Classic DEVS nach Zeigler [107, S. 104-105] durch ein 8-Tupel beschrieben:

$$N =< X, Y, D, \{M_d\}, EIC, EOC, IC, Select >,$$

dabei ist

X	Menge der Eingangsereignisse,
Y	Menge der Ausgangsereignisse,
D	Menge der Komponentennamen,
$\{M_d\}$	Spezifikation der Komponenten $d \in D$,
EIC	Menge der externen Eingangskopplungen,
EOC	Menge der externen Ausgangskopplungen,
IC	Menge der internen Kopplungen,
$Select$	Konfliktlösungsfunktion.

Nach Zeigler [107, S. 169] kann ein gekoppeltes Modell aber auch durch ein 7-Tupel:

$$N =< X, Y, D, \{M_d\}, I_d, Z_{i,d}, Select >$$

beschrieben werden. Diese Beschreibung unterscheidet sich nur in der Spezifikation der Kopplungen:

I_d	Menge der Einwirkungen auf d,
$Z_{i,d}$	Menge der Kopplungsbeziehungen von i auf d.

In der vorliegenden Arbeit wird die Spezifikation nach Zeigler [107, S. 104-105] verwendet.

Eine Besonderheit bei Classic DEVS ist die Verwendung der *Select*-Funktion. Da der Classic DEVS Koordinator keine gleichzeitigen internen Ereignisse verarbeiten kann, erfolgt eine Sequenzialisierung der Bearbeitung durch die Select-Funktion, die eine imminente Komponente $d \in D$ auswählt.

Damit ein Modell in Classic DEVS ausgeführt werden kann, wird ein Simulator benötigt. Die Dokumentation der Funktionsweise des Simulators erfolgt in der Regel in Pseudocode und wird als abstrakter Simulator bezeichnet. Der Simulator besteht aus den Komponenten Simulator, Koordinator und Root-Koordinator. Die Simulatorkomponenten werden wie in Abbildung 6 gezeigt auf das DEVS-Modell gemappt, was zu einer modelläquivalenten hierarchischen Simulatorstruktur führt. Die Kommunikation zwischen den Komponenten erfolgt durch ein Nachrichtenkonzept, welches auch als Simulationsprotokoll bezeichnet wird. Bei Classic DEVS werden folgende Nachrichten verwendet:

- i-Nachricht (i, t): Die i-Nachricht dient der Initialisierung der Simulatoren und Koordinatoren. Sie wird zu Beginn der Simulation vom Root-Koordinator versendet. Die Koordinatoren senden diese Nachricht an alle Komponenten $d \in D$.
- *-Nachricht (*, t): Die *-Nachricht wird vom Root-Koordinator versendet. Sie betrifft alle Komponenten, die ein internes Ereignis zum Zeitpunkt t eingeplant haben. Da diese Nachricht aber nur ein Simulator gleichzeitig erhalten darf, wird über die *Select*-Funktion beziehungsweise mehrere *Select*-Funktionen genau ein Simulator ausgewählt.
- y-Nachricht (y, t): Diese Nachricht wird vom Simulator beziehungsweise Koordinator an den jeweils übergeordneten Koordinator geschickt und enthält ein Ausgangsereignis. Dieses Ereignis wird auf Basis der im DEVS-Modell spezifizierten Kopplungen an die Simulatoren der betroffenen Komponenten als x-Nachricht versendet oder als y-Nachricht an den übergeordneten Koordinator.
- x-Nachricht (x, t): Diese Nachricht wird vom Koordinator zu einem untergeordneten Simulator geschickt und vermittelt ein externes Ereignis.

In machen Fällen wird auch noch eine done-Nachricht eingeführt, die das Ende der Bearbeitung einer Nachricht angibt. Diese Nachricht ist aber implementierungsabhängig und kann auch implizit durch einen Rücksprung aus einer Funktion realisiert werden. Das prinzipielle Nachrichtenkonzept zwischen den Komponenten des Simulators ist in Abbildung 7 dargestellt.

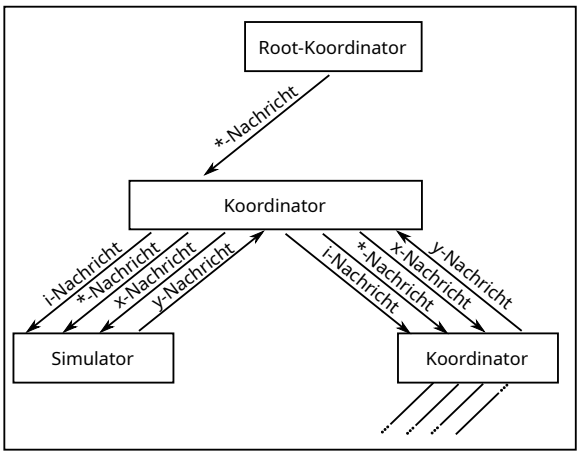


Abbildung 7: Nachrichtenkonzept im abstrakten Simulator.

Damit ein abstrakter Simulator implementiert werden kann, muss die Bearbeitung der Nachrichten in den jeweiligen Komponenten spezifiziert werden. Listing 4.1 zeigt die Spezifikation der Komponente Simulator nach Pichler [74, S. 178-179] beziehungsweise Zeigler [107, S. 199].

Listing 4.1: Classic DEVS Simulator.

```
1  Devs-simulator
2  variables:
3    parent                                              parent coordinator
4    tl                                                  time of last event
5    tn                                                  time of next event
6    DEVS                              associated model with total state (s, e)
7    y                               current output value of the associated model
8  when receive i-message (i, t) at time t
9    tl = t - e
10   tn = tl + ta(s)
11 when receive *-message (*, t) at time t
12   if t ≠ tn then
13     error: bad synchronization
14   end if
15   y = λ(s)
16   send y-message (y, t) to parent coordinator
17   s = δint (s)
18   tl = t
19   tn = tl + ta(s)
20 when receive x-message (x, t) at time t with input value x
21   if not (tl ≤ t ≤ tn) then
22     error: bad synchronization
23   end if
24   e = t - tl
25   s = δext (s, e, x)
26   tl = t
27   tn = tl + ta(s)
28 end Devs-Simulator
```

Bei einer i-Nachricht werden die Zeiten tl (letztes Ereignis) und tn (nächstes Ereignis) berechnet. Über die Zeit tn, die über die $ta(s)$-Funktion des atomaren Modells bestimmt wird, gibt der Simulator das nächste geplante Ereignis an.

Die *-Nachricht wird einem Simulator bei $t = tn$ geschickt. Wenn dies nicht der Fall ist, dann liegt ein Fehler vor. In dieser Nachricht wird im ersten Schritt die λ-Funktion des atomaren Modells ausgeführt, um damit ein Ausgangsereignis y zu generieren. Dieses Ausgangsereignis sendet der Simulator per y-Nachricht an den übergeordneten Koordinator. Danach ruft der Simulator die δ_{int}-Funktion des atomaren Modells auf, um einen neuen Zustand s zu berechnen. Zum Schluss werden durch Aufruf der ta-Funktion die Zeiten tl und tn berechnet.

Durch die x-Nachricht empfängt der Simulator ein Eingangsereignis. Ein Ereignis darf nur empfangen werden, wenn die aktuelle Simulationszeit t zwischen dem Zeitpunkt des letzten Ereignises tl und des nächsten geplanten Ereignises tn liegt, ansonsten liegt ein Fehler vor. Wenn die Simulationszeit $t = tl$ beziehungsweise $t = tn$ beträgt, deutet dies auf konkurrierende[1] Ereignisse hin, welche zulässig sind. Nach der Berechnung der vergangenen Zeit e wird ein neuer Zustand s über die δ_{ext}-Funktion des atomaren Modells berechnet. Danach werden die Zeiten tl und tn berechnet.

Damit gekoppelte Modelle, auch als Netzwerke bezeichnet, durch den Simulator ausgeführt werden können, wird ein Koordinator benötigt. Dieser koordiniert die Nachrichten zwischen den Simulatoren beziehungsweise weiteren Koordinatoren. Wie eingangs gezeigt, gibt es für Classic DEVS zwei Versionen für die Spezifikation gekoppelter Modelle. Demgemäß gibt es auch zwei Spezifikationen für den Koordinator. Listing 4.2 zeigt den Pseudocode des Koordinators nach Pichler [74, S. 180-181], auf welchen in dieser Arbeit Bezug genommen wird. Für die zweite Version sei auf Zeigler [107, S. 202] verwiesen.

Listing 4.2: Classic DEVS Koordinator.

```
1  Devs-coordinator
2  variables :
3    DEVN = (X, Y, D, {M_d}, EIC, EOC, IC, Select)  the associated network
4    parent                                          parent coordinator
5    tl                                              time of last event
6    tn                                              time of next event
7    event-list          list of elements (d, tn_d) sorted by tn_d and Select
8    d*                                             selected imminent child
9
10 when receive i-message (i, t) at time t
11   for d∈D do
12     send i-message (i, t) to child d
13   end for
14   sort event-list according to tn_d and Select
15   tl = max{tl_d|d∈D}
16   tn = min{tn_d|d∈D}
17 when receive *-message (*, t) at time t
18   if t ≠ tn then
```

[1] gleichzeitiges internes und externes Ereignis

```
19      error: bad synchronization
20    end if
21    d* = first(event-list)
22    send *-message (*, t) to d*
23    sort event-list according to tn_d and Select
24    tl = t
25    tn = min{tn_d|d∈D}
26  when receive x-message (x, t) at time t with external input x
27    if not (tl ≤ t ≤ tn) then
28      error: bad synchronization
29    end if
30    receivers := components r∈D influenced by external input x
31    for r in receivers do
32      send x-messages (x_r , t) with input value xr got from x
            through EIC
33    end for
34    sort event-list according to tn_d and Select
35    tl = t
36    tn = min{tn_d|d∈D}
37  when receive y-message (y, t) with output y from d*
38    if the output from d* influences the EOC then
39      send y-message (y_N, t) to parent with value y_N got from y
            through EOC
40    end if
41    check IC to get children influenced by the output y of d*
42    receivers := components r∈D influenced by d*
43    for r in receivers do
44      send x-messages (y_r, t ) to r with input value y_r got from y
            through IC
45    end for
46  end Devs-coordinator
```

Beim Empfang einer i-Nachricht wird diese an die jeweils untergeordneten Komponenten des Koordinators versendet. Anschließend wird eine Ereignisliste erstellt und sortiert. Zum Schluss werden die Zeiten tl und tn bestimmt.

Wenn eine *-Nachricht zum Zeitpunkt t empfangen wurde, wird zuerst geprüft, ob $t = t_n$ entspricht. Wenn dies nicht der Fall ist, liegt ein Fehler vor. Im Anschluss wird d^* aus der Ereignisliste ausgewählt. Da diese Liste sortiert ist, ist d^* das erste Element. Die *-Nachricht wird dann an die Simulatorkomponente von d^* geschickt. Zum Schluss wird die Ereignisliste wieder sortiert und die Zeiten tl und tn bestimmt.

Wenn der Koordinator eine x-Nachricht empfängt, wird zuerst geprüft, ob der aktuelle Simulationszeitpunkt zwischen tl und tn liegt. Wenn dies nicht der Fall ist, liegt ein Fehler vor. Da die x-Nachricht ein Eingangsereignis repräsentiert, müssen die Empfänger dieses Ereignisses bestimmt werden. Dies geschieht durch Auswertung der Kopplungsbeziehungen, die in External Input Coupling (EIC) des zugeordneten DEVS-Modells spezifiziert wurden. Daraufhin wird allen Simulatorkomponenten, deren Modelle von x abhängig sind, eine x-Nachricht geschickt. Zum Schluss wird wieder die Ereignisliste sortiert und es werden die Zeiten tl und tn bestimmt.

Die Behandlung der y-Nachricht ist schematisch in Abbildung 8 dargestellt. Beim Empfang einer y-Nachricht wird zuerst anhand der External Output Coupling (EOC)

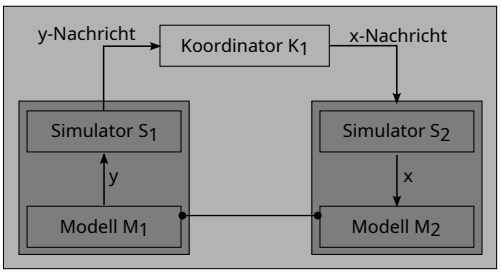


Abbildung 8: Behandlung einer y-Nachricht durch den Koordinator.

Spezifikation geprüft, ob dieses Ereignis an externe Outputs des gekoppelten DEVS-Modells zu senden ist. Dies ist in den Kopplungen von EOC spezifiziert. Wenn dies der Fall ist, wird eine y-Nachricht an den übergeordneten Koordinator geschickt. Im Anschluss wird anhand der Internal Coupling (IC) Spezifikation geprüft, ob dieses Ereignis an interne Komponenten des gekoppelten Modells zu senden ist. Wenn dies der Fall ist, werden die Empfänger über die Kopplungen in IC bestimmt und es wird an diese eine x-Nachricht geschickt.

Die letzte Komponente im abstrakten Simulator ist der Root-Koordinator, welcher in Anlehnung an Zeigler [107, S. 205] wie in Listing 4.3 dargestellt werden kann.

Listing 4.3: Classic DEVS Root-Koordinator.

```
1  Devs-root-coordinator
2  variables:
3    t                                              current simulation time
4    child                                  direct subordinate devs-coordinator
5
6  t = t0
7  send i-message (i, t) to child
8  t = tn of its child
9
10 while check termination condition
11   send *-message (*, t) to child
12   t = tn of its child
13 end while
14
15 end Devs-root-coordinator
```

Der Root-Koordinator setzt die Simulationszeit t auf den Startzeitpunkt t_0. Danach folgt die Initialisierung der anderen Simulatorkomponenten durch das Senden einer i-Nachricht an den untergeordneten Koordinator oder Simulator. Nach der Initialisierung wird t auf den nächsten Simulationszeitpunkt gesetzt, der vom untergeordneten Koordinator bestimmt wurde. Als Nächstes folgt die Simulationsschleife, in der eine *-Nachricht an den untergeordneten Koordinator geschickt wird. Nach Abarbeitung der *-Nachricht erfolgt die Zeitfortschaltung auf den nächsten Ereigniszeitpunkt. Die Simulationsschleife wird solange ausgeführt, bis eine definierte Abbruchbedingung erfüllt ist.

4.2 Parallel DEVS

Eine Weiterentwicklung von Classic DEVS ist der von Chow [19, 18] 1994 veröffentlichte PDEVS-Formalismus. Dieser Formalismus ist eine Weiterentwicklung von E-DEVS, welcher von Wang [98, 97] 1992 für das Hochleistungsrechnen entwickelt wurde. Die Idee von Wang beziehungsweise Chow ist in erster Linie die parallele Ausführung von zeitgleichen internen Ereignissen. Damit dies möglich ist, muss die Select-Funktion (vgl. Abschn. 4.1) der gekoppelten Modelle durch andere Mechanismen ersetzt werden. Da E-DEVS im Laufe der Jahre nicht weiter aufgegriffen wurde und PDEVS als direkter Nachfolger betrachtet werden kann, wird hier nicht weiter auf die Funktionsweise von E-DEVS eingegangen.

Die Spezifikation eines atomaren Modells in PDEVS ähnelt der von Classic DEVS. Es gibt eine funktionale Erweiterung, die das Verhalten von konkurrierenden Ereignissen spezifiziert und es wurden die sogenannten *Bags* eingeführt. Ein Bag sammelt alle Eingangsereignisse, die zu einem Zeitpunkt an einen Eingang eintreffen. Ein atomares Modell wird nach Chow [19] durch ein 8-Tupel beschrieben:

$$PDEVS =< X, S, Y, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta >,$$

dabei ist

X	die Menge aller Eingangsereignisse,
S	die Menge aller Zustände,
Y	die Menge aller Ausgangsereignisse,
$\delta_{int} : S \to S$	die interne Zustandsüberführungsfunktion,
$\delta_{ext} : Q \times X^b \to S$	die externe Zustandsüberführungsfunktion,
X^b, Y^b	Menge der Input-, Output-Bags,
$Q = \{(s, e) \mid s \in S,\ 0 < e < ta(s)\}$	die totale Zustandsmenge,
$\delta_{con} : S \times X^b \to S$	confluent beziehungsweise interne/externe Zustandsüberführungsfunktion,
e	die vergangene Zeit seit der letzten Transition,
$\lambda : S \to Y^b$	die Ausgabefunktion und
$ta : S \to \mathbb{R}^{\geq 0} \cup \{\infty\}$	die Zeitfortschrittsfunktion.

Die δ_{con} wird bei einem gleichzeitigen internen und externen Ereignis, was als konfluentes Ereignis bezeichnet wird, ausgeführt. Der Simulator für ein atomares PDEVS-Modell muß dementsprechend konfluente Ereignisse erkennen und behandeln. Bei Classic DEVS müssen Zustandsübergänge auf Grund von konfluenten Ereignissen durch die δ_{ext} Funktion behandelt werden. Dies gelingt nur, indem die vergangene Zeit e überprüft und dieser Fall entsprechend in der Modellbeschreibung spezifiziert wird. Die Notwendigkeit der Bags ist darin begründet, dass externe Ereignisse, die durch gleichzeitige interne und konfluente Ereignisse erzeugt wurden, gesammelt werden müssen.

Ein gekoppeltes Modell wird bei PDEVS nach Chow [19] durch ein 6-Tupel beschrieben:

$$N =< X, Y, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} >,$$

dabei ist

X	Menge der Eingangsereignisse,
Y	Menge der Ausgangsereignisse,
D	Menge der Komponentennamen,
$\{M_i\}$	Beschreibung der Komponenten mit $i \in D$,
$\{I_i\}$	Menge der Einwirkungen auf i,
$\{Z_{i,j}\}$	Menge der Kopplungsbeziehungen von i auf j.

Wie eingangs erwähnt, ist eine Zielstellung von PDEVS die parallele Ausführung gleichzeitiger interner Ereignisse. Demgemäß definieren gekoppelte PDEVS-Modelle keine *Select*-Funktion.

Das bei Classic DEVS (vgl. Abschn. 4.1) zur Abarbeitung von DEVS-Modellen eingeführte Nachrichtenkonzept wird auch bei PDEVS verwendet. Zwar hat Chow 1994 [17] einen abstrakten Simulator für PDEVS vorgestellt, dieser benutzt aber andere Bezeichnungen für die Nachrichten. Die übliche abstrakte Simulatorbeschreibung ist die nach Nutaro [107, S. 351-353]. In verschiedenen Veröffentlichungen wurde über Erfahrungen berichtet, dass der abstrakte Simulator gemäß Nutaro nicht immer erwartungsgemäß funktioniert. Nach Schwatinski [83] müsste der abstrakte Simulator um eine Nachricht erweitert werden. Eine ähnliche Beobachtung machte Martin in [57] bei der Simulation von Modellen mit *transitory States*. Der Autor dieser Arbeit konnte bei einer eigenen Implementierung dieses abstrakten Simulators kein fehlerhaftes Verhalten beobachten.

In Anlehnung an Nutaro [107, S. 351] kann der Simulator für atomare Modelle wie in Listing 4.4 beschrieben werden:

Listing 4.4: PDEVS Simulator.

```
1  Parallel-Devs-simulator
2  variables:
3    parent                                                parent coordinator
4    tl                                                    time of last event
5    tn                                                    time of next event
6    DEVS                              associated model with total state (s, e)
7    y                                                     output message bag
8  when receive i-message (i, t) at time t
9    tl = t - e
10   tn = tl + ta(s)
11 when receive *-message (*, t) at time t
12   if t ≠ tn then
13     error: bad synchronization
14   end if
15   y = λ(s)
```

```
16    send y-message (y, t) to parent coordinator
17  when receive x-message (x, t) at time t with input value x
18    if x = ∅ and t = tn then
19      s = δ_int(s)
20    else if x ≠ ∅ and t = tn then
21      s = δ_con(s)
22    else
23      e = t - tl
24      s = δ_ext(s, e, x)
25    end if
26    tl = t
27    tn = tl + ta(s)
28  end Parallel-Devs-Simulator
```

Der abstrakte Simulator von PDEVS hat viele Ähnlichkeiten mit dem von Classic DEVS. Modifikationen gibt es hinsichtlich der Zustandsüberführungsfunktionen. Alle δ-Funktionen werden in der Bearbeitungsroutine der x-Nachricht bearbeitet. Es wird durch eine Fallunterscheidung die entsprechende Überführungsfunktion ausgewählt. Dabei wird auch die leere Menge als Indikator verwendet.

In Anlehnung an Nutaro [107, S. 352-353] kann der Koordinator für ein gekoppeltes PDEVS Modell wie in Listing 4.5 beschrieben werden.

Listing 4.5: PDEVS Koordinator.

```
1   Parallel-Devs-coordinator
2   variables :
3     DEVN = (X, Y, D, {M_d}, {I_d}, {Z_i,d})      the associated coupled model
4     parent                                                parent coordinator
5     tl                                                    time of last event
6     tn                                                    time of next event
7     event-list                      list of elements (d, tn_d) sorted by tn_d
8     IMM                                                    imminent children
9     mail                                                     output mail bag
10    y_parent                                        output message bag to parent
11    y_d                             set of output message bags for each child d
12
13  when receive i-message (i, t) at time t
14    for d∈D do
15      send i-message (i, t) to child d
16    end for
17    sort event-list according to tn_d
18    tl = max{tl_d|d∈D}
19    tn = min{tn_d|d∈D}
20  when receive *-message (*, t) at time t
21    if t ≠ tn then
22      error: bad synchronization
23    end if
24    IMM = min(event-list)                     components with minimum tn
25    for r ∈ IMM do
26      send *-message (*, t) to r
27    end for
28  when receive x-message (x, t) at time t with external input x
29    if not (tl ≤ t ≤ tn) then
30      error: bad synchronization
```

```
31    end if
32    receivers = {r|r∈D, N ∈ I_r, Z_{N,r}(x) ≠ ∅}
33    for r ∈ receivers do
34      send x-message(Z_{N,r}(x), t) to r
35    end for
36    for r ∈ IMM and not in receivers do
37      send x-message(∅, t) to r
38    end for
39    sort event-list according to tn_d
40    tl = t
41    tn = min{tn_d|d∈D}
42  when receive y-message (y, t) with output y from d
43    if this is not the last d in IMM then
44      add(y_d, d) to mail
45      mark d as reporting
46    else
47      y_parent = ∅
48    end if
49    for d ∈ I_N do
50      if Z_{d,N}(y_d) ≠ ∅ then
51        add y_d to y_parent
52      end if
53    end for
54    send y-message(y_parent,t) to parent
55    for child r, x_r = ∅ do
56      for d such that d ∈ I_r do
57        if Z_{d,r}(y_d) ≠ ∅ then
58          add y_d to y_r
59        end if
60      end for
61    end for
62    receivers = {r|r ∈ D, y_r ≠ ∅}
63    for r ∈ receivers do
64      send x-message(y_r, t) to r
65    end for
66    for r ∈ IMM and not in receivers do
67      send x-message(∅, t) to r
68    end for
69    sort event-list according to tn_d
70    tl = t
71    tn = min{tn_d|d∈D}
72  end Parallel-Devs-coordinator
```

In der *-Nachricht werden alle Komponenten, bei denen die Bedingung $t = tn$ zutrifft, ausgewählt und in der Variablen IMM gespeichert. Diese erhalten im Anschluss eine *-Nachricht. Hier zeigt sich das parallele Abarbeiten von zeitgleichen internen Ereignissen. Als Folge einer *-Nachricht produzieren die Simulatoren nach Listing 4.4 ein Ausgangsereignis, welches beim Koordinator als y-Nachricht eintrifft. Alle bei einem Koordinator eingehenden y-Nachrichten werden in einer Liste gesammelt, bis alle empfangen wurden. Aus den empfangenen y-Nachrichten wird durch Auswertung der Kopplungsrelationen eine y-Nachricht für den übergeordneten Koordinator zusammengestellt und an diesen verschickt. Danach wird an alle untergeordneten Komponenten, die von einer y-Nachricht beeinflußt werden (genannt

Empfänger), eine x-Nachricht geschickt. Alle Komponenten die in der Variablen IMM enthalten sind und keine x-Nachricht bekommen haben, erhalten eine leere x-Nachricht.

Bekommt der Koordinator eine x-Nachricht, werden aus den Kopplungsbeziehungen die Empfänger ermittelt und an diese die x-Nachricht weitergeleitet. Komponenten, die in der Variablen IMM registriert und keine Empfänger der x-Nachricht sind, erhalten eine leere x-Nachricht. Am Ende jeder y-Nachricht beziehungsweise x-Nachricht wird die Ereignisliste sortiert und die Zeiten tn und tl werden ermittelt.

In der Definition von PDEVS nach Chow [19] gibt es keine benannten *Ports*. In der wesentlich späteren Publikation von Zeigler, Prähofer und Kim [106, S. 90] wird PDEVS unter Verwendung von Ports eingeführt. Die prinzipielle Dynamikspezifikation sowie der abstrakte Simulator sind identisch. Wann und durch wen die Erweiterung von PDEVS mit Ports eingeführt wurde konnte nicht eindeutig recherchiert werden.

4.3 Revised PDEVS

Ein Nachteil von Classic DEVS und PDEVS ist, dass atomare Modelle nicht unmittelbar Mealy-Verhalten abbilden können. Mealy-Verhalten kann nur mit Übergangszuständen, nach Zeigler [107, S. 95] *transitory States*, modelliert werden. Preyser zeigt in [77], dass die Ausführung von Modellen mit *transitory States* durch den PDEVS Simulator zu fehlerhaften Ergebnissen führen kann. Durch die zusätzlichen Zustände steigt der Modellierungsaufwand und die Modellkomplexität. Bereits 2007 veröffentlichte Traoé [92] einen Formalismus namens Easy DEVS, der die Modellierung von Mealy-Verhalten unterstützen soll. Ein weiterer Ansatz ist der von Preyser im Jahr 2018 [76] veröffentlichte RPDEVS-Formalismus. Im Jahr 2019 veröffentlichte Preyser [75] dazu einen abstrakten Simulator.

Ein atomares Modell wird nach Preyser [76] als 6-Tupel dargestellt:

$$RPDEVS =< X, S, Y, \delta, \lambda, ta >,$$

dabei ist:

X	die Menge aller Eingangsereignisse,
S	die Menge aller Zustände,
Y	die Menge aller Ausgangsereignisse,
$\delta : Q \times X^b \to S$	die Zustandsüberführungsfunktion,
X^b, Y^b	Menge der Input-, Output-Bags,
$Q = \{(s, e) \mid s \in S, 0 < e < ta(s)\}$	der totale Zustand,
e	die vergangene Zeit seit der letzten Transition,
$\lambda : S \times X^b \to Y^b$	die Ausgabefunktion und
$ta : S \to \mathbb{R}^{\geq 0} \cup \{\infty\}$	die Zeitfortschrittsfunktion.

Die wichtigsten Änderungen gegenüber PDEVS beziehen sich auf die Zustandsüberführungsfunktion δ und die Ausgabefunktion λ. Alle Zustandsüberführungsfunktionen wurden in einer Funktion vereinigt und die Ausgabefunktion ist nun direkt von den Eingangsereignissen abhängig. An der Ausgabefunktion ist das Mealy-Verhalten direkt erkennbar. Die Bags werden auch hier benötigt, da mehrere externe Ereignisse gleichzeitig auftreten können.

Die Spezifikation eines gekoppelten RPDEVS-Modells ist identisch zu PDEVS (s. Abschn. 4.2).

Das Nachrichtenkonzept bei RPDEVS basiert auf PDEVS. Im Gegensatz zu PDEVS berücksichtigt die *-Nachricht auch Eingangsereignisse.

In Anlehnung an [75] kann der abstrakte Simulator für ein atomares Modell wie in Listing 4.6 dargestellt werden.

Listing 4.6: RPDEVS Simulator.

```
1  RPDEVS-simulator
2  variables:
3    parent                                                parent coordinator
4    tl                                                    time of last event
5    tn                                                    time of next event
6    RDEVS                          associated model with total state (s, e)
7    y                                                     output message bag
8  when receive i-message (i, t) at time t
9    tl = t - e
10   tn = tl + ta(s)
11 when receive *-message (*, x, t) at time t
12   e = tl - t
13   y = λ(s, e, x)
14   send y-message (y, t) to parent coordinator
15 when receive x-message (x, t) at time t with input value x
16   s = δ(s, e, x)
17   tl = t
18   tn = tl + ta(s)
19 end RPDEVS-Simulator
```

Anders als bei den Simulatoren von Classic DEVS (Abschn. 4.1) und PDEVS (Abschn. 4.2) empfängt der Simulator eines atomaren Modells bei einer *-Nachricht ein Eingangsereignis. Das Eingangsereignis wird zusammen mit dem aktuellen Zustand an die λ-Funktion übergeben, welche daraufhin ein Ausgangsereignis y generiert. Dieses Ausgangsereignis wird über eine y-Nachricht an den übergeordneten Koordinator gesendet. Der Empfang einer x-Nachricht führt zum Aufruf der δ-Funktion, welche einen neuen Zustand des atomaren Modells berechnet.

Der Koordinator von RPDEVS ist in Anlehnung an [75] in Listing 4.7 dargestellt.

Listing 4.7: RPDEVS Koordinator.

```
1  RPDEVS-coordinator
2  variables :
3    DEVN = (X, Y, D, {M_d}, {I_d}, {Z_i,d})      the associated coupled model
4    parent                                                parent coordinator
```

```
  tl                                                    time of last event
  tn                                                    time of next event
  event-list                        list of elements (d, tn_d) sorted by tn_d
  IMM                                                    imminent children
  y_coupling                                       output message of coupling
  x_dr                                                        sub input bags
  x_r                                                 input bag of component r
  y_dN                                             sub output bag of coupling
  INF                                              set of influenced children
  INF'                                               INF for next lamda-iter.
  DELTA            set of children who need to conduct a state transition
  CHECK                             components with withdrawn input message

when receive i-message (i, t) at time t
  for d∈D do
    send i-message (i, t) to child d
  end for
  sort event-list according to tn_d
  tl = max{tl_d|d∈D}
  tn = min{tn_d|d∈D}
when receive *-message (*, x, t) at time t
  y_coupling = {}
  for (d,tn_d) in event-list with tn_d = t
    add d to IMM, DELTA and INF
    remove (d,tn_d) from event-list
  end
  for r in D with N in I_r
    if x_Nr != Z_Nr(x)
      x_Nr = Z_Nr(x)
      add r to INF and DELTA
      if x_Nr={}
        add r to CHECK
      end
    end
  end
  for r in INF
    x_r = {x_dr : d in I_r, x_dr != {}}
  end
  while CHECK != {}
    pick and remove r from CHECK
    if x_r={}
      if r not in IMM
        remove r from INF and DELTA
        for d in D with r in I_d
          x_rd = {}
          remove x_rd from x_d
          add d to CHECK
        end
        if r in I_N
          y_rN = {}
        end
      end
    end
  end
  INF'={}
```

```
60    for r in INF
61      send *-message(*,x_r,t)
62    end
63  when receive x-message (x, t) at time t with external input x
64    for r in DELTA
65      send x-message(x_r,t) to r
66    end
67    sort event-list according to tn_d
68    tl = t
69    tn = min{tn_d|d∈D}
70    IMM = {}
71  when receive y-message (y, t) with output y from d
72    remove d from INF
73    if d in I_N
74      y_dN = Z_dN(y_d)
75    end
76    for r in D with d in I_r
77      if x_dr != Z_dr(y_d)
78        x_dr = Z_dr(y_d)
79        if x_dr={}
80          add r to CHECK
81        end
82        add r to INF' and DELTA
83      end
84    end
85    if INF = {}
86      INF = INF'
87      INF' = {}
88      for r in INF
89        x_r = {x_dr : d in I_r, x_dr != {}}
90      end
91      while CHECK != {}
92        pick and remove r from CHECK
93        if x_r={}
94          if r not in IMM
95            remove r from INF and DELTA
96            for-each d in D with r in I_d
97              if x_rd != {}
98                x_rd = {}
99                remove x_rd from x_d
100               add d to INF, DELTA and CHECK
101             end
102             if r in I_N
103               y_rN = {}
104             end
105           end
106         end
107       end
108     end
109     for r in INF
110       send *-message(*,x_r,t) to component r
111     end
112     if INF = {}
113       y_coupling={y_dN : d in I_N, y_dN!={}}
114       send y-message(y_coupling,t) to parent
```

```
115       end
116     end
117   end RPDEVS-coordinator
```

Die erste Auffälligkeit dieses RPDEVS-Koordinators ist die Komplexität. Der Koordinator basiert im Gegensatz zu den Koordinatoren von Classic DEVS und PDEVS auf einem iterativen Lösungsansatz. Die größten Änderungen betreffen die *-Nachricht und y-Nachricht.

In der *-Nachricht (Zeile 25-62) werden zunächst alle Komponenten aus der Ereignisliste entnommen, bei denen die Bedingung $tn = t$ zutrifft (Zeile 27-30). Diese werden in den Variablen IMM, $DELTA$, und INF gespeichert (Zeile 27-30). Im Anschluss werden die externen Eingangskopplungen des gekoppelten Modells[2] geprüft. Sind Komponenten im Netzwerk von Eingangsereignissen x betroffen, dann werden diese zu INF und $DELTA$ hinzugefügt. Ist der Wert eines Eingangsereignisses die leere Menge, dann wird die betroffene Komponente zusätzlich zu $CHECK$ hinzugefügt. In der Variablen $CHECK$ befinden sich nun alle Komponenten, bei denen ein Eingangsereignis zurückgenommen wurde. Von diesen Komponenten müssen wiederum die beeinflussten Komponenten ermittelt werden. Diese werden ebenfalls in der Variablen $CHECK$ hinzugefügt. Gleichzeitig werden die Eingangsereignisse, die zurückgezogen wurden, aus den Input-Bags der Komponenten entfernt (Zeile 43-58). Wenn diese Komponenten nicht in der Variablen IMM enthalten sind, dann werden diese auch aus den Variablen INF und $DELTA$ entfernt. Danach erhalten alle Komponenten in INF eine *-Nachricht mit den dazugehörigen Eingangsereignissen (Zeile 60-62).

Beim Empfang einer y-Nachricht (Zeile 71-116) wird zunächst die Komponente d, die die Quelle des Ereignisses repräsentiert, aus der Variablen INF entfernt (Zeile 72). Im Anschluss wird geprüft, ob das Ereignis einen Einfluss auf das externe Netzwerk hat, wenn dies der Fall ist, wird das Ereignis in das Output-Bag geschrieben (Zeile 73-75). Danach werden alle Komponenten, die von diesem Ereignis betroffen sind, in die Variablen INF' und $DELTA$ geschrieben. Wenn das Ereignis eine leere Menge repräsentiert, dann wird die Komponente zusätzlich in die Variable $CHECK$ geschrieben (Zeile 76-84). Danach wird geprüft, ob die Variable INF leer ist. Wenn diese Variable leer ist, bedeutet dies, dass der aktuelle Iterationsschritt beendet ist und ein neuer beginnt. Dazu wird der Inhalt der Variablen INF' in INF verschoben (Zeile 85-90). Im Anschluss wird über die Variable $CHECK$ geprüft, ob Eingangsereignisse zurückgezogen wurden und entsprechende Auswirkungen korrigiert werden müssen (Zeile 91-108). Danach bekommen alle Komponenten in der Variablen INF eine *-Nachricht mit den zugewiesenen Eingangsereignissen (Zeile 109-111). Wenn die Variable INF leer ist, dann wird die y-Nachricht für den übergeordneten Koordinator vorbereitet und verschickt (Zeile 112-115).

Der Empfang einer x-Nachricht (Zeile 63-70) kennzeichnet das Ende der Iteration. Hier werden die endgültigen Eingangsereignisse an die in $DELTA$ gespeicherten

[2] im Kontext von RPDEVS wird das gekoppelte Modell als Netzwerk bezeichnet

Komponenten versendet. Im Anschluss wird die Ereignisliste des Koordinators sortiert, die Zeiten tl sowie tn bestimmt und die Listenvariable IMM auf leere Liste gesetzt.

4.4 Zusammenfassung

In diesem Kapitel wurde die Entwicklung des DEVS-Formalismus betrachtet. Die ersten Erweiterungen hatten das Ziel, den sequentiellen Simulationsalgorithmus in einen nebenläufigen[3] Algorithmus zu transformieren. Daraus folgte der Formalismus PDEVS. Dieser Formalismus war sehr lange State of the Art, aber im letzten Jahrzehnt stellte sich heraus, dass DEVS und PDEVS Defizite in Bezug auf die Modellierung von Mealy-Verhalten aufweisen. Warum diese Erkenntnis so viel Zeit benötigte, kann nicht konkret beantwortet werden. Vermutlich liegt es an den mit DEVS bearbeiteten Anwendungen, die anfangs maßgeblich aus dem militär-logistischen Bereich kamen.

Der Einsatz von DEVS im Kontext forschungsnaher ingenieurtechnischer Anwendungen führte zur Erweiterung RPDEVS, die eine einfachere und übersichtlichere Modellierung von Modellen mit Mealy-Verhalten unterstützt. Der RPDEVS-Formalismus wurde noch für Anwendungen mit Logik-Gattern eingesetzt, aber sonst nicht weiter verfolgt. Junglas [50] zeigte, dass auch der RPDEVS-Formalismus noch Defizite aufweist. An einem einfachen Beispiel weist Junglas nach, dass die Kausalität von gleichzeitigen Ereignissen nicht immer richtig umgesetzt wird. Ein weiteres Problem stellen bei RPDEVS Systeme mit Rückkopplungen dar, die algebraische Schleifen enthalten. Hier kann es passieren, dass die Abarbeitung (Simulation) von Modellen zu einer Endlosschleife führt, verursacht durch die Iteration in der y-Nachricht. Dieses Verhalten begründet Junglas mit einer zu starken Vereinfachung der Realität auf Modellebene.

Junglas schlägt im Kontext seiner Untersuchungen eine erneute Erweiterung vor, die die Kausalität von gleichzeitigen Ereignissen wieder herstellen soll. Eine solche Erweiterung wird im nachfolgenden Kapitel vorgestellt.

[3] Nebenläufigkeit im Sinne von Parallelität.

5 NSA-DEVS

Aufbauend auf den Ideen von RPDEVS schlägt Junglas [50] eine Erweiterung des DEVS-Formalismus vor. Den Grundgedanken formuliert er wie folgt:

„Modeling experience teaches us that a mathematical problem in the description or simulation of a model often has its roots in an oversimplification of the system one wants to describe."

Sein Ziel ist es, zu starke Vereinfachungen der Realität in der Modellierung zu kompensieren. Wenn wir technische Systeme betrachten, dann fällt es uns schwer, realistische Beispiele zu finden, die ein Mealy-Verhalten besitzen. Das liegt vermutlich daran, dass jedes System eine Verzögerung besitzt, welche wir aber vernachlässigen oder nicht betrachten. Junglas schlägt vor:

- die Verarbeitung von Eingangsereignissen zu verzögern und
- die Lebensdauer von Zuständen immer größer als Null zu setzen.

Durch die zweite Bedingung werden die bei PDEVS üblichen *transitory States* verboten. Damit die Simulationszeit durch die Verzögerungen nicht verfälscht wird, schlägt Junglas die Verwendung von hyperreellen Zahlen ${}^*\mathbb{R}$ aus der NSA vor [36].

Die Menge der hyperreellen Zahlen beinhaltet die infinitesimale Zahl $\varepsilon \in {}^*\mathbb{R}$, die größer ist als Null aber kleiner als jede positive reelle Zahl. Durch die Verwendung von ε können hyperreelle Zahlen durch zwei reelle Zahlen, in der Form $a + b\varepsilon$ mit $a, b \in \mathbb{R}$ ausgedrückt werden. Dies ist zwar nur eine Teilmenge der hyperreellen Zahlen, die aber für die Modellierung ausreichend ist. Es ist auch möglich, eine hyperreelle Zahl einer reellen Zahl zuzuordnen, dies wird als Standardanteil $st(a + b\varepsilon) = a$ bezeichnet. Hier ist zu erkennen, dass der infinitesimale Anteil durch die hyperreellen Zahlen bestimmt wird und somit keinen Einfluss auf die reellwertige Simulationszeit hat. Der Name des neuen DEVS-Formalismus ist NSA-DEVS. Der Ansatz, hyperreelle Zahlen zur Darstellung der Simulationszeit einzusetzen, ist nicht völlig neu und wurde bereits von Barros [3] für strukturvariable Systeme im Kontext der hybriden Simulation verfolgt.

In den nachfolgenden Abschnitten werden die Modellbeschreibung und die Abarbeitungsalgorithmen des NSA-DEVS-Formalismus entwickelt. Weiterhin wird ein Diagramm zur visuellen Beschreibung von NSA-DEVS-Modellen eingeführt.

5.1 Modellbeschreibung

In [44] wurde vom Autor dieser Arbeit aufbauend auf den Ideen von Junglas [50] der NSA-DEVS-Formalismus spezifiziert. Wie bei den DEVS-Formalismen im Kapitel 4 unterscheidet auch NSA-DEVS zwischen atomaren und gekoppelten Modellen. Ein atomares Modell wird durch ein 7-Tupel repräsentiert:

$$NSA\text{-}DEVS =< X, S, Y, \tau, ta, \delta, \lambda >,$$

dabei ist

X	die Menge aller Eingangsereignisse an allen Ports,
S	die Menge aller Zustände,
Y	die Menge aller Ausgangsereignisse an allen Ports,
$\tau \in {}^*\mathbb{R}^{>0}_{\text{fin}}$	die Eingangsverzögerung,
$ta : S \to {}^*\mathbb{R}^{>0}_{\text{fin}} \cup \{\omega\}$	die Zeitfortschrittsfunktion,
$\delta : Q \times X^+ \to S$	die Zustandsüberführungsfunktion,
X^+	Menge gleichzeitiger Ereignisse an den Eingangsports,
$\lambda : Q \times X^+ \to Y^+$	die Ausgabefunktion,
Y^+	Menge gleichzeitiger Ereignisse an den Ausgangsports,
$Q = \{(s, e) \mid s \in S,\ 0 < e \leq ta(s)\}$	der totale Zustand.

Der Kerngedanke der Erweiterung ist die Einführung von Eingangsverzögerungen und das Verbot von *transitory States*. Mit den eingeführten Modifikationen sollen die Defizite von RPDEVS bezüglich der Kausalität bei gleichzeitigen Ereignissen behoben werden. Damit die Änderungen nicht die reelle Simulationszeit verfälschen, wird diese mit hyperreellen Zahlen ${}^*\mathbb{R}$ abgebildet. Wie eingangs bereits erwähnt, wird aus der Menge der hyperreellen Zahlen die Zahl ε verwendet, die größer ist als Null, aber kleiner als jede positive reelle Zahl. Ebenfalls in dieser Menge enthalten ist die Zahl ω, die wie folgt definiert ist: $\omega := 1/\varepsilon$. Damit ist ω unendlich und wird durch *infinity* repräsentiert.

Die Simulationszeit wird durch zwei Zahlen a und b aus der Menge der reellen Zahlen beschrieben: $t = a + b\varepsilon$. Die Zahl a repräsentiert hier die reelle Simulationszeit und die Zahl b den infinitesimalen Anteil. Die Verwendung der hyperreellen Zahlen hat den Vorteil, dass diese Verzögerungen keinen Einfluss auf die reelle Simulationszeit haben. Dabei ist es egal, wie klein die Simulationszeitschritte sind, da die infinitesimalen Verzögerungen immer kleiner sind.

Die Eingangsverzögerungen werden durch τ beschrieben. τ gibt an, wie lange sich die Verarbeitung des letzten Eingangsereignisses verzögert: $tn = t + \tau$. Da innerhalb der Verzögerungszeit weitere Ereignisse eintreten können, wird die Verzögerung bei jedem Ereignis aktiv und die Zeit tn immer neu berechnet. Die Eingangsverzögerung τ muss größer sein als Null, damit eine Verzögerung vorhanden ist, aber sie muss kleiner sein als ω, da unendliche Verzögerungen dazu führen würden, dass die Eingangsereignisse niemals verarbeitet werden.

Die Dynamik eines atomaren Modells ist in Abbildung 9 schematisch dargestellt. In-

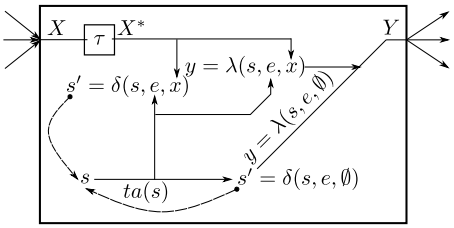


Abbildung 9: Dynamik eines atomaren NSA-DEVS

terne Ereignisse werden durch die Zeitfortschrittsfunktion $ta(s)$ eingeplant. Bei Eintritt eines internen Ereignisses, wird zuerst ein Ausgangsereignis durch die Funktion λ erzeugt. Diese berechnet auf Basis des Zustands s und der vergangenen Zeit in diesem Zustand e ein Ausgangsereignis $y \in Y$. Anschließend wird ein neuer Zustand s' durch die Funktion δ auf Basis des aktuellen Zustands s und der vergangenen Zeit e berechnet. Externe Ereignisse $x \in X$ werden gemäß der Eingangsverzögerung τ verzögert. Nach Abschluss der Verzögerung wird zuerst ein Ausgangsereignis durch die λ-Funktion erzeugt. Dies erfolgt auf Basis des aktuellen Zustands s, der vergangenen Zeit e und der verzögerten Eingangsereignisse $x \in X^*$. Anschließend wird ein neuer Zustand s' durch die δ-Funktion berechnet. Dieser wird auf Basis des aktuellen Zustandes s', die vergangene Zeit e und den verzögerten Eingangsereignisse $x \in X^*$ bestimmt. Da bei einem externen Ereignis die durch ein internes Ereignis eingeplante Lebenszeit $ta(s)$ des aktuellen Zustandes unterbrochen wird, gilt für die vergangene Zeit e die Bedingung $e \leq ta(s)$. Der Sonderfall, dass bei Eintritt eines externen Ereignisses $e = ta(s)$ ist, zeigt an, dass zeitgleich ein externes und internes Ereignis vorliegt. Dieser Fall wird auch als konfluentes Ereignis bezeichnet.

Eine wichtige Änderung gegenüber anderen DEVS-Formalismen betrifft die Verwendung von Ports. Ports wurden in den Arbeiten von Livny [55] eingeführt und auch bei den diskutierten DEVS-Formalismen im Kapitel 4 gibt es Erweiterungen mit Ports. Für den NSA-DEVS-Formalismus wird vom Autor dieser Arbeit eine neue Definition der Ports eingeführt, mit der die Notwendigkeit von Bags entfällt. Die Eingangs- und Ausgangsmengen sind definiert als

$$\begin{aligned} X &= \{(p, v) | p \in P_{in}, v \in X_p\} \\ Y &= \{(p, v) | p \in P_{out}, v \in Y_p\} \end{aligned}$$

dabei sind P_{in} und P_{out} Mengen von Namen der Eingangs- und Ausgangsports und X_p beziehungsweise Y_p repräsentieren die Wertemengen der Eingangsports beziehungsweise Ausgangsports p. Die Menge X^+ repräsentiert die Menge von gleichzeitigen Ereignissen an den unterschiedlichen Eingangsports:

$$X^+ := \left\{ \{(p_1, v_1), \ldots, (p_n, v_n)\} \,|\, n \in \mathbb{N}_0,\, p_i \in P_{in}, p_i \neq p_j \text{ for } i \neq j,\, v_i \in X_{p_i} \right\}$$

Für die Ausgangsports Y^+ gilt eine analoge Definition. Wie bereits weiter oben erwähnt, wurde für Ports bisher in der Literatur eine andere Definition verwendet. Dort ist es auch erlaubt, verschiedene Ausgangsports mit einem Eingangsport zu koppeln. Da dabei gleichzeitige Ereignisse auftreten können, werden bei diesen Definitionen Bags benötigt. In der Definition von NSA-DEVS ist es untersagt, mehrere Ausgangsports mit einem Eingangsport zu koppeln, dadurch entfällt die Notwendigkeit von Bags.

Ein gekoppeltes Modell wird im NSA-DEVS-Formalismus durch ein 7-Tupel definiert:

$$N =< X, Y, D, \{M_d\}, EIC, EOC, IC >,$$

dabei ist

X	die Menge aller Eingangsereignisse an allen Ports,
Y	die Menge aller Ausgangsereignisse an allen Ports,
D	die Menge der Komponentennamen,
$\{M_d\}$	die Beschreibung der Komponenten mit $d \in D$,
EIC	die Menge der externen Eingangskopplungen,
EOC	die Menge der externen Ausgangskopplungen,
IC	die Menge der internen Kopplungen.

Die Beschreibung eines gekoppelten Modells in NSA-DEVS entspricht im Wesentlichen der bei RPDEVS. Der einzige Unterschied besteht in der Spezifikation der Kopplungen, die von Classic DEVS übernommen wurde.

5.2 Simulatorbeschreibung

Das abstrakte Simulator-Konzept in NSA-DEVS ist identisch zu den anderen DEVS-Formalismen und wurde erstmals vom Autor in [44] veröffentlicht. Auch hier gibt es eine hierarchische Struktur bestehend aus Simulatoren, Koordinatoren und einem Root-Koordinator. Das Nachrichtenkonzept wurde ebenfalls von den früheren DEVS-Formalismen übernommen und umfasst die vier Nachrichtentypen i-Nachricht, *-Nachricht, y-Nachricht und x-Nachricht.

Der Root-Koordinator und der Koordinator sind identisch zu PDEVS (s. Abschn. 4.2) bis auf den Unterschied, dass alle Zeiten durch hyperreelle Zahlen abgebildet werden. Die Implementierung des Simulators zur Abarbeitung atomarer Modelle unterscheidet sich zu PDEVS und RPDEVS, da hier die Eingangsverzögerungen τ realisiert werden müssen. Demgemäß wird nachfolgend nur der abstrakte Simulator für atomare NSA-DEVS Modelle diskutiert.

Der abstrakte Simulator für atomare NSA-DEVS Modelle ist in Listing 5.1 dargestellt.

Listing 5.1: Abstrakter NSA-DEVS-Simulator.

```
1  properties:
2    parent
3    tl
4    tn
5    model                    (NSA-DEVS incl. τ and total state (s,e))
6    y
7    x*
8
9  when receive i-message(i,t) at time t
10   tl = t - e
11   tn = tl + ta(s)
12
13 when receive *-message(*,t) at time t
14   e = t - tl
15   y = λ(s,e,x*)
16   send y-message(y,t) to parent coordinator
17
18 when receive x-message(x,t) at time t
19   if x == ∅
20     e = t - tl
21     s = δ(s,e,x*)
22     x* = ∅
23     if ta(s) == ω
24       tn = ω
25     else if st(ta(s)) == 0
26       tn = t + ta(s)
27     else
28       tn = st(t + ta(s))
29     tl = t;
30   else
31     if not (x* == ∅)
32       add events from x to x*
33     else
34       x* = x
35     tn = t + τ
```

Um die Eingangsverzögerungen zu realisieren, wurde im Simulator eine neue Variable x^* eingeführt. In dieser Variablen werden Eingangsereignisse zwischengespeichert und erst nach der Verzögerung verarbeitet. Um dieses Verhalten umzusetzen, wird in der Bearbeitung der x-Nachricht unterschieden, ob in x ein Eingangsereignis vorliegt oder ob x die leere Menge beinhaltet (Zeile 19ff). Wenn x ein Eingangsereignis beinhaltet, dann wird dies in x^* gespeichert (Zeile 32) und der Zeitpunkt des nächsten Ereignisses berechnet: $tn = t + \tau$ (Zeile 35). Wenn x die leere Menge enthält, dann wird die Zustandsüberführungsfunktion $\delta(s, e, x^*)$ ausgeführt und der Inhalt der Variablen x^* wird gelöscht (Zeile 21f). Anschließend erfolgt die Zeitfortschaltung, in der drei Fälle unterschieden werden (Zeile 23ff).

1. Wenn die Funktion $ta()$ als Ergebnis ω liefert, dann wird der Zeitpunkt des nächsten Ereignisses tn gleich ω gesetzt.

2. Wenn der reelle Anteil der Zeitfortschaltung gleich Null ist, dann ergibt sich der Zeitpunkt des nächsten Ereignisses wie folgt: $tn = t + ta(s)$. Dabei handelt

es sich um eine infinitesimale Zeitfortschaltung, die nur Auswirkung auf den hyperreellen Anteil hat.

3. Anderenfalls muss der reelle Anteil der Zeitfortschaltungsfunktion größer sein als Null. Der Zeitpunkt des nächsten Ereignisses ergibt sich wie folgt: $tn = \mathrm{st}(t + ta(s))$. Das heißt, der hyperreelle Anteil wird auf Null gesetzt.

Die Bearbeitung der *-Nachricht unterscheidet sich kaum im Vergleich zu PDEVS. Lediglich der Aufruf der λ-Funktion wurde modifiziert, da hier nunmehr die gespeicherten Eingangsereignisse aus x^* übergeben werden.

Zur Implementierung muss der abstrakte Simulator in Listing 5.1 modifiziert werden. In Programmiersprachen und auf Rechnersystemen können keine hyperreellen Zahlen abgebildet werden. Aus diesem Grund gibt es eine Vereinfachung, in der die Zeit mit Hilfe eines Vektors, der aus zwei Elementen besteht, abgebildet wird. Das erste Element gibt den Anteil der reellen Zeit an und das zweite Element den infinitesimalen Anteil. Die beiden Elemente repräsentieren die Zeit in der Form $a + b\varepsilon$. Dabei ist a das erste Element des Zeitvektors $t(1)$ und b das zweite Element $t(2)$.

Für das Debugging von Modellen ist es hilfreich, infinitesimale Anteile durch kleine endliche Zeiten zu ersetzen. Dafür wird die Variable μ eingeführt, die einen reellen Faktor repräsentiert. Die Simulationszeit ist damit wie folgt definiert:

$$t' = \begin{cases} (t(1), t(2)) & \text{if } \mu = 0, \\ (t(1) + \mu\, t(2), 0) & \text{if } \mu > 0. \end{cases}$$

Wenn die Variable μ gleich Null ist, dann erfolgt keine Transformation und die Simulationszeit besteht aus einem finiten und einem infinitesimalen Anteil. Wenn sie allerdings größer ist als Null, dann wird der infinitesimale Anteil $t(2)$ mit der Variable μ multipliziert und auf die reelle Simulationszeit addiert. Damit gibt es keinen infinitesimalen Anteil mehr in der Simulationszeit und sie ist nur noch reell.

Diese Änderungen haben nur Auswirkung auf die Bearbeitung der x-Nachricht, welche in Listing 5.2 zu sehen ist.

Listing 5.2: Algorithmus zur Implementierung der x-Nachricht.

```
1  when receive x-message(x,t) at time t
2    if x == ∅
3      e = [t(1) - tl(1), t(2) - tl(2)]
4      s = δ(s,e,x*)
5      x* = ∅
6
7      tb = ta(s)
8      if tb == [0,0]
9        tb = [0, r]
10     if tb(1) == 0
11       if μ == 0
12         tn = [t(1), t(2) + tb(2)]
13       else
14         tn = [t(1) + μ*tb(2), 0]
15     else
```

```
16        tn = [t(1) + tb(1), 0]
17      tl = t;
18    else
19      if not (x* == ∅)
20        add events from x to x*
21      else
22        x* = x
23      if μ == 0
24        tn = [t(1) + tau(1), t(2) + tau(2)]
25      else
26        tn = [t(1) + tau(1) + μ*tau(2), 0]
```

Die x-Nachricht ist wieder unterteilt in einen Zweig für die Eingangsverzögerung und einen für die Zustandsüberführung mit der nachfolgenden Zeitfortschaltung. Im Fall einer Eingangsverzögerung werden die Eingangsereignisse in x^* zwischengespeichert und anschließend folgt die Zeitfortschaltung (Zeile 19-26). Hier wird geprüft, ob die Variable μ gleich Null ist. Wenn dies der Fall ist, ergibt sich die Zeitfortschaltung wie folgt: $tn = [t(1) + tau(1), t(2) + tau(2)]$. Wenn μ ungleich Null ist, dann berechnet sich die Zeitfortschaltung wie folgt: $tn = [t(1) + tau(1) + \mu \cdot tau(2), 0]$.

Wenn als Eingangsereignis die leere Menge empfangen wird, dann wird der Zweig mit der Zustandsüberführung ausgeführt (Zeile 3-17). Es werden die gespeicherten Eingangsereignisse x^* an δ übergeben und anschießend wird die Variable x^* geleert. Die Zeitfortschaltung ist in diesem Fall etwas aufwendiger im Vergleich zum Fall der Eingangsverzögerung. Als lokale Zeitvariable wird hier tb verwendet, die im ersten Schritt mit der durch $ta()$ bestimmten Zeitfortschaltung initialisiert wird. Für den Fall, dass die $ta()$-Funktion als Zeitfortschaltung den Wert $[0, 0]$ liefert, wird ein infinitesimaler Defaultwert r gesetzt, der τ_{def} mit $\tau_{def} = r\varepsilon$ darstellt. Standardmäßig ist der Wert von r gleich Eins, aber er kann verändert werden. Damit wird vom Simulator sichergestellt, dass keine Zeitfortschaltung gleich Null stattfindet. Wenn der reelle Anteil der Zeitfortschaltung gleich Null ist und die Variable μ auch Null beinhaltet, dann erfolgt nur eine infinitesimale Zeitfortschaltung, die sich wie folgt ergibt: $tn = [t(1), t(2) + tb(2)]$. Ist der reelle Anteil gleich Null und die Variable μ ungleich Null, dann erfolgt eine reelle Zeitfortschaltung mit dem Faktor μ: $tn = [t(1)+\mu \cdot tb(2), 0]$. Bei einer reellen Zeitfortschaltung wird der infinitesimale Anteil auf Null gesetzt. Damit ergibt sich die Zeitfortschaltung wie folgt: $tn = [t(1)+tb(1), 0]$.

Der Autor dieser Arbeit hat alle im Kapitel 4 diskutierten DEVS Formalismen in der SCE MATLAB implementiert (Classic-DEVSforMATLAB [11], PDEVSforMATLAB [13] und RPDEVSforMATLAB [14]). Darauf aufbauend wurde der NSA-DEVS-Formalismus (NSA-DEVSforMATLAB [44], [12]) implementiert und anhand von vier unterschiedlich komplexen Beispielmodellen getestet [45].

5.3 Visuelle Modellierung mit DEVS-Diagrammen

Die Modellierung von atomaren Modellen geschieht üblicherweise in Mengennotation. Für Ingenieure ist diese Notation oft nicht intuitiv. Hier werden Diagramme bevorzugt eingesetzt, da diese oft anschaulicher sind. Für atomare Classic DEVS

Modelle hat Song [87] eine grafische Repräsentation eingeführt, die in dieser Arbeit als DEVS-Diagramm bezeichnet wird. Das DEVS-Diagramm wurde durch Freymann [30] um neue Elemente erweitert. Pawletta [71] passte diese Diagrammtechnik für die Modellierung von atomaren NSA-DEVS-Modellen an.

Abbildung 10 zeigt das Grundgerüst eines DEVS-Diagramms. Ein atomares Mo-

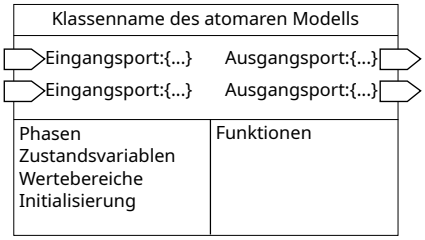
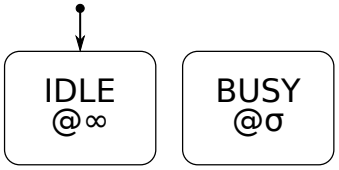
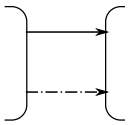
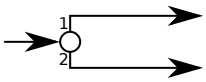


Abbildung 10: Rahmen eines DEVS-Diagramms zur Darstellung eines atomaren Modells nach Song [87]

dell kann als Inkarnation einer Modellklasse gesehen werden. In der oberen Box des Diagramms wird der Name der atomaren Modellklasse spezifiziert. In der mittleren Box werden die Eingangs- und Ausgangsports sowie ein die Dynamik beschreibendes Zustandsdiagramm angegeben. Letzteres ist in Abbildung 10 noch nicht dargestellt. Bei den Portdefinitionen werden jeweils der Portname sowie der zulässige Ereignistyp und dessen Wertebereich angegeben. Ein Port kann immer nur einen Ereignistyp verarbeiten. Die Kombination von Ereignistyp und Ereigniswert wird beim DEVS-Diagramm als Message bezeichnet. Die Ausgangsports befinden sich auf der rechten Seite. Auch hier werden die jeweiligen Portnamen und der dazugehörige Wertebereich angegeben. In der unteren linken Box werden alle Zustandsvariablen mit dem dazugehörigen Wertebereich und einem Initialisierungswert aufgelistet. In der unteren rechten Box werden nach Freymann [30] Aktivitäten definiert, die im Fall von NSA-DEVS aber nicht benötigt werden. Dieser Platz kann für Funktionsdefinitionen, die zur Vereinfachung der Dynamikbeschreibung im Zustandsdiagramm benutzt werden, verwendet werden.

Die Beschreibung der Dynamik erfolgt über ein Zustandsdiagramm, welches auch als Phasendiagramm bezeichnet wird. In Tabelle 1 sind die Elemente zur Spezifikation von DEVS-Diagrammen für NSA-DEVS zusammengefasst.

Tabelle 1: Elemente der DEVS-Diagramme für NSA-DEVS

Element	Beschreibung
	**Eingangs- bzw. Ausgangsport** Portname mit Definition des Ereignistyps und dessen Wertebereichs (kurz msgType).
	**Phasen beziehungsweise Hauptzustände** Darstellung einer Phase mit der dazugehörigen Lebenszeit, definiert mit dem @-Operator.
	**Transition** Die Pfeile beschreiben eine Transition zwischen Phasen. Die durchgezogene Linie gibt dabei eine Transition aufgrund eines externen Ereignisses an. Die Strich-Punkt-Linie beschreibt eine Transition aufgrund eines internen oder eines konfluenten Ereignisses.
	**Condition-Junction** Die Condition-Junction spezifiziert eine Fallunterscheidung, die an Prioritäten und Bedingungen geknüpft ist. Sie dient der Darstellung von Spezifikationen in der Zustandsüberführungsfunktion δ sowie der Ausgabefunktion λ.
$inportName?msg$	**Definition eines Phasenübergangs aufgrund eines bestimmten externen Ereignisses.** Externes Ereignis am Port *Name* als Nachricht *msg* in Form von Ereignistyp und Ereigniswert.
$inports?$	**Definition eines Phasenübergangs aufgrund irgendeines externen Ereignisses.** Der Phasenübergang erfolgt, wenn ein beliebiges zulässiges externes Ereignis vorliegt. Dabei ist nicht relevant, welches Ereignis oder an welchen Ports ein Ereignis anliegt.
$@[guard]$	**Transitionsbedingung** Definition einer Transitionsbedingung (*guard*) mit dem Ergebnis *true* oder *false*.

$/\{outportName!msg, \ldots,$ $stateVar1 = value, \ldots\}$	**Transitionsaktion.** Definition von Transitionsaktionen. Hier werden Ausgangsereignisse in Form von Nachrichten *msg* für Ports definiert. Zusätzlich können Zustandsvariablen verändert werden. Alle Aktionen werden durch ein Komma separiert.
#Kommentar	**Kommentar** Dient der Dokumentation.

Die Phasen bilden eine Teilmenge der Zustandsmenge. Jede Phase besitzt eine Lebenszeit, die mit dem @-Operator definiert wird und aus der Zeitfortschrittsfunktion $ta(s)$ folgt. Zwischen den Phasen werden durch Linien mit Pfeilen Zustandstransitionen beschrieben. In den DEVS-Diagrammen wird zwischen zwei Linientypen unterschieden:

1. Durchgezogene Linie: Die durchgezogene Linie beschreibt eine Transition aufgrund eines externen Ereignisses.
2. Strich-Punkt-Linie: Die Strich-Punkt-Linie beschreibt eine Transition aufgrund eines internen Ereignisses beziehungsweise eines konfluenten Ereignisses. Letzteres bedeutet, das gleichzeitig ein internes und ein externes Ereignis vorliegt.

Damit Fallunterscheidungen in Transitionen dargestellt werden können, wurde schon von Freymann [30] die *Condition-Junction* eingeführt. Mit Fallunterscheidungen kann oft eine bessere Übersichtlichkeit der Diagramme erreicht werden. Mit sogenannten *Guards* können Bedingungen für Transitionen beschrieben werden. Die Bedingungen werden auf Basis von Zustandswerten oder externen Ereignissen definiert. Ein Guard wird mit dem @-Operator eingeleitet, dem in rechteckigen Klammern ein Bedingungsausdruck folgt. Eine Phasentransition kann mit Aktionen kombiniert werden. Unter Aktionen werden Änderungen von Zustandswerten und die Generierung von Ausgangsereignissen verstanden. Aktionen werden mit dem /-Operator eingeleitet, dem in geschweiften Klammern eine Liste mit Anweisungen folgt, die durch Kommata separiert werden.

Weiterhin können Transitionen durch externe Ereignisse ausgelöst werden. Hier werden zwei Fälle unterschieden. Die Spezifikation einer Transition aufgrund eines bestimmten Ereignisses an einem bestimmten Eingangsport erfolgt mit der Notation *inportName?msgType*. Soll eine Transition in Folge eines beliebigen Ereignisses an einem beliebigen Eingangsport ausgelöst werden, so wird dies mit der Notation *inports?* definiert. Der zweite Fall unterstützt auch die Behandlung simultaner Eingangsereignisse, indem nachfolgend unter Nutzung von Guards und Condition-Junctions Fallunterscheidungen spezifiziert werden.

Zustandsänderungen gewöhnlicher Zustandsvariablen, hierzu zählen alle Variablen außer der Zustandsvariablen *Phase*, und Ausgangsereignisse werden in Form von Transitionsaktionen definiert. Transitionsaktionen werden syntaktisch mit $/\{\ldots\}$

beschrieben. In den Klammern kann eine Liste von Anweisungen definiert werden, die per Kommata separiert werden. Die Spezifikation von Ausgangsereignissen entspricht der Syntax von externen Ereignissen, wobei das ? durch ein ! ersetzt wird, also *outportName!msg*.

Freymann [30] führte zusätzlich noch Prioritäten ein, um eine Abarbeitungsreihenfolge zu definieren. Dabei wird jedem Transitionspfad eine Nummer größer gleich 1 zugewiesen, wobei die Nummer Eins die höchste Priorität repräsentiert. Außerdem führte Freymann [30] Kommentare ein, welche mit # gekennzeichnet werden.

5.4 Zusammenfassung

In diesem Kapitel wurden die Modellspezifikation und das Simulatorkonzept des NSA-DEVS-Formalismus eingeführt. Zusätzlich wurde die visuelle Modellbeschreibung mit DEVS-Diagrammen auf den NSA-DEVS-Formalismus angepasst.

Es ist zu konstatieren, dass die Grundstruktur von NSA-DEVS den DEVS-Formalismen aus Kapitel 4 entspricht. NSA-DEVS ist eine Weiterentwicklung im Sinne von RPDEVS mit der Zielstellung, die Abbildung und Abarbeitung von Mealy-Verhalten in DEVS-Modellen zu verbessern. Bei der Modellspezifikation baut NSA-DEVS direkt auf RPDEVS auf, indem die Dynamikbeschreibung mit nur einer Zustandsüberführungsfunktion erfolgt und Eingangsereignisse unmittelbar das Ausgangsverhalten beeinflussen können. Zusätzlich führt NSA-DEVS infinitesimale Eingangsverzögerungen unter Verwendung einer hyperreellen Zeitbasis ein. Dadurch wird die Kausalität von Ereignisfolgen auf Modellebene sichergestellt, woraus deutlich vereinfachte Abarbeitungsalgorithmen auf Simulatorebene folgen. Aufwendige Ereignisiterationen zur Laufzeit wie bei RPDEVS entfallen. Ein Vergleich der Simulatorkonzepte von RPDEVS und NSA-DEVS anhand der abstrakten Simulatoren zeigt, dass der NSA-DEVS-Formalismus zu wesentlich einfacheren Abarbeitungsalgorithmen führt. Die Komplexität der Abarbeitungsalgorithmen von NSA-DEVS entspricht damit wieder der von PDEVS.

Der Ansatz des NSA-DEVS-Formalismus hat Ähnlichkeiten mit der *Super-Dense-Time* [56, 67] oder den *Hidden-Time-Stamp-Fields* [33]. Diese Ansätze dienen der Sortierung und Sicherstellung der Abarbeitungsreihenfolge von verteilten Simulatoren bei gleichzeitigen Ereignissen und verfolgen damit eine andere Zielstellung.

Im nächsten Kapitel soll überprüft werden, ob sich der NSA-DEVS-Formalismus auch für komplexe Modellstrukturen mit sehr vielen Ereignissen eignet. Es wird außerdem untersucht, ob sich gemischt diskret-ereignisorientierte und kontinuierliche (hybride) Problemstellungen mit dem Quantized State System (QSS)-Ansatz nach Kofmann [15] unter Nutzung von NSA-DEVS lösen lassen.

6 Anwendungsstudie

In diesem Kapitel wird eine Anwendungsstudie eingeführt, um die Eignung des NSA-DEVS-Formalismus in Kombination mit Parallelverarbeitung zur Lösung realistischer Problemstellungen der M&S zu untersuchen. Die Anwendungsstudie enthält ein Beispiel aus dem Bereich der Produktionstechnik welches auf Arbeiten von Larek und Schmidt [51, 82] basiert. Anhand von ausgewählten atomaren Modellen wird die grundlegende Modellierung unter Verwendung von DEVS-Diagrammen gemäß Abschnitt 5.3 aufgezeigt. Die konkrete Implementierung der Modelle und deren Ausführung erfolgt mit der Simulationsumgebung NSA-DEVSforMATLAB [12] auf einem HPC-System. Teilaspekte der nachfolgenden Untersuchungen wurden bereits in [46] veröffentlicht.

Wie von Larek, Schmidt, Hagendorf et al. [51, 82, 39, 52] gezeigt, sind realistische Problemstellungen im Bereich der Produktionstechnik oft nicht mit manuell ausgeführten sequentiellen Simulationsläufen untersuchbar. Meist besteht die Zielstellung darin, aus unterschiedlichen Produktionsvarianten die beste auszuwählen und für diese dann optimale Parametereinstellungen zu finden. Aus Sicht der M&S handelt es sich dabei um eine simulationsbasierte Struktur- und Parameteroptimierung [38]. Aufgrund der Vielzahl zu untersuchender Modellstrukturen und Modellparameter müssen die Simulationsexperimente systematisch geplant, automatisiert und mit weiteren numerischen Verfahren kombiniert werden.

Schmidt [82] führt zur Klassifikation simulationsbasierter Experimente die drei Phasen *frühzeitig, mittelfristig* und *langfristig* ein. Die Phasen unterscheiden sich im Aufwand und verfolgen unterschiedliche Ziele. Während die ersten zwei Phasen vorrangig Parameteranalysen umfassen, um die Anzahl der zu untersuchenden Modellstrukturen und Parameter zu reduzieren, gilt es in der dritten Phase, die best möglichen Lösungen per simulationsbasierter Optimierung zu finden.

Neben den Phasen klassifiziert Schmidt Simulationsexperimente nach ihrer Komplexität in *einfach*, *komplex* und *hochkomplex*. Sieht man von einfachen Explorationen ab, so sind bereits die typischen Parameteranalysen der frühen und mittleren Phase komplex. Struktur- und Parameteroptimierungen in der langfristigen Phase sind in der Regel hochkomplex. Zur Automatisierung derartiger Experimente entwickelt er einen Konzeptrahmen und implementiert diesen in Software. Um den Rechenaufwand zu bewältigen, nutzt er auch Methoden der Parallelverarbeitung, ist dabei aber auf Standardhardware mit wenigen Cores beschränkt. Seine praktischen Experimente und seine darauf basierenden Hochrechnungen bezüglich der voraussichtlich benötigten Rechenzeit zeigen, dass die Problemstellung dieser Anwendungsstudie nach [51] mit dieser Technologie nicht lösbar ist.

In den nachfolgenden Abschnitten wird die Anwendungsstudie nach Larek und Schmidt [51, 82] nochmals aufgegriffen und daran ein erweiterter Lösungsansatz für komplexe und hochkomplexe Experimente unter Nutzung eines HPC-Systems und von NSA-DEVS vorgestellt. Bei der Modellierung der Fertigungseinrichtungen wird der Detailierungsgrad mit dem Dreischichtenmodell (*Physik, Prozess und Materialfluss*) gegenüber Schmidt und Larek erhöht. Mit der Physikmodellschicht werden kontinuierlich ablaufende Prozesse exakt abgebildet. Schmidts Konzeptrahmen zur automatischen Experimentausführung wird für HPC-Systeme angepasst. Praktisch umgesetzt und analysiert wird ein simulationsbasiertes Screening, das heißt ein komplexes Experiment der frühzeitigen Phase. Auf Basis der Untersuchungsergebnisse des Screenings wird analog zu Schmidt [82] das Laufzeitverhalten einer komplexen Sensitivitätsanalyse und einer hochkomplexen Struktur- und Parameteroptimierung abgeschätzt, nunmehr aber unter Verwendung einer HPC-Plattform mittlerer Größe.

6.1 Einführung der Problemstellung für die Anwendungsstudie

Es wird eine fertigungstechnische Prozesskette gemäß Abbildung 11 untersucht, wie sie von Larek [51] und Schmidt [82] bereits eingeführt wurde. Die Prozesskette um-

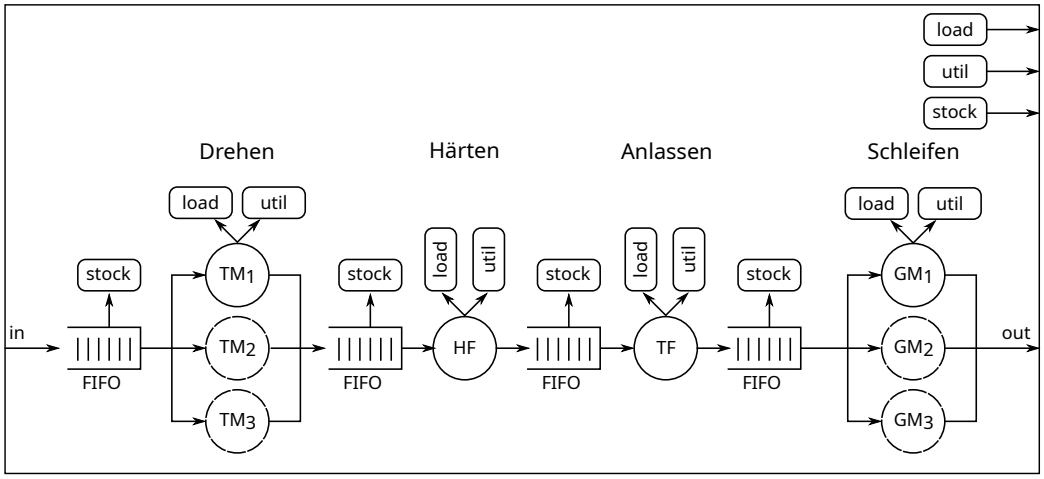


Abbildung 11: Struktur und Bewertungsgrößen der Prozesskette nach [51, 82]

fasst die Arbeitsschritte: Drehen, Volumenhärten, Anlassen und Schleifen, welche durch Puffer entkoppelt sind. In die Prozesskette werden Rohteile eingeschleust und es erfolgt im ersten Prozessschritt eine Drehbearbeitung (Turning Machine, TM). Danach folgen die thermischen Behandlungen Volumenhärten (Hardening Furnace, HF) und Anlassen (Tempering Furnace, TF) und als letzter Prozessschritt eine Schleifbearbeitung (Grinding Machine, GM). Die Puffer vor den jeweiligen Arbeitseinrichtungen arbeiten nach einer First In - First Out (FIFO) Policy und haben eine

maximale Kapazität von 400 Werkstücken. Nachfolgend werden die Puffer auch als Queues bezeichnet.

Nach Schmidt [82] gilt es, für die Prozesskette eine optimale Struktur und Parameterkonfiguration zu bestimmen. Die Strukturvarianten ergeben sich aus der möglichen Variation der Anzahl eingesetzter Dreh- und Schleifmaschinen sowie des Einsatzes unterschiedlicher Ofentypen für die Wärmebehandlungsoperationen. Maximal können bis zu drei parallel arbeitende Drehmaschinen und ebenfalls bis zu drei Schleifmaschinen eingesetzt werden. Bei den Öfen stehen jeweils drei verschiedene Typen zur Auswahl, die sich in ihrer Kapazität, Größe und Leistung unterscheiden. Die Parameter der Ofentypen sind in Tabelle 2 aufgelistet. Aus der möglichen Variation der Fertigungseinrichtungen ergeben sich 81 Strukturvarianten.

Tabelle 2: Kenngrößen der Ofentypen

	Typ 1	**Typ 2**	**Typ 3**
Heizleistung [W]	6.400	15.000	20.000
Kapazität [Stk]	20	40	70
Wärmekapazität [J/K]	6.000	10.000	12.000
Verlustleistung [W/K]	6	8	10

Die Dreh- und Schleifmaschinen können gemäß den Parameterangaben in den Tabellen 3 und 4 unterschiedlich konfiguriert werden. Die Parameter haben Auswirkun-

Tabelle 3: Parameter der Drehmaschinen

	Minimum	Maximum
$ap[mm]$	0,5	1,5
$f[mm/U]$	0,1	0,5
$vc[m/min]$	150	350

Tabelle 4: Parameter der Schleifmaschinen

	Minimum	**Maximum**
$vfr1[mm/min]$	0,48	1,86
$vfr2[mm/min]$	0,19	0,53
$vfr3[mm/min]$	0,03	0,16
$vc[m/s]$	30	50

gen auf die Arbeitsgeschwindigkeit, aber auch auf die benötigte Leistung. Bei den

Drehmaschinen können die Schnitttiefe (ap), der Vorschub (f) und die Schnittgeschwindigkeit (vc) und bei den Schleifmaschinen drei Vorschubgrößen ($vfr1$, $vfr2$, $vfr3$) sowie die Schnittgeschwindigkeit (vc) variiert werden.

Die Bewertungsgrößen der Prozesskette sind in Tabelle 5 aufgeführt.

Tabelle 5: Bewertungsgrößen der Prozesskette

$Espec$	pro Bauteil benötigte Energie
$loadPeak$	maximal benötigte elektrische Leistung
$procTime$	Produktionszeit pro Bauteil
$pcStock$	Summe der Maximalbelegungen aller Queues
$thrput$	Anzahl der fertiggestellten Bauteile
$pcUtil$	Summe der Auslastungen der Fertigungseinrichtungen

Das Entwurfsziel nach Schmidt [82] ist es, die benötigte Energie pro Bauteil ($Espec$), die maximal benötigte elektrische Leistung ($loadPeak$), die Produktionszeit pro Bauteil ($procTime$) und die maximale Belegung der Queues ($pcStock$) zu minimieren. Gleichzeitig soll die Anzahl fertiggestellter Bauteile ($thrput$) und die Auslastung der Auslastung der Fertigungseinheiten ($pcUtil$) maximiert werden. Die gewichtete Zusammenfassung aller Bewertungsgrößen wird durch folgende Gütefunktion dargestellt:

$$f(\Theta) = \frac{1}{6}Espec + \frac{1}{6}loadPeak + \frac{1}{6}procTime + \frac{1}{6}pcStock - \frac{1}{6}thrput - \frac{1}{6}pcUtil \quad (6.1)$$

6.2 Modellierung der Produktionskette für die Anwendungsstudie

In diesem Abschnitt wird die Modellierung der Produktionskette betrachtet. Da eine Vielzahl von Modellvarianten zu untersuchen ist, wird zuerst auf die grundlegende Strukturierung des gesamten Modells im Kontext mit einem *Experimental Frame* (EF) eingegangen. Anschließend wird die Modellierung der Fertigungseinrichtungen, hier auch als Maschinenmodelle bezeichnet, betrachtet. Da für die Prozesskette eine detaillierte energetische Bewertung erfolgen soll, müssen die Fertigungseinrichtungen bis auf die Ebene der *Maschinenphysik* modelliert werden. Hierzu wird ein Schichtenmodell eingeführt. Anschließend wird die detaillierte Modellierung eines Maschinenmodells am Beispiel eines Ofens für Wärmebehandlungsoperationen aufgezeigt.

6.2.1 Struktur des Gesamtmodells

In diesem Abschnitt wird die Struktur des Simulationsmodells vorgestellt. In Vorbereitung auf die simulationsbasierten Experimente wird das Simulationsmodell äquivalent zu Schmidt [82] gemäß des Konzepts des *Experimental Frame* (EF) [106,

S. 27-29] strukturiert. Das Simulationsmodell besteht demnach aus einem EF und einem Modell, welches als *Model Under Study* (MUS) bezeichnet wird. Im vorliegenden Fall bildet die zu untersuchende Prozesskette das MUS. Der EF interagiert mit dem MUS. Der EF setzt Einflussgrößen und analysiert Ausgangsgrößen des MUS. Abbildung 12 zeigt das allgemeine Konzept in Anlehnung an [82].

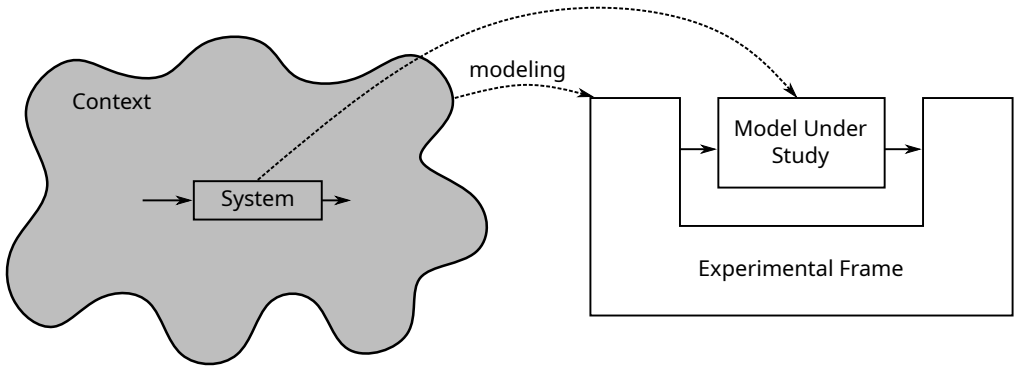


Abbildung 12: Konzept des EF in Anlehnung an [82]

Der EF besteht in der Regel aus den drei Komponenten: *Generator*, *Transducer* und *Acceptor*. Die Komponenten und der Aufbau des EF sind in Abbildung 13 dargestellt. Der Generator hat die Funktion, das MUS mit Eingangsgrößen anzuregen

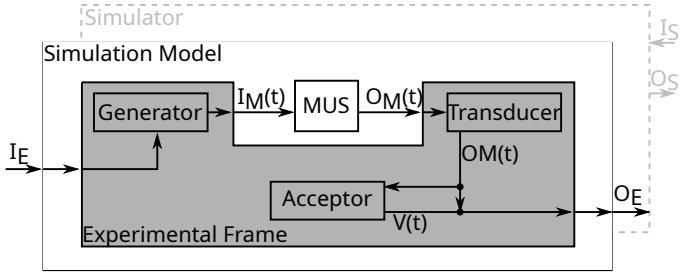


Abbildung 13: Aufbau des EF nach [107]

beziehungsweise Eingangsgrößen zu definieren. Der Transducer nimmt die Ausgangsgrößen des MUS auf und übersetzt beziehungsweise rechnet diese in Kenngrößen um, die in der späteren Auswertung benötigt werden. Der Acceptor dient der Überwachung der Simulation und kann zum Beispiel bei einer Überschreitung von Grenzwerten die Simulation vorzeitig abbrechen. Die Besonderheit an diesem Konzept ist, dass alle Komponenten des EF Modelle sind. Das MUS kann für verschiedene Experimente genutzt werden, weil es keine spezifischen Experimentabhängigkeiten enthält.

Für die Anwendungsstudie dieser Arbeit wird der Acceptor nicht benötigt, so dass hier der EF nur aus einem Generator und einem Transducer besteht. Nachfolgend wird der prinzipielle Aufbau des Generators und des MUS sowie des Transducers aufgezeigt.

Generator

Die Funktion des Generators im EF-Konzept besteht darin, die in einem Experiment zu untersuchenden Größen I_E in Eingabetrajektorien $I_M(t)$ für das MUS zu transformieren. Die Aufgabenstellung der hier präsentierten Anwendungsstudie orientiert sich an Larek [51] und Schmidt [82].

Danach enthält I_E drei Vorgaben für die in der Prozesskette enthaltenen Drehmaschinen (*ap*, *f*, *vc*; vgl. Abschn. 6.1, Tab. 3) und vier Vorgaben für die enthaltenen Schleifmaschinen (*vfr1*, *vfr2*, *vfr3*, *vc*; vgl. Abschn. 6.1, Tab. 4). I_E ist damit ein 7-Tupel.

Gemäß Larek [51] und Schmidt [82] muss für ein konkretes Experiment noch die achte Größe max_fifo spezifiziert werden. Diese legt die Größe der Puffer vor den Bearbeitungsstationen Drehen, Härten, Anlassen und Schleifen der Prozesskette fest. Diese Größe ist kein Bestandteil von I_E, aber von $I_M(t)$. Der vom Generator an das MUS gesendete Input ist damit ein 8-Tupel.

Für die hier präsentierte Anwendungsstudie wurde der Generator als gekoppeltes Modell bestehend aus 8 NSA-DEVS Atomic-Modellen implementiert.

MUS

Die Struktur der obersten Hierarchieebene des MUS zeigt Abbildung 14. Die Prozesskette (vgl. Abschn. 6.1, Abb. 11) mit den Fertigungseinrichtungen Drehen, Härten, Anlassen und Schleifen sowie den zugehörigen Puffern (Queues) bildet das zu untersuchende System.

Das MUS erhält die Eingabegrößen I_M vom vorgelagerten Generator des EF. Im EF-Konzept kann es sich dabei um Eingabetrajektorien $I_M(t)$ handeln. In der hier betrachteten Anwendungsstudie wird diese Funktionalität des EF jedoch nicht benötigt. Bei allen Komponenten von I_M handelt es sich hier um Konstanten zur Parametrierung des MUS.

In einer Fertigungsprozesskette sind die Werkstücke von zentraler Bedeutung. Diese werden im MUS als Entitäten modelliert. Diesbezüglich wurde das MUS als halbgeschlossenes System realisiert. Das heißt, die Werkstücke werden als Rohteile in das MUS nicht über I_M vom EF-Generator eingeschleust, sondern über einen Entitäten-Generator (EG) innerhalb des MUS erzeugt. Am Ende der Fertigungsprozesskette stellen die Entitäten Fertigteile dar. Diese werden als MUS-Output an den EF-Transducer weitergeleitet.

Die Fertigungsstationen Drehen, Härten, Anlassen und Schleifen sind im MUS jeweils als gekoppelte Modelle realisiert. Beim Drehen und Schleifen sind die vorgelagerten Puffer jeweils in das Teilmodell der Fertigungsstation integriert. Die Puffer

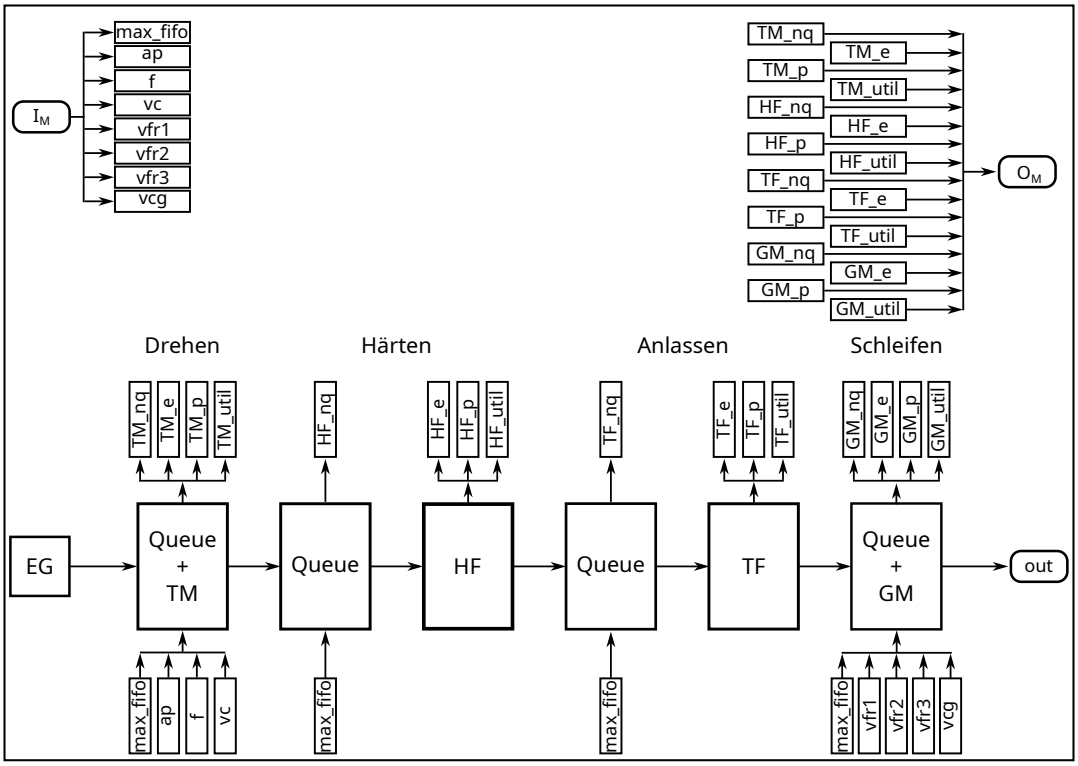


Abbildung 14: Struktur des MUS

vor dem Härten und Anlassen sind dagegen auf der obersten Ebene des MUS als gekoppeltes Modell realisiert.

Transducer

Der Transducer hat im EF-Konzept die Funktion die Ergebnisgrößen $O_M(t)$ des MUS aufzunehmen und daraus die Ergebnisgrößen O_E eines Experiments zu bestimmen.

In der hier besprochenen Anwendungsstudie ist $O_M(t)$ ein 19-Tupel, welches teilweise aus Trajektorien besteht, aber auch einzelne Werte enthält. $O_M(t)$ beinhaltet zunächst die Informationen der Batch-Belegungen beim Härten und Anlassen. Darüber hinaus werden jeweils die benötigten elektrischen Leistungen aller Fertigungsschritte (Drehen, Härten, Anlassen und Schleifen) geliefert. Ebenso werden die Trajektorien der Auslastungen, der Pufferbelegungen und des Energieverbrauchs aller vier Fertigungsstationen übermittelt. Zusätzlich schleust das MUS über $O_M(t)$ alle Fertigteile in Form von Entitäten an den Transducer.

Die Verarbeitung der über $O_M(t)$ vom MUS gelieferten Informationen wird im Transducer über zwei gekoppelte Modelle *KPI_Calculation* und *TargetFunction* realisiert. Das Modell KPI_Calculation bestimmt aus $O_M(t)$ den Key Performance Indicator (KPI) bestehend aus *Espec*, *loadPeak*, *procTime*, *pcStock*, *thrput* und *pcUtil* (vgl. Abschn. 6.1). Unter Verwendung dieser 6 Werte des KPI wird im zweiten Modell der Gütefunktionalwert $f(\Theta)$ berechnet.

An die überlagerte Experimentsteuerung gibt der Transducer die Ergebnisse in Form eines 7-Tupels bestehend aus dem KPI und dem Gütefunktionalwert als O_E aus.

6.2.2 Struktur der Maschinenmodelle

Die Struktur der Maschinenmodelle orientiert sich an der mehrschichtigen Architektur nach Pawletta [72]. Gemäß Abbildung 15 werden drei Modellschichten unterschieden:

1. Materialflussmodell: Abbildung diskret-ereignisorientierter Operationen des Produktionsprozesses,
2. Prozesssteuerungsmodell: Abbildung von Steuerungsoperationen,
3. Physikmodell: Abbildung von Maschinenoperationen auf Basis von Differenzialgleichungen oder von Messdaten.

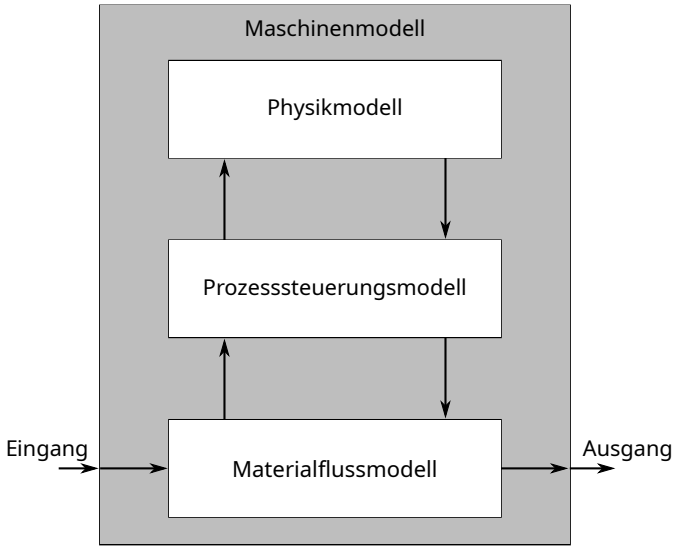


Abbildung 15: Struktur der Maschinenmodelle nach [72]

Die Dreischichtenstruktur bietet eine hohe Variabilität. So können zum Beispiel unterschiedliche Physikmodelle oder Prozesssteuerungsmodelle der Maschine erstellt und an einem identischen Materialflussmodell getestet werden. Je nach Einsatzzweck des Maschinenmodells, zum Beispiel offline oder online Simulation, können die Modellschichten einen unterschiedlichen Detaillierungsgrad besitzen.

6.2.3 Modellierung eines Ofens

Am Beispiel der Fertigungseinrichtung eines Ofens für die Arbeitsoperationen Härten und Anlassen soll in diesem Abschnitt der Aufbau eines Maschinenmodells aufgezeigt werden. Wie im vorangegangenen Abschnitt diskutiert, ist das in Abbildung 16 dargestellte Maschinenmodell *furnace* in ein Physikmodell (PM), ein Prozesssteuerungsmodell (CM) und ein Materialflussmodell (MF) gegliedert.

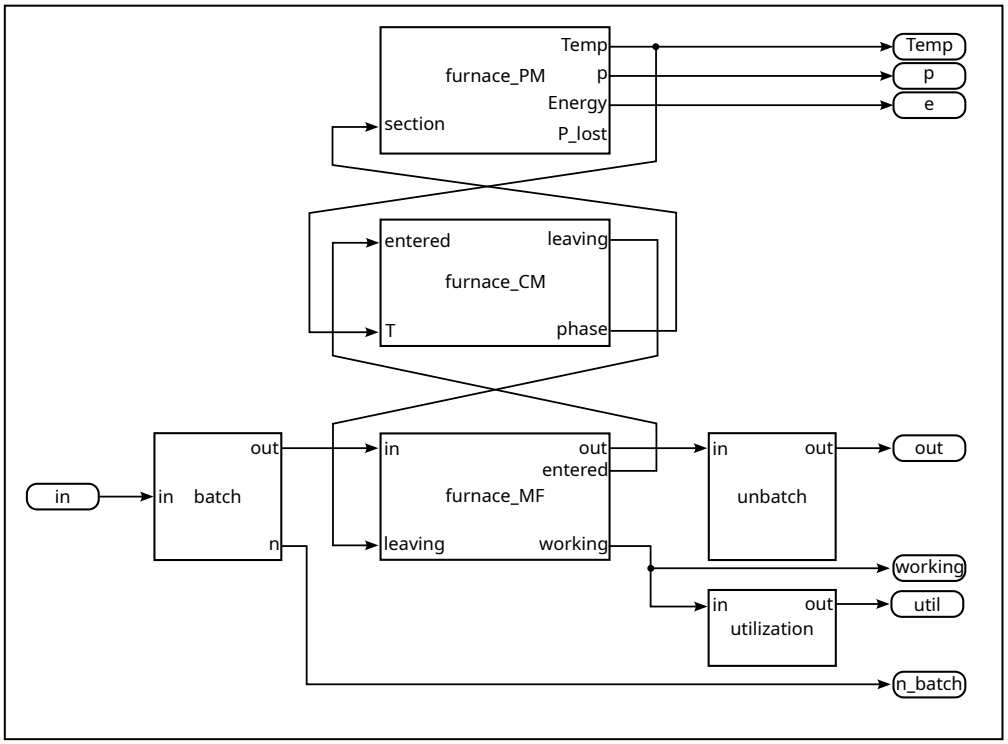


Abbildung 16: Aufbau des Ofenmodells *furnace*

Physikmodell (PM)

Das PM des Ofens besteht, wie in Abbildung 17 gezeigt, aus den vier gekoppelten Modellen *unload*, *heating*, *furnace* sowie *power_calc* und einem atomaren Modell *heat_controller*.

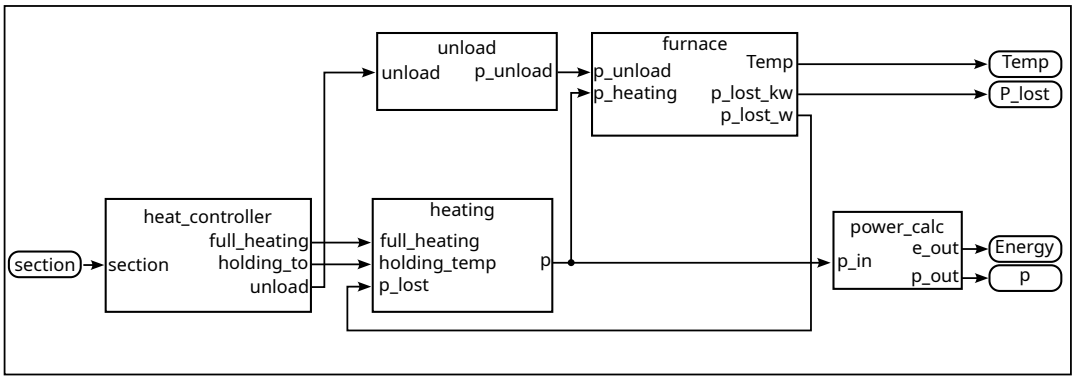


Abbildung 17: Aufbau des Physikmodells (PM)

Das atomare Modell *heat_controller* übersetzt Eingangsereignisse vom Prozesssteuerungsmodell (CM) bezüglich auszuführender Arbeitsoperationen in physikalische Leistungsberechnungen. Für den Ofen werden folgende vier Operationen unterschieden:

1. *load*: Der Ofen wird gerade beladen.
2. *heatup*: Der Ofen wird mit maximaler Leistung aufgeheizt.
3. *hold*: Der Ofen hält die Temperatur konstant.
4. *unload*: Der Ofen wird entladen.

Die Arbeitsoperationen sind im PM mit drei Leistungsberechnungsfällen abgebildet, welche durch den *heat_controller* per Ausgangsereignis aktiviert werden.

1. *full_heating*: Die maximale Heizleistung wird aktiviert.
2. *holding_to*: Die aktuelle Temperatur wird gehalten.
3. *unload*: Die Leistung für das Entladen wird aktiviert.

Der Leistungsberechnungsfall *unload* gilt für die Arbeitsoperationen *load* und *unload*, bei denen durch das Öffnen und Be-/Entladen des Ofens Energie entzogen wird. Die im Teilmodell *unload* berechnete Leistung *p_unload* spiegelt diese Verlustleistung wieder. Der Leistungsberechnungsfall *full_heating* gilt für die Arbeitsoperation *heatup*. Die für diesen Fall im Teilmodell *heating* berechnete Leistung p stellt die maximal zuführbare elektrische Heizleistung dar. Der Leistungsberechnungsfall *holding_to* gilt für die Arbeitsoperation *hold*. Die für diesen Fall im Teilmodell *heating* berechnete Leistung p entspricht der Verlustleistung des Ofens auf Grund des Wärmeaustausches über das Ofengehäuse mit der Umgebung.

Das thermische Verhalten des Ofens folgt aus den zuvor beschriebenen Leistungsanteilen und wird im Teilmodell *furnace* durch folgende Differenzialgleichung (DGL) beschrieben:

$$\frac{\mathrm{d}T}{\mathrm{d}t} = \frac{1}{C_o}[-(T - T_e)k_A + P_{heating} - P_{unload}]. \tag{6.2}$$

Abbildung 18 zeigt die Umsetzung der DGL mit atomaren NSA-DEVS Modellen. Das atomare Modell *S0_Temp* ist eine Inkarnation der Modellklasse *hIntegrator*, die DGLs nach dem QSS-Verfahren löst. Ausgangsgrößen des Teilmodells *furnace* sind

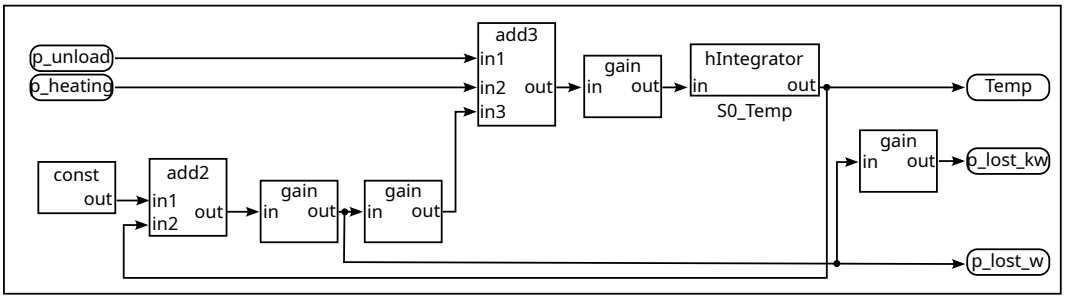


Abbildung 18: Teilmodell *furnace*

die aktuelle Ofentemperatur und die Verlustleistung aufgrund von Wärmeaustausch über das Gehäuse mit der Umgebung in Kilowatt sowie in Watt.

Das Teilmodell *power_calc* berechnet aus der aktuell zugeführten elektrischen Leistung p in Watt die Leistung in Kilowatt und die Energie in Kilowattstunden. Für die Berechnung der Energie wird ebenfalls ein QSS-Integrator verwendet.

Zur numerischen Lösung von Differenzialgleichungen werden üblicherweise Ordinary Differential Equation (ODE)-Solver eingesetzt. Diese berechnen Stützstellen der Lösungsfunktion über ein Zeitintervall. Im Kontext einer DES-Simulation ergeben sich die Zeitintervalle für einen ODE-Solver aus jeweils zwei aufeinanderfolgenden Ereigniszeitpunkten. Daraus folgen zumeist sehr viele kurze Zeitintervalle. Dies kann sich negativ auf die Stabilität des ODE-Solvers auswirken. Ein alternativer Ansatz ist die QSS-Methode nach Kofmann [15]. Bei diesem Ansatz wird ein Quantum zur Diskretisierung einer kontinuierlichen Zustandsgröße definiert. Für den Ofen wurde ein Temperaturquantum von 10 Kelvin festgelegt. Ein QSS-Integrator berechnet in Abhängigkeit des Zustandes und der Eingangsgröße die Zeit, die benötigt wird, um ein neues Quantum zu erreichen. Dies hat den Vorteil, dass bei Änderungsrate Null keine Ereignisse zur Neuberechnung des Zustandes auftreten. Bei hohen Änderungsraten nimmt in Abhängigkeit des Quantums die Dichte der Ereigniszeitpunkte zu. Für dieses Modell wurde der QSS-Integrator 1. Ordnung mit Hysterese nach Kofman [15] implementiert.

Prozesssteuerungsmodell (CM)

Das CM des Ofens dient der Steuerung, Kontrolle und Überwachung der Arbeitsoperationen. Es besteht, wie in Abbildung 19 gezeigt, aus zwei atomaren Modellen: *checkTemperature* und *furnace_controller*.

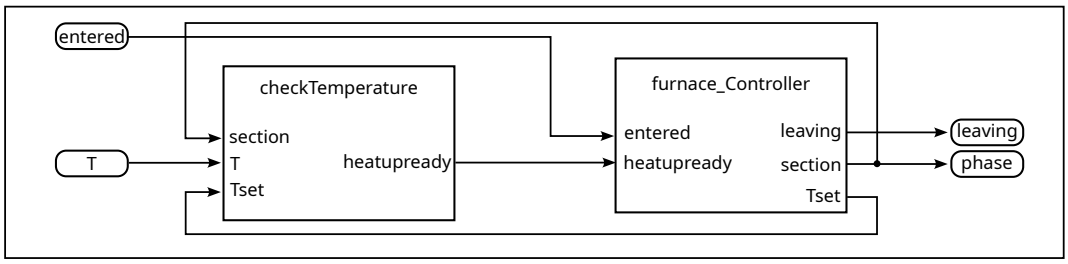


Abbildung 19: Aufbau des Prozesssteuerungsmodell (CM)

Das atomare Modell *furnace_controller* dient der Steuerung der Arbeitsoperationen des Ofens. Diese entspricht einer Ablaufsteuerung und ist in Abbildung 20 dargestellt. Das Modell ist mit einem DEVS-Diagramm gemäß Abschnitt 5.3 beschrieben. Analog zu den Arbeitsoperationen besteht es aus den Phasen *LOAD*, *HEATUP*, *HOLD* und *UNLOAD*. Initial befindet sich die Steuerung in der Phase *LOAD*. Wenn

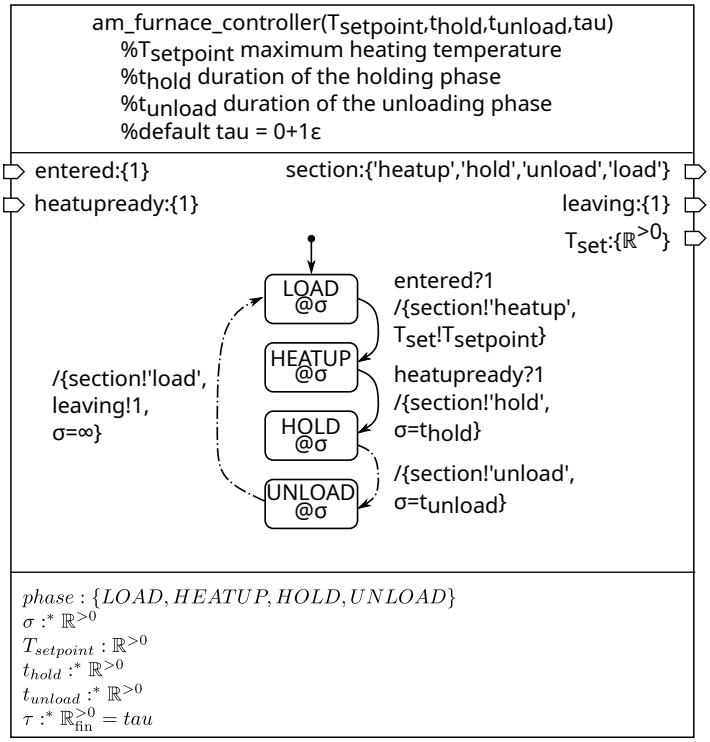


Abbildung 20: DEVS-Diagramm des *furnace_controller*

am Port *entered* ein Ereignis mit dem Wert 1 eintrifft, werden am Ausgangsport *section* der Wert *'heatup'* und am Port T_{set} der vorgegebene Wert T_{setpoint} als Ereignis versendet. T_{setpoint} ist der Temperatursollwert des Ofens. Gleichzeitig erfolgt ein Zustandsübergang in die Phase *HEATUP*. In diesem Zustand bleibt das Modell solange bis am Port *heatupready* ein Ereignis eintrifft, welches signalisiert, dass der Ofen die Solltemperatur erreicht hat. Jetzt wird am Port *section* das Ereignis *'hold'* versendet und das Modell wechselt in die Phase *HOLD*. Dieser Zustand bleibt für die Zeitspanne t_{hold} aktiv. Nach Ablauf der Zeitspanne t_{hold} wird am Port *section* das Ereignis *'unload'* ausgegeben und das Modell wechselt in die Phase *UNLOAD*. Dieser Zustand bleibt für t_{unload} Zeiteinheiten aktiv. Anschließend wird am Port *section* das Ereignis *'load'* ausgegeben und am Port *leaving* wird ein Ereignis mit dem Wert 1 ausgegeben. Danach wechselt das Modell in die Phase *LOAD* und plant mit $\sigma = \infty$ kein neues Zeitereignis ein.

Das atomare Modell *checkTemperature* dient der Temperaturüberwachung während des Aufheizens in der Phase *HEATUP*. Es erhält vom PM die aktuelle Ofentemperatur T und vergleicht diese mit der vom Teilmodell *furnace_controller* vorgegebenen Solltemperatur T_{setpoint}. Ist die Solltemperatur erreicht, wird über den Port *heatupready* an das Modell *furnace_controller* ein Ereignis gesendet.

Materialflussmodell (MF)

Das MF-Modell besteht nur aus einem atomaren Modell *furnace_server*, welches über den Eingangsport *in* ein Ereignis vom Typ Batch empfangen kann, wobei unter einem Batch eine Anzahl von gleichartigen Werkstücken verstanden wird. Das Modell definiert die Zustände *idle* und *working*. Der Initialzustand ist *idle*. Bei Empfang eines Ereignisses am Port *in* erfolgt ein Zustandsübergang nach *idle*. Gleichzeitig wird über jeweils ein Ausgangsereignis an den Ports *entered* sowie *working* dem CM und der vorgelagerten Queue im MF die Belegung des Servers signalisiert. Bei Empfang eines Ereignisses vom CM am Port *leaving* wird der Batch über den Port *out* an die nächste Komponente im MF versendet und der vorgelagerten Queue im MF über den Port *working* per Ereignis signalisiert, dass der Server frei ist.

6.3 Spezifikation eines Experiments

In diesem Abschnitt wird die Spezifikation eines Experiments beschrieben. Nach Schmidt [82] werden simulationsbasierte Experimente in die drei Phasen frühzeitig, mittelfristig und langfristig unterteilt. Die Phasen unterscheiden sich im Aufwand des Experiments und verfolgen unterschiedliche Ziele. Die frühzeitige und mittelfristige Phase beinhaltet meist Parameteruntersuchungen mit dem Ziel, Parameter, die einen geringen Einfluss auf das Ergebnis haben, auszusortieren. Dadurch wird die Komplexität für anschließende Optimierungen in der langfristigen Phase eingegrenzt.

Die frühzeitige Phase kann als Experimentziel beispielsweise ein Screening beinhalten. Bei diesem Experiment werden signifikante und nicht signifikante Parameter ermittelt. Parameter, die nicht signifikant sind, müssen bei späteren Experimenten

nicht mehr betrachtet werden. In der mittelfristigen Phase kann eine Sensitivitätsanalyse durchgeführt werden. Hier werden auch signifikante und nicht signifikante Parameter ermittelt, allerdings wird der Einfluß von Parametern quantitativ bewertet. In der langfristigen Phase werden signifikante Parameter oft mittels Optimierungsverfahren eingestellt.

In der vorliegenden Arbeit wird nur die frühzeitige Phase genauer betrachtet. Als Verfahren wird das Design Of Experiments (DOE) verwendet. Dieses Verfahren variiert die Eingangsparameter in ihrem Wertebereich und prüft, ob sich die Ausgangswerte dadurch verändern. Für die auszuführenden Experimente wird die *Statistics and Machine Learning* Toolbox von The Mathworks [59] verwendet. Wie in der Einleitung zu diesem Kapitel bereits dargestellt, zählt die Anwendungsstudie nach der Klassifikation von Schmidt [82] zu den *hochkomplexen Experimenten*, da unterschiedliche Modellstrukturen und Modellparametrierungen zu untersuchen sind. Für das gegebene Problem müssen pro Modellstruktur 32 Sätze von Eingangsparametern analysiert werden. Bei 81 möglichen Modellstrukturen sind demzufolge 2592 Simulationsläufe auszuführen.

Zur automatisierten Durchführung derartig komplexer Experimente führte Schmidt in [82] einen Konzeptrahmen ein, auf dem auch in dieser Arbeit aufgebaut wird. Ein wesentliches Element des Konzeptrahmens ist das System Entity Structure (SES)/Model Base (MB) Framework. Dieses wurde von Zeigler [104] eingeführt und seitdem kontinuierlich weiterentwickelt (z.B. [80, 70, 28]). Das Framework ermöglicht die Beschreibung von Modellvarianten und die Ableitung sowie Generierung von ausführbaren Modellen. Abbildung 21 zeigt das prinzipielle Vorgehen der SES-MB-basierten Modellbildung und Simulation. In einer MB werden parametrierbare Modelle organisiert. Im Sinne von DEVS können dies atomare oder gekoppelte Modelle sein. Mit der SES, einer speziellen Baumstruktur, werden die Modellvarianten unter Verwendung von formalen Referenzen auf Modelle in der MB spezifiziert. Die Ableitung einer konkreten Modellvariante erfolgt mit der Methode *prune*. Das Ergebnis, die *Pruned Entity Structure*, ist eine simulatorunabhängige Baumstruktur, welche die Modellstruktur und die Parametrierung der enthaltenen Komponenten beschreibt. Die Generierung eines ausführbahren Modells erfolgt mit der Methode *build*. Hier werden die formalen Referenzen auf die Modelle in der MB aufgelöst. Zur automatisierten Ableitung von Modellvarianten kann eine SES Variablen definieren, welche in Knotenattributen des SES-Baumes ausgewertet werden. Die Wertezuweisung an SES-Variablen erfolgt beim Aufruf der Methode *pruning*.

Zur automatisierten Durchführung von Experimenten erweitert Schmidt in seinem Konzeptrahmen das SES/MB Framework um eine Experiment Control (EC) und eine Excecution Unit (EU). Die EC steuert den Experimentablauf auf Basis eines spezifizierten Experimentziels sowie vorgegebener Experimentschritte. Die EU dient der Ausführung eines Modells und stellt die Verbindung zwischen Modell und Simulationslaufzeitsystem her. Demgemäß kommuniziert die EC mit dem SES/MB Framework und der EU. Die EC sendet an das SES/MB Framework eine Wertebelegung der SES-Variablen und startet über die Methoden des Frameworks die Auswahl einer Modellkonfiguration und die Generierung eines ausführbaren Modells. Dieses

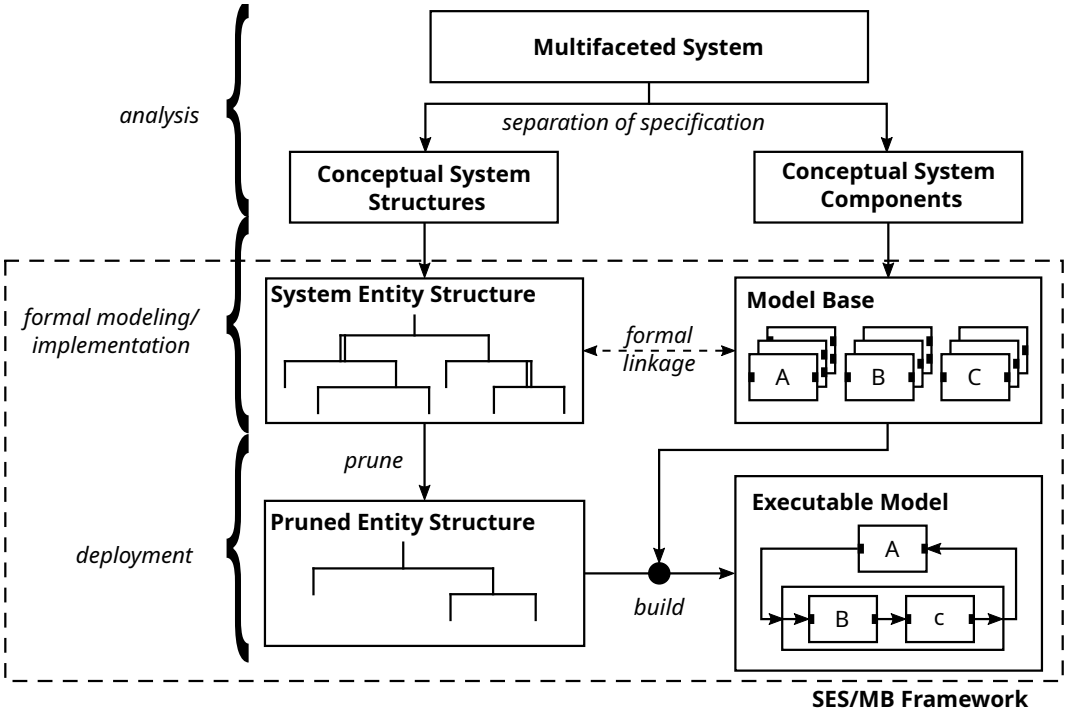


Abbildung 21: Vorgehensmodell der SES/MB-basierten Modellbildung und Simulation nach [70]

wird an die EC zurückgegeben. Die EC setzt die notwendigen Parameter und Methoden zur Ausführung des Modells und sendet das Modell mit den Ausführungsinformationen an die EU. Dort wird das Modell ausgeführt und die Ergebnisse werden an die EC zurückgesendet.

Im Fokus der vorliegenden Arbeit liegt die Beschleunigung von hochkomplexen Experimenten durch Parallelverarbeitung auf HPC-Infrastrukturen. Vor diesem Hintergrund wurde der Konzeptrahmen nach Schmidt [82] wie in Abbildung 22 dargestellt modifiziert. Danach kommuniziert die EU direkt mit dem SES/MB-Framework. Bei einer parallelen Ausführung von Simulationsläufen muss für jeden Simulationslauf ein Modell generiert werden. Die direkte Kommunikation zwischen EU und SES/MB-Framework reduziert erheblich den Kommunikationsaufwand. Unter Vorgaben sind alle Methoden und Parameter spezifiziert, die der EC zur Verfügung zu stellen sind. In der EC werden mittels vollständiger Enumeration aus den Wertebereichen der SES-Variablen die zu untersuchenden Modellkonfigurationen in Form von Sätzen von SES-Variablenbelegungen bestimmt (kodierte Modellkonfigurationen). Anschließend bestimmt daraus die Experimentmethode für jede kodierte Modellkonfiguration die Eingangsparametersätze. Auf Basis der kodierten Konfigurationen und der zu untersuchenden Eingangsparameter wird ein Experimentplan erstellt. Nachfolgend wird zur Ausführung des Experimentplans ein HPC-Job erstellt und an die EU zur Ausführung gesendet. Die EU kommuniziert mit dem SES/MB-Framework und weist dieses auf Basis des HPC-Jobs mit einem Experimentplan an, die zu untersuchenden Modelle zu generieren. Die EU kann die Simulationsläufe zu

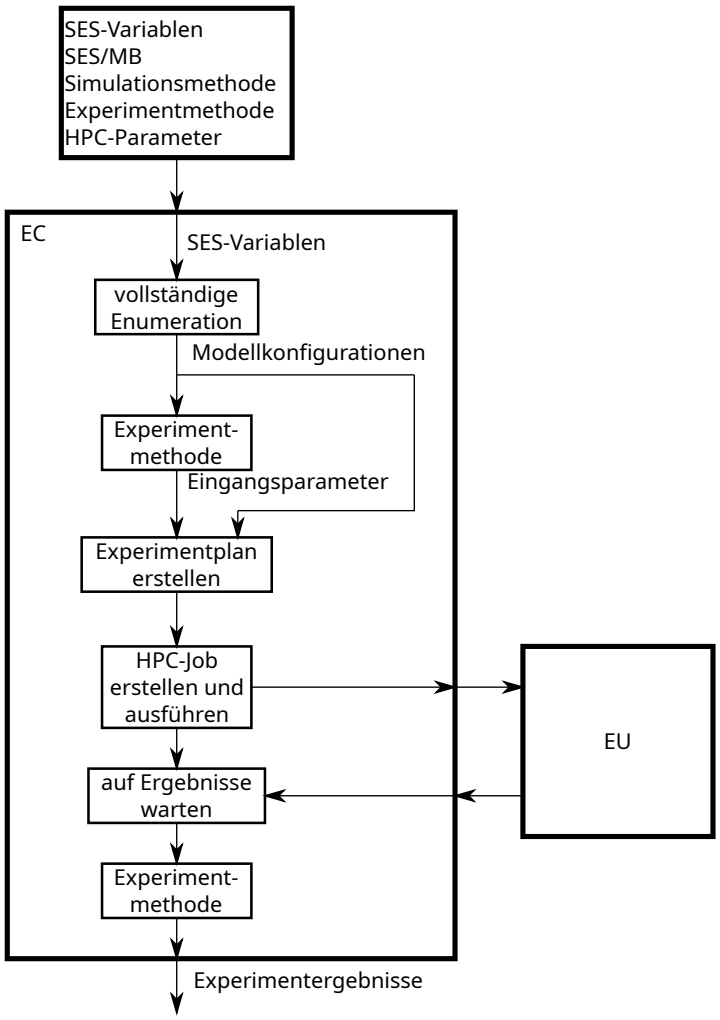


Abbildung 22: Konzeptrahmen für die Ausführung von hochkomplexen Experimenten auf HPC-Infrastrukturen

untersuchender Modelle parallel oder sequenziell organisieren. In den nächsten Abschnitten wird die EU näher betrachtet. Die Ergebnisse der Simulationsläufe werden an die EC zurückgegeben, wo sie durch die jeweilige Experimentmethode weiterverarbeitet werden.

6.4 Sequentielle Ausführung eines Experiments

Das Konzept zur sequentiellen Ausführung von komplexen und hochkomplexen Experimenten durch die EU zeigt Abbildung 23. Voraussetzung ist eine HPC-Infrastruktur, welche jobbasiert arbeitet. Die EU empfängt einen HPC-Job, welcher die SES/MB und den Experimentplan enthält. Die SES/MB wird an das SES/MB-Framework und der Experimentplan an den *Workgenerator* übergeben.

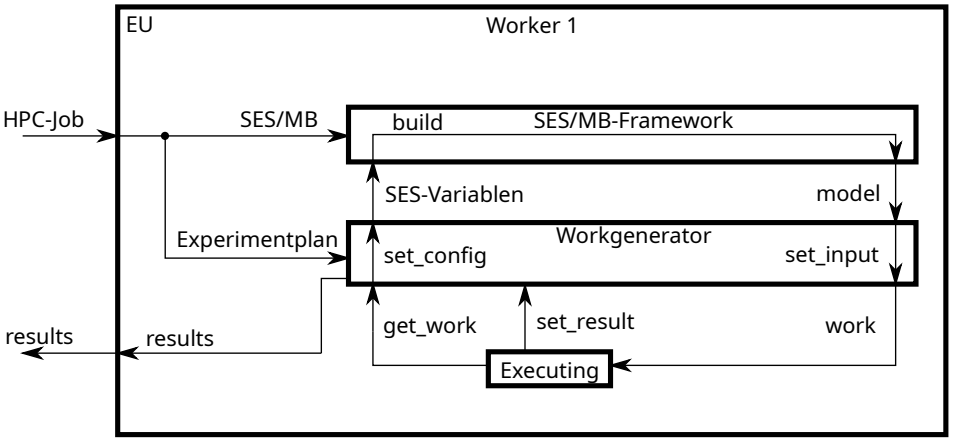


Abbildung 23: Konzept zur sequentiellen Ausführung von Simulationsläufen bei komplexen und hochkomplexen Experimenten

Anschließend erhält der Workgenerator die Anforderung *get_work*. Daraufhin erstellt der Workgenerator nach dem Experimentplan die Konfiguration für einen Simulationslauf und übergibt die notwendige SES-Variablenbelegung dem SES/MB-Framework (*set_config*). Das Framework generiert daraufhin ein ausführbares Modell (*model*). Dieses wird vom Workgenerator gemäß dem Experimentplan um die notwendigen Eingangsparametern ergänzt (*set_input*). Anschließend wird der vollständig konfigurierte Simulationslauf (*work*) ausgeführt und die Ergebnisse werden dem Workgenerator übergeben (*set_result*). Dieser organisiert alle Ergebnisse bis der Experimentplan abgearbeitet ist. Abschließend werden die Ergebnisse (*results*) an die EC zurückgesendet.

6.5 Parallele Ausführung eines Experiments

Zur Beschleunigung der Ausführung von komplexen und hochkomplexen Experimenten wurde eine EU zur parallelen Ausführung von Simulationsläufen nach dem Konzept in Abbildung 24 entwickelt.

Im Gegensatz zur sequentiellen Ausführung werden mehrere (mindestens zwei) *Worker* verwendet. Zur Organisation und Koordination der Ausführung wird ein *Loadbalancer* verwendet, der im Worker 1 integriert ist. Der Loadbalancer wird benötigt, da die Ausführung der Simulationsläufe unterschiedlich viel Zeit in Anspruch nehmen kann. Die Kommunikation mit dem Workgenerator übernimmt der Loadbalancer. Dieser fordert Arbeit an (*get_work*) und verteilt vollständig konfigurierte Simulationsläufe (*work*) an die Worker. Hat ein Worker seinen Simulationslauf beendet, werden die Ergebnisse über den Loadbalancer an den Workgenerator weitergegeben (*result, set_result*) und der Loadbalancer fordert neue Arbeit an (*work*). Durch

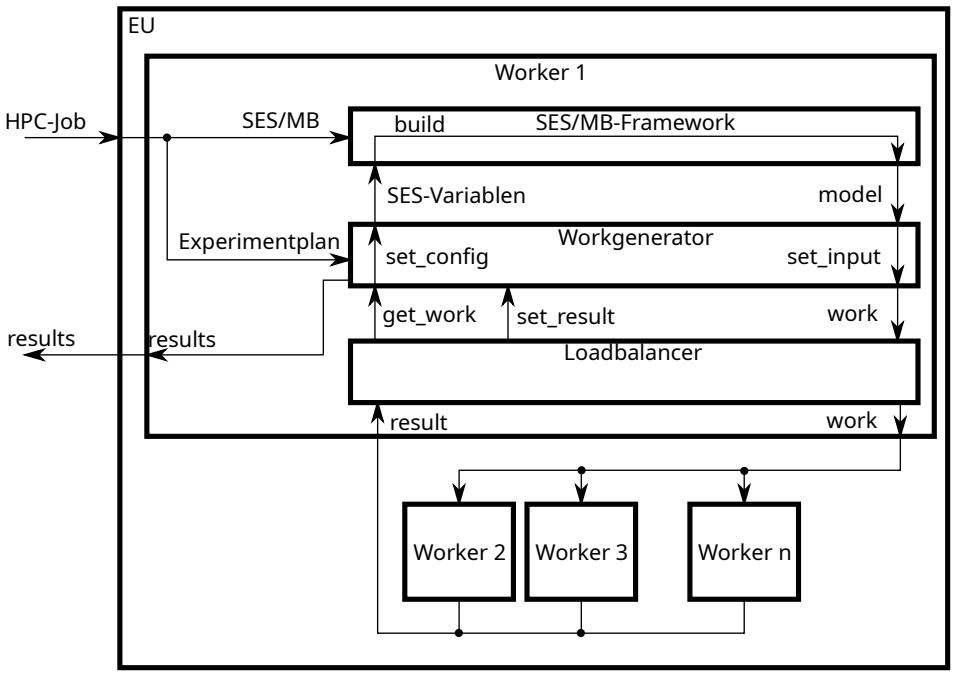


Abbildung 24: Konzept zur parallelen Ausführung von Simulationsläufen bei komplexen und hochkomplexen Experimenten

dieses Vorgehen werden die Worker ausgeglichen beschäftigt und Wartezeiten der Worker vermieden.

6.6 Laufzeitstudien und Abschätzungen

Bereits 2021 wurde vom Autor dieser Arbeit ein HPC-System mittlerer Größe bestehend aus drei Nodes mit jeweils zwei AMD Epyc 7552 aufgebaut. Diese Multicore-Prozessoren verfügen über jeweils 48 Cores. Das Gesamtsystem stellt damit 288 Cores bereit. Die Installation wurde durch die Private Hochschule für Wirtschaft und Technik (PHWT), Vechta/Diepholz finanziert und konnte für einen begrenzten Zeitraum am Standort Diepholz für Laufzeituntersuchungen im Kontext dieser Arbeit eingesetzt werden. Das HPC-System trägt den Namen Seneca. Die vollständige Spezifikation von Seneca ist in Tabelle 6 aufgelistet.

Tabelle 6: Seneca Hardware- und Software-Übersicht

Nodes	3
Cores	288
Processors	AMD Epyc 7552
Main memory	1536 GiByte
High-speed network	100 GBit/s InfiniBand
Management network	1 GBit/s Ethernet
Operating system	OpenSuse Leap 15.3
Middleware	OpenHPC
Cluster management	Warewulf
Job Scheduler	SLURM
Software	GCC 9.3.0, GSL 2.6, open MPI 4.1.1, ucx 1.13.0, libfabric 1.13.0, hwloc 2.1.0, Matlab R2021a

Voruntersuchungen

Zunächst wurde das HPC-System mit dem bekannten SNE-Benchmark CP1/CP2-MC [7, 6] getestet. Bei diesem Benchmark ist eine simulationsbasierte Parameterstudie für ein Feder-Masse-System auszuführen. Das Benchmark-Problem wurde in C unter Verwendung von Open MPI und in Matlab mit Hilfe der Parallel Computing Toolbox (PCT) von The Mathworks implementiert. Die PCT unterstützt eine Vielzahl von Technologien für die verteilte und parallele Verarbeitung. Im Rahmen der Voruntersuchung wurden die Varianten `parfor`, `spmd` und `parsim` getestet. Erwartungsgemäß lieferte die C/MPI-Implementierung auf Seneca um Größenordnungen bessere Laufzeitergebnisse als die untersuchten Matlab/PCT-Varianten. Im Sinne des Implementierungskomforts schnitt die PCT-Variante `parsim` am besten ab. Allerdings ist `parsim` nur für sehr einfach strukturierte Anwendungsprobleme geeignet. Bereits der CP1/CP2-MC-Benchmark ist für parsim zu komplex, so dass keine signifikanten Beschleunigungen durch Parallelverarbeitung erzielt werden konnten. Die detaillierten Ergebnisse dieser Voruntersuchung wurden bereits in [43] veröffentlicht.

Die erste Voruntersuchung hat nochmals deutlich gezeigt, wie entscheidend eine effiziente Implementierung des Simulationslaufzeitsystems für hohe Verarbeitungsgeschwindigkeiten auch auf HPC-Systemen ist. Für praxistaugliche Anwendungen kommen demnach nur Implementierungen in compilierbaren Sprachen in Frage. Im Kontext der vorliegenden Arbeit wäre damit eine Realisierung des NSA-DEVS-Formalismus beispielsweise in C oder C++ wünschenswert. Aufgrund der Komplexität des Formalismus konnte im Rahmen der zur Verfügung stehenden Projektzeit jedoch nur eine erste prototypische Implementierung realisiert werden. Hierzu wurde die etablierte Entwicklungsumgebung Matlab benutzt. Das im vorangegangen Abschnitt beschriebene Konzept zur parallelen Experimentausführung wurde nach den Erfahrungen der ersten Voruntersuchung mit der PCT-Variante `spmd` umgesetzt.

Bei der genauen Analyse der im Rahmen der ersten Voruntersuchung durchgeführten Speedup-Studien, erwiesen sich die PCT-Varianten `spmd` und `parfor` als ähnlich leistungsfähig. Allerdings fielen bei beiden Varianten an jeweils zwei Stellen im Speedup-Verlauf scheinbare Anomalien auf, die dort vorläufig als Treppeneffekt bezeichnet wurden. Wegen dieser Beobachtung wurde eine zweite Voruntersuchung, nunmehr aber unter Verwendung der Prototypimplementierung NSA-DEVSforMATLAB [12] und einer Strukturvariante des im Abschnitt 6.1 eingeführten Modells einer Produktionskette, durchgeführt.

Erneut wurden die sequentielle Laufzeit t_s sowie parallele Laufzeiten t_p unter Verwendung unterschiedlicher Prozessorzahlen bestimmt. Im Falle von Seneca sind unter Prozessoren die Cores zu verstehen. Bei Verwendung der PCT muss auf jedem Core eine Matlab-Instanz laufen. Im PCT-Kontext wird dann von Workern gesprochen. Diese Bezeichnung wird auch für die Speedup-Darstellung in Abbildung 25 benutzt. Zum besseren Verständnis der Abbildung sei darauf hingewiesen, dass die

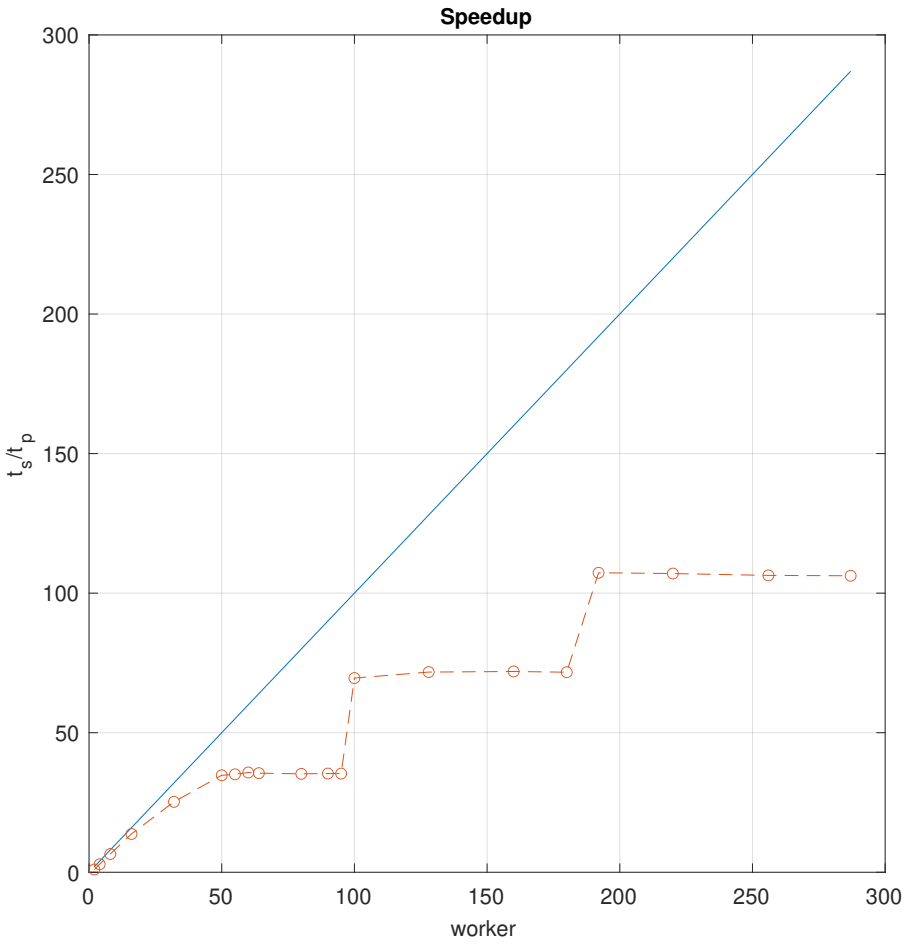


Abbildung 25: Speedup-Verlauf der zweiten Voruntersuchung

Speedup-Studie bereits bei 287 Workern endet, obwohl auf Seneca 288 Cores zur Verfügung stehen und auch die PCT-Lizenz 288 Instanzen umfasst. Eine dieser Instanzen wird allerdings PCT-intern benutzt und steht damit nicht für die eigentliche Anwendung zur Verfügung.

Bei der ersten Voruntersuchung wiesen alle parallel auszuführenden Simulationsläufe den gleichen Rechenbedarf auf. Deshalb konnte bereits zu Beginn einer Experimentdurchführung eine einfache statische Lastverteilung vorgenommen werden. Dieses Vorgehen ist bei der Anwendungsstudie dieser Arbeit nicht möglich, weil verschiedene Strukturvarianten mit unterschiedlichen Rechenbedarfen zu simulieren sind. Um das Problem zu lösen führt ein Worker während der gesamten Experimentlaufzeit eine dynamische Lastverteilung durch. Die Speedup-Studie beginnt deshalb mit zwei Workern. Dabei findet noch keine tatsächliche Parallelverarbeitung statt, aber bereits eine Problemlösung durch den parallelen Algorithms. Die Ausführungszeit wird meist als t_1-Laufzeit bezeichnet und ist Basis zur Bewertung des Parallelisierungsoverheads. Im Rahmen der Voruntersuchung wurde für t_s/t_1 ein sehr guter Wert von 0,99 ermittelt. Dementsprechend verläuft die Speedup-Kurve bis 32 Worker nahe dem idealen Speedup.

Ab 50 Worker tritt dann aber eine komplette Stagnation ein. Bei oberflächlicher Betrachtung könnte man den Verlauf des ersten Drittels der Speedup-Kuve mit dem Amdahlschen Gesetz begründen wollen (vgl. Abschn. 2.4). Dies wäre aber eine Fehlinterpretation, denn an der Stütztelle 100 Worker verdoppelt sich der Speedup dann plötzlich und ab 192 Worker kommt es nochmals zu einem Sprung, was ähnlich bereits in der ersten Voruntersuchung [43] auffiel und dort als Treppeneffekt bezeichnet wurde.

Bei der Interpretation der gesamten Speedup-Kurve muss man berücksichtigen, dass die PCT ein Prozessormapping nach folgendem Schema durchführt: Bei bis zu 95 Worker werden ausschließlich Cores des ersten Nodes verwendet. Erst ab 96 Worker verteilt die PCT möglichst gleichmäßig auf die Cores des ersten und zweiten Nodes, also je 48 Worker der Anwendung auf jedem Node. Daraus kann man folgern, dass die Stagnation ab 50 Worker im ersten Drittel der Speedup-Kurve auf einen sogenannten Speicherflaschenhals hinweist. Das ist ein bekanntes Phänomen bei Architekturen mit gemeinsamen Speicher. Bei 96 Worker beginnt die PCT mit einer gleichmäßigen Verteilung auf zwei Nodes, dass heißt beide Nodes führen jeweils 48 Worker der Anwendung aus. Dies reduziert konkurrierende Speicherzugriffe schlagartig, was zu dem beobachteten ersten Sprung im Speedup führt. An den nächsten Stützstellen 128, 160 und 180 Worker stecken dann aber beide Nodes bereits wieder in ihren Speicherflaschenhälsen, was sich im Seedup-Verlauf als erneutes Plateau darstellt. Die Cores des dritten Nodes werden erst ab 192 Worker durch die PCT in das Mapping einbezogen, wodurch es zum zweiten Sprung kommt und sich der Speedup nochmals um etwa ein Drittel erhöht. An den restlichen Stützstellen 220, 256 und 287 Worker befinden sich alle drei Nodes wieder im Bereich ihrer Speicherflaschenhälse.

Nach der Speedup-Analyse der zweiten Voruntersuchung kann man, unter der Annahme, dass die Erklärung des beobachteten Treppeneffekts zutreffend ist, zusammenfassend feststellen, dass die Konstruktion von Seneca (3 Nodes mit je 2 x 48

Cores AMD/NUMA) für die hier betrachtete Anwendungsstudie in Kombination mit NSA-DEVSforMATLAB und der PCT nicht optimal ist, weil es zu Speicherflaschenhälsen kommt. Ein solches Ergebnis ist durchaus praxistypisch, weil es kaum möglich ist, für komplexe Anwendungen ohne vorhergehende Experimente ein optimales HPC-System zu entwerfen. Beim konkreten System Seneca, einer inklusive Software circa 80.000 Euro-Investition kommt hinzu, dass dieses System nicht allein auf die Bedürfnisse dieser Arbeit hin entworfen wurde, sondern für eine Vielzahl von Projekten einsetzbar sein sollte. Im Kontext dieser Arbeit ergab die zweite Voruntersuchung, dass für eine maximale Beschleunigung erwartungsgemäß alle drei Nodes einzusetzen sind. Dies ist ab einer Worker-Anzahl von 192 der Fall. Trotz des für die hier betrachtete Anwendung nicht optimalen Designs, könnte Seneca nach den Ergebnissen der Speedup-Studie mit einer Investition von circa 15.000 Euro pro Node aufgerüstet werden und damit die Anwendungsstudie weiterhin signifikant beschleunigen.

Im Rahmen der zweiten Voruntersuchung wurde neben der Speedup-Analyse noch die Effizienz der dynamischen Lastverteilung überprüft. Wie bereits angemerkt, wird diese während der gesamten Experimentlaufzeit durch den Worker 1 realisiert. Dieser beauftragt zunächst alle weiteren zur Verfügung stehenden Worker mit jeweils einem auszuführenden Simulationslauf. Sobald ein Simulationslauf abgeschlossen ist, werden die Ergebnisse an den Worker 1 zurückgegeben. Dieser übermittelt danach sofort einen weiteren Simulationsauftrag an den frei gewordenen Worker. Zur Bewertung dieser Verteilungsstrategie wurden die Laufzeiten aller ausgeführten Simulationen sowie die Wartezeiten der Worker, bis diese wieder mit einem neuen Auftrag belegt werden, gemessen.

Die Ergebnisse dieser Untersuchung sind in Abbildung 26 in Form von zwei Diagrammen dargestellt. Aus dem oberen Diagramm ist ersichtlich, dass die Laufzeiten der Simulationsläufe in Abhängigkeit der parallel eingesetzten Worker im Bereich von circa 50 bis 250 Sekunden liegen. Das untere Diagramm stellt die mittlere Wartezeit der Worker auf einen neuen Simulationsauftrag dar. Weitgehend unabhängig von der Anzahl der parallel eingesetzten Worker liegt diese Leerlaufzeit bei circa 1,5 Millisekunden. Im Verhältnis zu den Ausführungszeiten der Simulationsläufe sind die Wartezeiten also vernachlässigbar. Damit konnte eine sehr gute Effizienz der implementierten dynamischen Lastverteilung nachgewiesen werden.

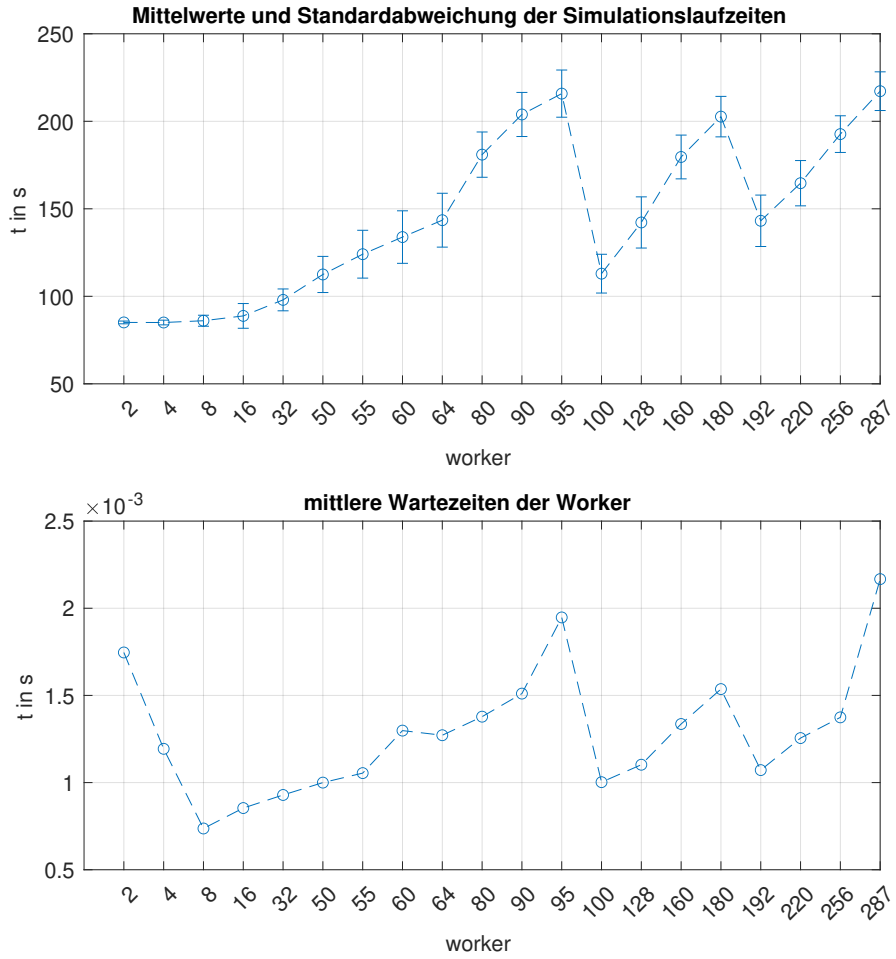


Abbildung 26: Mittelwerte und Standardabweichung der Simulationslaufzeiten sowie mittlere Wartezeiten der Worker-Instanzen

Screening-Experiment

Nach Abschluss der Voruntersuchungen wurde anhand der Produktionsstrecke ein Parameter-Screening unter Verwendung der Prototypimplementierung NSA-DEVSforMATLAB und des HPC-Systems Seneca durchgeführt.

Nach Schmidt [82] ist das Screening ein simulationsbasiertes Experiment der frühzeitigen Phase beim Planen von Fertigungssystemen. Screening-Verfahren wie beispielsweise *Design Of Experiments* (DOE) [84] oder *Elementary Effect* (EF) haben einen hohen Rechenbedarf und gehören nach Schmidt damit zur Klasse der komplexen Experimente. Im Falle der hier betrachteten Anwendungsstudie mit ihren 81 möglichen Strukturvarianten und jeweils 7 Systemparametern ist der Parametervektor nach DOE 32-fach zu variieren, um die Signifikants der Systemparameter zumindest qualitativ bewerten zu können. Damit ergibt sich ein Rechenaufwand von 2.592 Simulationsläufen (81 Strukturvarianten x 32 Parametervariationen).

Um eine grobe Vergleichbarkeit des durchgeführten Screenings mit den Ergebnissen in Schmidt [82] zu ermöglichen, wurde als Simulationshorizont ein Produktionszeitraum von drei Tagen gewählt. Das Ergebnis des Screenings ist in Abbildung 27 dargestellt. Auf der Abszisse sind alle 81 untersuchten Strukturvarianten abgetragen. Auf den Ordinaten sind die 7 untersuchten Parameter bezüglich der 6 Bewertungsgrößen: Produktionszeit pro Bauteil (*procTime*), Anzahl gefertigter Bauteile (*thrput*), aufgewendete Energie pro Bauteil (*Espec*), maximal erforderliche elektrische Leistung (*loadPeak*), Summe der Auslastung aller Fertigungseinrichtungen (*pcUtil*) und Maximalbelegung aller Puffer (*pcStock*) aufgeführt. Die für eine Bewertungsgröße in einer Struktur als signifikant qualifizierten Parameter sind jeweils mit einem Kästchen markiert. Parameter, die für eine Bewertungsgröße innerhalb einer Struktur als nicht-signifikant identifiziert wurden, tragen keine Markierung und werden als lokal nicht-signifikant bezeichnet. Bei der Durchsicht aller lokal nicht-signifikanten Parameter können als Untermenge die für eine Struktur global nicht-signifikanten Parameter bestimmt werden, also die Parameter, die keine Signifikanz für alle 6 Bewertungsgrößen besitzen. Diese wurden in der Abbildung 27 mit einem roten Kreuz gekennzeichnet.

Die global nicht-signifikanten Parameter können in nachfolgenden Experimenten ignoriert werden. Dadurch kann beispielsweise die Dimension des Suchraums bei simulationsbasierten Optimierungen verringert werden. In der untersuchten Anwendungsstudie ist dies bei 4 Strukturvarianten der Fall. Insgesamt wurden 8 Parameter als global nicht-signifikant identifiziert.

Entsprechend den Erkenntnissen aus den Voruntersuchungen wurde das Screening auf Seneca unter Verwendung von 192 Workern durchgeführt. Die Experiment-Laufzeit betrug circa 4 Stunden. Bei der parallelen Ausführung des Experiments wurden die Laufzeiten der 2.592 Simulationsläufe jeweils einzeln gemessen. Die Laufzeiten variierten relativ stark zwischen 6 und 35 Minuten, was einerseits durch die unterschiedlichen Rechenbedarfe der Strukturvarianten erklärbar ist, aber auch auf die bereits in den Voruntersuchungen identifizierten Flaschenhälse bezüglich der Speicherzugriffe auf Seneca zurückzuführen ist. Im Mittel ergab sich eine Laufzeit pro Simulation von 17 Minuten.

Würde man das Screening-Experiment auf einer zu Seneca vergleichbaren Prozessortechnologie sequentiell ausführen, würden sich die Laufzeiten der notwendigen Simulationsläufe auf fast 31 Tage summieren. Setzt man dazu die 4 Stunden der parallelen Ausführung auf Seneca ins Verhältnis, ergibt sich ein Speedup von circa 186.

Abbildung 27: Ergebnis des Screenings

Sensitivitätsanalyse

In der mittelfristigen Entwurfsphase ist es unter Umständen gewünscht, den Einfluss der Systemparameter auf die Bewertungsgrößen eines Fertigungssystems auch quantitativ zu untersuchen. Dafür werden Verfahren der Sensitivitätsanalyse wie beispielsweise nach Sobol [86] eingesetzt, die noch rechenaufwendiger als das Parameter-Screening sind und ebenfalls zur Klasse der komplexen Experimente gehören.

Schmidt führt in [82] für die Anwendungsstudie eine Laufzeitabschätzung einer Sensitivitätsanalyse nach Sobol durch. Diese basiert auf der ihm damals zur Verfügung

stehenden 8-Core-Technik und während eines Screenings gemessener Minimal- und Maximallaufzeiten der Simulationsläufe von 1,4 bis 5,4 Minuten. Unter der Annahme eines idealen Speedups kommt er für ein Sensitivitätsanalyse-Experiment auf Minimal-/Maximallaufzeiten von 46 bis 171 Tagen. Setzt man in der Hochrechnung von Schmidt den beim Screening auf Seneca ermittelten Speedup von 186 an, dann verringert sich die Laufzeit für eine Sensitivitätsanalyse auf eine Größenordnung von 2 bis 7 Tagen.

Eine Sensitivitätsanalyse unter Verwendung des Simulators NSA-DEVSforMATLAB würde derzeit wegen seiner noch nicht optimierten Implementierung im Bereich von 10 bis 35 Tagen liegen.

Struktur- und Parameteroptimierung

Nach Schmidt [82] schließt der Entwurf von Fertigungssystemen mit der langfristigen Phase ab. Bei der hier betrachteten Anwendungsstudie wäre in dieser Phase schließlich eine Struktur- und Parameteroptimierung durchzuführen. Schmidt schlägt hierfür Verfahren wie *Simulated Annealing* (SA) oder *Genetic Algorithm* (GA) vor. Erfahrungsgemäß ist der Rechenbedarf solcher Optimierungen noch höher als bei Sensitivitätsanalysen.

Genauere Laufzeitabschätzungen sind für solche Experimente vorab kaum möglich. Beim derzeitigen Ausbau von Seneca (3 Nodes) und der noch prototypischen Implementierung des Simulators NSA-DEVSforMATLAB wären aber bereits praktische Experimente möglich, allerdings mit Laufzeiten von über einer Woche.

6.7 Zusammenfassung

Die Anwendungsstudie stellt eine gemischt diskret-ereignisorientierte und kontinuierliche (hybride) Problemstellung dar. Es zeigte sich, dass diese problemlos auf Basis des QSS-Anatzes mit der NSA-DEVSforMATLAB Prototypimplementierung gelöst werden konnte.

Die Bearbeitung der Anwendungsstudie zeigte darüber hinaus, dass mit dem NSA-DEVS Formalismus in Verbindung mit Parallelverarbeitung auf einem mittelgroßen HPC-System wie Seneca komplexe und hochkomplexe simulationsbasierte Experimente mit akzeptablem Zeitaufwand lösbar sind. So konnte ein auf 2.592 Simulationsläufen basierendes Parameter-Screening in nur 4 Stunden durchgeführt werden, gegenüber 31 Tagen bei sequentieller Arbeitsweise. Die Messungen ergaben, dass der realisierte Parallelverarbeitungsansatz gut skaliert. Damit könnte durch Aufrüstung des HPC-Systems um weitere Nodes die Ausführungszeit des Screening-Experiments noch weiter verkürzt werden.

Für eine typische Sensitivitätsanalyse ist die Anzahl der auszuführenden Simulationsläufe noch wesentlich größer. Schmidt gibt für das Verfahren nach Sobol für die Anwendungsstudie circa 360.000 erforderliche Simulationsläufe an, was auf seiner Plattform zu einer hochgerechneten Laufzeit von 46 bis 171 Tagen führt. Derartige Experimente waren mit der ihm zur Verfügung stehenden Technik also noch nicht

möglich. Die Anwendung seiner Abschätzung auf Seneca führt zu einem Laufzeitergebnis von 2 bis 7 Tagen, was in der Praxis bereits eine akzeptable Größenordnung wäre.

Der Rechenaufwand von Optimierungsexperimenten wird letztendlich durch die geforderte Ergebnisgenauigkeit bestimmt. Bei der in der Anwendungsstudie untersuchten Prozesskette wären auf Seneca innerhalb einer Woche mehrere hunderttausend Simulationsläufe möglich, wodurch praxisrelevante Ergebnisgenauigkeiten mittels stochastischer Suche erzielbar sein sollten.

7 Zusammenfassung und Ausblick

Zur Bearbeitung der im ersten Kapitel aufgestellten Fragestellungen war eine relativ breite Grundlagenbasis erforderlich, die von den allgemeinen Aspekten der Parallelverarbeitung über Methoden zur Beschleunigung von DES-Studien bis hin zu den vielfältigen Ausprägungen des DEVS-Formalismus reicht. Alle Teilgebiete sind durch eine bereits lang andauernde Entwicklungsgeschichte geprägt, die bis in die 1970er und 1960er Jahre zurückreicht. Um den aktuellen Stand der Technik in den für diese Arbeit relevanten Gebieten nachvollziehbar darzustellen, wurde der Grundlagenteil in drei separate Kapitel gegliedert.

Aus der Analyse der DEVS-Entwicklung ergab sich, dass der originale, von B. P. Zeigler 1976 vorgeschlagene DEVS-Formalismus seinen Ursprung im Bereich der diskret-ereignisorientierten Systemklasse hatte. Spätere Arbeiten konnten aufzeigen, dass DEVS aber auch sehr gut auf hybride Systemdynamiken erweiterbar ist. Insbesondere hinsichtlich eines organischen Einbezugs kontinuierlicher Systeme lieferte E. Kofmann mit seinem QSS-Ansatz ab 2001 wichtige Beiträge. Auch die anfängliche Fokusierung von Zeigler auf den Moore-Ansatz im diskret-ereignisorientierten Bereich wurde durch Weiterentwicklungen in Richtung Mealy-Dynamik aufgeweitet. Arbeiten von P. Junglas wiesen 2020 allerdings nach, dass der bis dahin vor allem durch PDEVS und RPDEVS definierte Stand der Technik bezüglich der Unterstützung diskret-ereignisorientierter Problemstellungen mit Mealy-Dynamik immer noch durch ernsthafte Defizite gekennzeichnet war. Zur Behebung dieser Probleme schlug er einen Lösungsversuch mittels der Non-Standard Analysis (NSA) vor. Der Befund von Junglas sowie seine Lösungsidee waren die Grundlage für die erste in dieser Arbeit behandelte Forschungsfrage: Ist es möglich, einen praktisch anwendbaren NSA-DEVS-Formalismus zu entwickeln?

Die Bearbeitung dieser Fragestellung und die erreichten Ergebnisse wurden im fünften Kapitel dokumentiert. Zunächst wurde, wie im DEVS-Bereich üblich, eine mengentheoretische Modellspezifikation für NSA-DEVS ausgearbeitet. Anschließend wurde ein vollständiger Abarbeitungsalgorithmus für einen sogenannten abstrakten Simulator in Pseudo-Code-Notation entwickelt. Zusätzlich wurde eine Diagrammtechnik zur visuellen Spezifikation von Modellen, die bereits für Classic DEVS eingeführt wurde, auf NSA-DEVS angepasst. Als Ergebnis der Bearbeitung der ersten Forschungsfrage kann konstatiert werden, dass die NSA-Idee von Junglas sehr zielführend ist, also die Ausarbeitung eines praktisch anwendbaren Formalismus ermöglicht. Besonders überzeugend ist, dass dieser Formalismus durch die Einführung einer hyperrellen Zeitbasis die Behandlung von Mealy-Verhalten gegenüber dem bisherigen Stand der Technik (PDEVS, RPDEVS) in einer sehr intuitiven Weise für Anwender ermöglicht. Darüber hinaus führt die NSA-Idee zu einer starken Komplexitätsreduzierung der abstrakten Simulatoralgorithmen gegenüber den bisherigen Ansätzen.

Dieser Aspekt ist insbesondere bei der praktischen Implementierung von Simulatoren nicht zu unterschätzen. So war es im Rahmen einer begrenzten Projektzeit möglich, den NSA-Formalismus zumindest prototypisch vollständig unter Verwendung der SCE Matlab zu implementieren und zu testen. Der Entwicklungsstand wurde der interessierten Community unter dem Namen NSA-DEVSforMATLAB als Open-Source über GitHub zur Verfügung gestellt.

Im sechsten Kapitel wurde die Bearbeitung der zweiten Forschungsfrage, ob praxisrelevante komplexe und hochkomplexe Simulationsstudien unter Verwendung eines NSA-DEVS-Simulators in akzeptabler Zeit auf mittelgroßen HPC-Systemen (max. 100.000 Euro) durchführbar sind, dokumentiert. Dazu wurde zunächst, dass bereits von R. Larek und A. Schmidt untersuchte Praxisproblem einer Produktionskette mit hybriden Charakter für energetische Analysen, vorgestellt. Anschließend wurde die Modellierung der Produktionskette mit Hilfe des neu entwickelten NSA-DEVS-Formalismus sowie die Spezifikation der durchzuführenden Simulationsstudien (Experimente) unter Verwendung des SES/MB-Frameworks demonstriert. Dem Ansatz von A. Schmidt folgend, wurden dann konkrete Excecution Units (EUs) zur sequentiellen und parallelen Durchführung von komplexen und hochkomplexen Experimenten unter Verwendung eines HPC-Systems entwickelt. Abschließend wurden die Ergebnisse beispielhafter Laufzeitstudien auf dem HPC-System Seneca präsentiert. Dieses System wurde durch die Private Hochschule für Wirtschaft und Technik (PHWT), Vechta/Diepholz finanziert und vom Autor dieser Arbeit entworfen und realisiert.

Im Ergebnis der Bearbeitung der zweiten Forschungsfrage können zwei Aussagen getroffen werden: (i) Der entwickelte NSA-DEVS-Formalismus ist in Kombination mit dem QSS-Ansatz für die Modellierung und Simulation komplexer hybrider Problemstellungen bestehend aus kontinuierlichen und diskret-ereignisorientierten Teilsystemen geeignet. Dies trifft insbesondere auch auf diskret-ereignisorientierte Teilsysteme mit Mealy-Charakteristik zu. (ii) Bereits beim gegenwärtigen Stand der Hardwareentwicklung ist die Durchführung komplexer und hochkomplexer DES-Studien unter Verwendung des NSA-DEVS-Formalismus auf einem HPC-System mittlerer Größe in akzeptabler Zeit möglich. Es deutet sich auch an, dass diese Studien weiteres Potential für Beschleunigungen besitzen, da sie sehr gut skalieren.

Die Entwicklung des NSA-DEVS-Formalismus konnte mit der vorliegenden Arbeit vollständig abgeschlossen werden. In einem Folgeprojekt der Forschungsgruppe Computational Engineering & Automation (CEA), HS Wismar wird nunmehr in Zusammenarbeit mit P. Junglas, PHWT an Verbesserungen der Anwenderunterstützung durch Ausbau der Modellbibliothek sowie an weiteren Leistungsuntersuchungen gearbeitet. In diesem Kontext soll unter anderem anhand des SNE-Benchmarks C22 die Effizienz des NSA-DEVS-Formalismus bei sehr anspruchsvollen Queueing-Problemen geprüft werden.

Die im Rahmen dieser Arbeit entstandene Simulatorimplementierung NSA-DEVSforMATLAB hat einen prototypischen Status. Wie die im sechsten Kapitel präsentierten Ergebnisse zeigen, ist der Simulator aber bereits für praxisrelevante Anwendungsprobleme einsetzbar. Da die bisherige Implementierung vollständig in

M-Code ausgeführt wurde, besteht aber grundsätzlich noch ein großes Beschleunigungspotential, welches künftig gehoben werden sollte. Dazu müssen die laufzeitkritischen Simulatorbestandteile in eine compilierbare Programmiersprache portiert und optimiert werden. Hierfür wären insbesondere die Sprachen C oder C++ geeignet, weil diese neben Fortran direkt durch das Matlab-External-Interface (MEX) unterstützt werden. Eine Einbindung der Kompilate über die MEX-Technologie erscheint unbedingt erforderlich, damit potentiellen Anwendern die komfortable SCE-basierte Arbeitsweise weiterhin erhalten bleibt.

Um Co-Simulationen und andere verteilte Simulationen zu ermöglichen, wäre als Erweiterung des Simulators die Implementierung einer HLA-Schnittstelle wünschenswert. Im Rahmen der SCE Matlab kann eine solche Implementierung unter Verwendung der von der FG CEA bereits 2008 entwickelten MatlabHLA-Toolbox mit relativ geringem Aufwand realisiert werden.

Literaturverzeichnis

[1] AMD. *4TH GEN AMD EPYC™ PROCESSOR ARCHITECTURE*. Techn. Ber. Advanced Micro Devices, 2024.

[2] G. M. Amdahl. „Validity of the single processor approach to achieving large scale computing capabilities.“ In: *AFIPS '67 (Spring) Proceedings* (1967). doi: 10.1145/1465482.1465560, S. 483–485.

[3] F. Barros. „On the representation of time in modeling & simulation“. In: *Winter Simulation Conference*. Dez. 2016, S. 1571–1582. DOI: 10.1109/WSC.2016.7822207.

[4] A. Bode und W. Händler. *Rechnerarchitektur II - Strukturen*. Wiesbaden: Springer Berlin Heidelberg, 1983. ISBN: 978-3-540-12267-8.

[5] F. Breitenecker. „Models, methods and experiments - a new structure for simulation systems.“ In: *Mathematics and Computers in Simulation* 34.3 (Sep. 1992), S. 231–260. DOI: 10.1016/0378-4754(92)90004-z.

[6] F. Breitenecker, G. Höfinger, T. Pawletta, S. Pawletta und R. Fink. „ARGESIM Benchmark on Parallel and Distributed Simulation“. In: *SNE Simulation News Europe* 17.1 (2007). ISSN 0929-2268, S. 53–56.

[7] F. Breitenecker, I. Husinsky und G. Schuster. „Comparison of Parallel Simulation Techniques“. In: *SNE Simulation News Europe* 4.10 (1994). ISSN 0929-2268, S. 21–22.

[8] R. Bryant. „Simulation on a Distributed System“. In: *1st International Conference on Distributed Systems*. IEEE, 1979, S. 544–552.

[9] A. W. Burks, H. H. Goldstine und J. von Neumann, Hrsg. *Preliminary Discussion of the Logical Design of an Electronic Computing Instrument*. Institute for Advanced Study, Princeton. 2ed Edition, 1946.

[10] N. Carriero und D. Gelernter. „How to Write Parallel Programs.“ In: *The MIT Press.* (1990). ISBN: 0-262-03171-X.

[11] CEA Wismar. *Classic-DEVSforMATLAB on GitHub*. https://github.com/cea-wismar/Classic-DEVSforMATLAB. 2023.

[12] CEA Wismar. *NSA-DEVSforMATLAB on GitHub*. https://github.com/cea-wismar/NSA-DEVSforMATLAB. 2022.

[13] CEA Wismar. *PDEVSforMATLAB on GitHub*. https://github.com/cea-wismar/PDEVSforMATLAB. 2022.

[14] CEA Wismar. *RPDEVSforMATLAB on GitHub*. https://github.com/cea-wismar/RPDEVSforMATLAB. 2022.

[15] F. E. Cellier und E. Kofman. *Continuous System Simulation*. Berlin, Heidelberg: Springer Science und Business Media, 2006. ISBN: 978-0-387-26102-7.

[16] K. Chandy und J. Misra. „Asynchronous Distributed Simulation via a Sequence of Parallel Computations“. In: *Communications of the ACM* 24 (Apr. 1981), S. 198–206. DOI: 10.1145/358598.358613.

[17] A. C. H. Chow, B. P. Zeigler und D. H. Kim. „Abstract simulator for the parallel DEVS formalism“. In: *Fifth Annual Conference on AI, and Planning in High Autonomy Systems*. 1994, S. 157–163. DOI: 10.1109/AIHAS.1994.390488.

[18] A. C. H. Chow. „Parallel DEVS: A Parallel, Hierarchical, Modular Modeling Formalism and its Distributed Simulators“. In: *Transactions of The Society for Computer Simulation International* 13.2 (1996), S. 55–67.

[19] A. C. H. Chow und B. P. Zeigler. „Parallel DEVS: a parallel, hierarchical, modular, modeling formalism“. In: *Proceedings of the 26th conference on Winter simulation, WSC 1994, Lake Buena Vista, FL, USA, December 11-14, 1994*. Hrsg. von D. A. Sadowski, A. F. Seila, M. S. Manivannan und J. D. Tew. ACM, 1994, S. 716–722. DOI: 10.1109/WSC.1994.717419. URL: https://doi.org/10.1109/WSC.1994.717419.

[20] E. R. Christensen. „Hierarchical optimistic distributed simulation: Combining DEVS and Time Warp.“ Dissertation. The University of Arizona., 1990. URL: http://hdl.handle.net/10150/185241.

[21] *Cluster Building Recipes*. 3.3. OpenHPC, a Linux Foundation Collaborative Project. 2025. URL: https://github.com/openhpc/ohpc/wiki/3.x.

[22] J. Dahmann, R. Fujimoto und R. Weatherly. „The Department Of Defense High Level Architecture“. In: *Proceedings of the 29th Conference on Winter Simulation* (Mai 2000). DOI: 10.1145/268437.268465.

[23] R. Fink. „Untersuchungen zur Parallelverarbeitung mit Wissenschaftlich-technischen Berechnungsumgebungen“. Diss. Rostock: Universität Rostock, 2007.

[24] R. Fink, S. Pawletta und T. Pawletta. „SCE based Parallel Processing and Applications in Simulation“. In: *Simulation Notes Europe* 16 (Sep. 2006), S. 37–50.

[25] M. J. Flynn. „Some Computer Organizations and Their Effectiveness.“ In: *IEEE Transactions on Computers* C-21.9 (1972), S. 948–960. DOI: 10.1109/TC.1972.5009071.

[26] M. J. Flynn. „Very High-Speed Computing Systems.“ In: *IEEE Xplore* (1966). doi: 10.1109/PROC.1966.5273, S. 519–527.

[27] M. J. Flynn und K. W. Rudd. „Parallel architectures.“ In: *ACM Comput. Surv.* 28.1 (1996), S. 67–70. DOI: 10.1145/234313.234345.

[28] H. Folkerts, T. Pawletta, C. Deatcu und B. Zeigler. „Automated, Reactive Pruning of System Entity Structures for Simulation Engineering“. In: *SpringSim'20*. Mai 2020, S. 12.

[29] I. Foster. *Designing and Building Parallel Programs - Concepts and Tools for Parallel Software Engineering*. Amsterdam: Addison-Wesley, 1995. ISBN: 978-0-201-57594-1.

[30] B. Freymann. „Aufgabenorientierte Multi-Robotersteuerungen auf Basis des SBC-Frameworks und DEVS“. Diss. TU Clausthal, 2022.

[31] R. Fujimoto, R. Bagrodia, R. Bryant, K. Chandy, D. Jefferson, J. Misra, D. Nicol und B. Unger. „Parallel discrete event simulation: The making of a field“. In: *Winter Simulation Conference*. Dez. 2017, S. 262–291. DOI: 10.1109/WSC.2017.8247793.

[32] R. M. Fujimoto. „Development of the parallel and distributed simulation field.“ In: *Simulation: Transactions of the Society for Modeling and Simulation International*. 100.12 (2024). doi: 10.1177/00375497241261407, S. 1197–1223.

[33] R. M. Fujimoto. *Parallel and Distributed Simulation Systems*. New York: Wiley, 2000. ISBN: 978-0-471-18383-9.

[34] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek und V. Sunderam. *Pvm 3 User's Guide And Reference Manual*. Techn. Ber. TN 37831. Oak Ridge National Laboratory, 1994.

[35] E. Glinsky und G. Wainer. „DEVStone: a Benchmarking Technique for Studying Performance of DEVS Modeling and Simulation Environments“. In: *Ninth IEEE International Symposium on Distributed Simulation and Real-Time Applications*. Nov. 2005, S. 265–272. ISBN: 0-7695-2462-1. DOI: 10.1109/DISTRA.2005.18.

[36] R. Goldblatt. *Lectures on the Hyperreals*. New York: Springer, 1998.

[37] O. Hagendorf und T. Pawletta. „A Framework for Simulation Based Structure and Parameter Optimization of Discrete Event Systems“. In: *Discrete-Event Modeling and Simulation: Theory and Applications*. Hrsg. von G. Wainer und P. Mosterman. ISBN-13: 9781420072334. USA: CRC Press Inc. of Tailor & Francis Group, 2011. Kap. 8, S. 199–222.

[38] O. Hagendorf. „Simulation Based Parameter and Structure Optimiesation of Discrete Event Systems“. Diss. Liverpool John Moores University, 2009.

[39] O. Hagendorf, T. Pawletta und R. Larek. „An approach to simulation-based parameter and structure optimization of MATLAB/Simulink models using evolutionary algorithms“. In: *Simulation - Transactions of The Society for Modeling and Simulation International* (Aug. 2013), S. 1115–1127. DOI: 10.1177/0037549713500066.

[40] P. Heinisch und K. Ostaszewski. „MatCL: A new easy-to use OpenCL toolbox for MathWorks Matlab.“ In: *IWOCL 18: Proceedings of the International Workshop on OpenCL*. doi: 10.1145/3204919.3204927. 2018.

[41] D. W. Hoffmann. *Grundlagen der Technischen Informatik* -. 6. aktualisierte Auflage. Carl Hanser Verlag GmbH Co KG, 2020. ISBN: 978-3446463141.

[42] D. Jammer, P. Junglas und S. Pawletta. „Solving ARGESIM Benchmark CP2 'Parallel and Distributed Simulation' with Open MPI and Matlab PCT - Lattice Boltzmann Simulation“. In: *SNE Simulation News Europe* 33.2 (2023). doi: 10.11128/sne.33.bncp2.10646, S. 93–100.

[43] D. Jammer, P. Junglas und S. Pawletta. „Solving ARGESIM Benchmark CP2 'Parallel and Distributed Simulation' with Open MPI/GSL and Matlab PCT - Monte Carlo and PDE Case Studies“. In: *SNE Simulation News Europe* 32.4 (2022). doi: 10.11128/sne.32.bncp2.10625, S. 211–220.

[44] D. Jammer, P. Junglas, T. Pawletta und S. Pawletta. „A Simulator for NSA-DEVS in Matlab“. In: *Proceedings of ASIM2022 – 26. Symposium Simulationstechnik*. Wien, 2022, S. 93–100. DOI: 0.11128/arep.20.a2005.

[45] D. Jammer, P. Junglas, T. Pawletta und S. Pawletta. „Implementing Standard Examples with NSA-DEVS“. In: *SNE Simulation News Europe* 32.4 (2022). DOI: 10.11128/sne.32.tn.10623, S. 195–202.

[46] D. Jammer, P. Junglas, T. Pawletta und S. Pawletta. „Modeling and Simulation of a Real-world Application using NSA-DEVS“. In: *SNE Simulation News Europe* 33.4 (2023). doi: 10.11128/sne.33.tn.10662, S. 149–156.

[47] D. Jammer, S. Pawletta, G. Kunert und T. Pawletta. „Beschleunigung eines Reinforcement-Learning-Algorithmus durch Parallelverarbeitung für Robotikanwendungen“. In: *Durak U. et al., editors. Proc. ASIM STS/GMMS Symposium*. Braunschweig, 2019, S. 49–52. DOI: 10.11128/arep.57.

[48] D. Jefferson. „Virtual Time.“ In: *ACM Trans. Program. Lang. Syst.* 7 (Juli 1985), S. 404–425. DOI: 10.1145/3916.3988.

[49] D. Jefferson, B. Beckman, F. Wieland, L. Blume, M. Loreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, L. V. Warren, J. Wedel, H. Younger und S. Bellenot. „Distributed Simulation and the Time Warp Operating System.“ In: *11th Symposium on Operating Systems Principles (SOSP)*. Nov. 1987, S. 77–93.

[50] P. Junglas. „NSA-DEVS: Combining Mealy Behaviour and Causality“. In: *SNE Simulation News Europe* 31.2 (2021). doi: 10.11128/sne.31.tn.10564, S. 73–80.

[51] R. Larek. „Ressourceneffiziente Auslegung von fertigungstechnischen Prozesskette durch Simulation und numerische Optimierung“. Diss. Universität Bremen, 2012.

[52] R. Larek, E. Brinksmeier, T. Pawletta und O. Hagendorf. „Model-Based Planning of Resource Efficient Process Chains Using System Entity Structures“. In: *Future Trends in Production Engineering*. Hrsg. von E. U. G. Schuh R. Neugebauer. Berlin: Springer Pub., Juni 2011, S. 361–371. ISBN: 978-3-642-24490-2. DOI: 10.1007/978-3-642-24491-9_36.

[53] S. Leye. „Toward guiding simulation experiments“. Diss. Rostock: Universität Rostock, 2013. DOI: 10.18453/rosdok_id00001365.

[54] S. Leye und A. M. Uhrmacher. „GUISE - a tool for GUIding simulation experiments“. In: *Proceedings - Winter Simulation Conference*. Dez. 2012.

[55] M. Livny. *The Study of Load Balancing Algorithms for Decentralized Distributed Processing Systems*. Techn. Ber. University of Wisconsin-Madison Department of Computer Sciences, 1984.

[56] O. Maler, Z. Manna und A. Pnueli. „From Timed to Hybrid Systems“. In: *Proceedings of the Real-Time: Theory in Practice, REX Workshop*. London: Springer-Verlag, 1992, S. 447–484.

[57] C. R. Martin, G. G. Trabes und G. A. Wainer. „A New Simulation Algorithm for PDEVS Models with Time Advance Zero“. In: *Proceedings of the Winter Simulation Conference*. WSC '20. Orlando, Florida: IEEE Press, 2021, S. 2208–2220. ISBN: 9781728194998.

[58] C. Märtin. *Rechnerarchitekturen: Cpus, Systeme, software-schnittstellen.* Fachbuchverlag Leipzig im Carl Hanser Verl, 2001.

[59] T. MathWorks. *Statistics and Machine Learning Toolbox Version 25.2*. 2025. URL: https://de.mathworks.com/help/pdf_doc/stats/stats.pdf.

[60] G. H. Mealy. „A method for synthesizing sequential circuits.“ In: *The Bell System Technical Journal* 34.5 (1955). doi: 10.1002/j.1538-7305.1955.tb03788.x, S. 1045–1079.

[61] H. Mehl. *Methoden verteilter Simulation.* Vieweg, 1994.

[62] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.1*. Nov. 2023. URL: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf.

[63] J. Misra. „Distributed Discrete-Event Simulation“. In: *Computing Surveys* 18 (März 1986), S. 39–65. DOI: 10.1145/6462.6485.

[64] C. Moler. „Why there isn't a parallel MATLAB“. In: *Matlab News and Notes*. Spring, 1995, S. 12.

[65] E. F. Moore. „Gedanken-Experiments on Sequential Machines“. In: *Automata Studies* 34 (1956). doi: 10.1515/9781400882618-006, S. 129–154. DOI: doi:10.1515/9781400882618-006.

[66] J. von Neumann. „First draft of a report on the EDVAC“. In: *IEEE Annals of the History of Computing* 15.4 (1993), S. 27–75. DOI: 10.1109/85.238389.

[67] J. Nutaro. „Toward a Theory of Superdense Time in Simulation Models“. In: *ACM Trans. Model. Comput. Simul.* 30.3 (2020). DOI: 10.1145/3379489.

[68] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*. 2024. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-6-0.pdf.

[69] S. Pawletta. „Erweiterung eines wissenschaftlich-technischen Berechnungs- und Visualisierungssystems zu einer Entwicklerumgebung für parallele Applikationen“. Diss. Rostock: Universität Rostock, 1998.

[70] T. Pawletta, U. Durak und A. Schmidt. „Modeling and Simulation of Versatile Technical Systems Using an Extended System Entity Structure/Model Base Infrastructure“. In: Jan. 2019, S. 393–418. ISBN: 9780128135433. DOI: 10.1016/B978-0-12-813543-3.00018-4.

[71] T. Pawletta, D. Jammer, P. Junglas und S. Pawletta. „Visual NSA-DEVS Modeling Using an Adapted DEVS Diagram“. In: *Proceedings of ASIM2024 – 27. Symposium Simulationstechnik*. München, 2024. DOI: 10.11128/arep.47.a4727.

[72] T. Pawletta, A. Schmidt und P. Junglas. „A Multimodeling Approach for the Simulation of Energy Consumption in Manufacturing“. In: *SNE Simulation News Europe* 27.2 (2017). DOI: 10.11128/sne.27.tn.10377, S. 115–124.

[73] D. Peng. „Reusing Simulation Experiments for Model Composition and Extension“. Diss. Rostock: Universität Rostock, 2017. DOI: 10.18453/rosdok_id00001864.

[74] F. Pichler und H. Schwärtzel. *CAST Methods in Modelling: Computer Aided Systems Theory for the Design of Intelligent Machines*. doi: 10.1007/978-3-642-95680-5. Springer Berlin Heidelberg, 1992. ISBN: 978-3-642-95682-9.

[75] F. J. Preyser, B. Heinzl und W. Kastner. „RPDEVS Abstract Simulator“. In: *SNE Simulation News Europe* 29.2 (2019). doi: 10.11128/sne.29.tn.10473, S. 79–84.

[76] F. J. Preyser, B. Heinzl und W. Kastner. „RPDEVS: Revising the Parallel Discrete Event System Specification“. In: *9th Vienna Int. Conf. Mathematical Modelling*. Wien, 2018, S. 242–247.

[77] F. J. Preyser, B. Heinzl, P. Raich und W. Kastner. „Towards Extending the Parallel-DEVS Formalism to Improve Component Modularity“. In: *Proc. of ASIM-Workshop STS/GMMS*. Lippstadt, 2016, S. 83–89. ISBN: 978-3-901608-48-3.

[78] M. Rabe, S. Spieckermann und S. Wenzel. *Verifikation und Validierung für die Simulation in Produktion und Logistik - Vorgehensmodelle und Techniken*. Berlin Heidelberg: Springer Science & Business Media, 2008. ISBN: 978-3-540-35281-5.

[79] J. Risco-Martín, S. Mittal, J. Fabero, M. Zapater und R. Hermida. „Reconsidering the performance of DEVS modeling and simulation environments using the DEVStone benchmark“. In: *SIMULATION* 93 (Feb. 2017), S. 459–476. DOI: 10.1177/0037549717690447.

[80] J. Rozenblit und B. Zeigler. „Representing and constructing system specifications using the system entity structure concepts“. In: Jan. 1993, S. 604–611. DOI: 10.1145/256563.256742.

[81] S. Rybacki, J. Himmelspach, E. Seib und A. M. Uhrmacher. „Using workflows in M&S software“. In: *Proceedings - Winter Simulation Conference*. Dez. 2010. DOI: 10.1109/WSC.2010.5679134.

[82] A. Schmidt. „Variantenmanagement in der Modellbildung und Simulation unter Verwendung des SES/MB Frameworks“. Diss. Universität Rostock, 2018. DOI: 10.18453/rosdok_id00002492.

[83] T. Schwatinski und T. Pawletta. „An Advanced Simulation Approach for Parallel DEVS with Ports“. In: *Proceedings of the 2010 Spring Simulation Multiconference*. SpringSim '10. Orlando, Florida: Society for Computer Simulation International, 2010, S. 147–154. ISBN: 9781450300698. DOI: 10.1145/1878537.1878690.

[84] K. Siebertz, D. v. Bebber und T. Hochkirchen. *Statistische Versuchsplanung - Design of Experiments (DoE)*. 2. Aufl. Berlin Heidelberg New York: Springer-Verlag, 2017. ISBN: 978-3-662-55743-3.

[85] D. B. Skillicorn und D. Talia. „Models and Languages for Parallel Computing.“ In: *ACM Computing Surveys.* 28 (1998).

[86] I. Sobol. „Sensitivity Estimates for Nonlinear Mathematical Models“. In: *Mathematical Modelling and Computational Experiments* 4 (1993), S. 407–414.

[87] H. S. Song und T. G. Kim. „DEVS Diagram Revised: A Structured Approach for DEVS Modeling“. In: *Journal of the Korea Society for Simulation* 21.2 (2012), S. 19–30.

[88] S. Straßburger. „Distributed simulation based on the high level architecture in civilian application domains“. ISBN 1-56555-218-0. Diss. Magdeburg: Universität Magdeburg, 2001.

[89] A. S. Tanenbaum und T. Austin. *Rechnerarchitektur.* Pearson, 2014.

[90] The MathWorks. *MATLAB Parallel Computing Toolbox Version 25.2*. 2025. URL: https://de.mathworks.com/help/pdf_doc/parallel-computing/parallel-computing.pdf.

[91] Top 500. *Top 500 The List*. 2025. URL: https://www.top500.org/.

[92] M. K. Traoré. „Easy DEVS“. In: *Proceedings of the 2007 spring simulation.* 2007, S. 214–216.

[93] T. Ungerer. *Datenflussrechner.* Springer Vieweg, 1993.

[94] T. Ungerer. *Parallelrechner und parallele Programmierung* -. Heidelberg: Spektrum Akademischer Verlag, 1997. ISBN: 978-3-827-40231-8.

[95] VDI 3633: Blatt 1. *Simulation von Logistik-, Materialfluss- und Produktionssysteme: Grundlagen*. Düsseldorf, 2013.

[96] G. Wainer, E. Glinsky und M. Gutierrez-Alcaraz. „Studying performance of DEVS modeling and simulation environments using the DEVStone benchmark“. In: *Simulation* 87 (Mai 2011), S. 555–580. DOI: 10.1177/0037549710395649.

[97] Y. H. Wang. „Discrete event simulation on a massively parallel computer.“ Dissertation. The University of Arizona., 1992. URL: http://hdl.handle.net/10150/185913.

[98] Y. H. Wang und B. P. Zeigler. „Extending the DEVS formalism for massively parallel simulation“. In: *Discret. Event Dyn. Syst.* 3.2-3 (1993), S. 193–218. DOI: 10.1007/BF01439849. URL: https://doi.org/10.1007/BF01439849.

[99] K. Watkins. *Discrete event simulation in C.* McGraw-Hill Book Company, 1993.

[100] R. Weicker. „Dhrystone: a synthetic systems programming benchmark“. In: *Commun. ACM* 27 (Okt. 1984), S. 1013–1030. DOI: 10.1145/358274.358283.

[101] J. White. „High-level framework for network-based resource sharing.“ In: *AFIPS 76: Proceedings of the June 7-10, 1976, national computer conference and exposition.* doi: 10.17487/RFC0707. 1976.

[102] M. Wijtvliet, H. Corporaal und A. Kumar. *Blocks - auf dem Weg zu Energieeffizienten, Grobkörnigen, Rekonfigurierbaren Architekturen (CGRA).* Springer Vieweg. in Springer Fachmedien Wiesbaden GmbH, 2023. ISBN: 978-3-031-36650-5.

[103] P. Wilsdorf, M. Dombrowsky, A. M. Uhrmacher, J. Zimmermann und U. v. Rienen. „Simulation Experiment Schemas –Beyond Tools and Simulation Approaches.“ In: *2019 Winter Simulation Conference (WSC).* 2019, S. 2783–2794. DOI: 10.1109/WSC40007.2019.9004710.

[104] B. P. Zeigler. *Multifacetted Modelling and Discrete Event Simulation.* Academic Press, 1984.

[105] B. P. Zeigler. *Theory of Modeling and Simulation.* 1st ed. New York: Wiley-Interscience, 1976.

[106] B. P. Zeigler. *Theory of Modeling and Simulation.* 2st ed. USA: Academic Press, Inc., 2000.

[107] B. P. Zeigler, A. Muzy und E. Kofman. *Theory of Modeling and Simulation: Discrete Event & Iterative System Computational Foundations.* 3rd. doi: 10.1016/C2016-0-03987-6. USA: Academic Press, Inc., 2018. ISBN: 978-0-12-813370-5.

Bildverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Abkürzungsverzeichnis

CA Central Arithmetical

CC Central Control

CD Compute Device

CEA Computational Engineering & Automation

CM Prozesssteuerungsmodell

CPU Central Processing Unit

CU Compute Unit

CUDA Compute Unified Device Architecture

DCT Distributed Computing Toolbox

DES Discrete Event System

DEVS Discrete Event System Specification

DGL Differenzialgleichung

DOE Design Of Experiments

DSP Digital Signal Processor

EC Experiment Control

ECS Erlangen Classification System

E-DEVS Extended Discrete Event System Specification

EF Experimental Frame

EF Elementary Effect

EG Entitäten-Generator

EIC External Input Coupling

EM Experimentmethode

EOC External Output Coupling

EPIC Explicitly Parallel Instruction Computing

EU Excecution Unit

FIFO First In - First Out

FPU Floating Point Unit

GA Genetic Algorithm

GM Grinding Machine

GPGPU General Purpose Computation on Graphics Processing Unit

GPU Graphics Processing Unit

GSL GNU Scientific Library

HF Hardening Furnace

HLA High Level Architecture

HPC High Performance Computing

I Input

IC Internal Coupling

IEEE Institute of Electrical and Electronics Engineers

IF Infinity Fabric

KPI Key Performance Indicator

LAN Local Area Network

LP logischer Prozess

M Memory

M&S Modellbildung und Simulation

MB Model Base

MEX Matlab-External-Interface

MF Materialflussmodell

MIMD Multiple Instruction Multiple Data

MISD Multiple Instruction Single Data

MPI Message Passing Interface

MPP Massively Parallel Processing System

MUS Model Under Study

NSA Non-Standard Analysis

NSA-DEVS Non-Standard Analysis Discrete Event System Specification

NUMA Non-Uniform Memory Access

O Output

ODE Ordinary Differential Equation

OpenCL Open Computing Language

OpenMP Open Multi-Processing

PCAM Partitionierung Communication Agglomeration Mapping

PCT Parallel Computing Toolbox

PDEVS Parallel Discrete Event System Specification

PE Processing Element

PHWT Private Hochschule für Wirtschaft und Technik

PM Physikmodell

POSIX Portable Operating System Interface

PVM Parallel Virtual Machine

QSS Quantized State System

RPC Remote Procedure Call

RPDEVS Revised Parallel Discrete Event System Specification

SA Simulated Annealing

SBE simulationsbasiertes Experiment

SCE Scientific Computing Environment

SES System Entity Structure

SM Simulationsmethode

SIMD Single Instruction Multiple Data

SISD Single Instruction Single Data

SM Streaming Multiprocessor

TF Tempering Furnace

TM Turning Machine

UMA Uniform Memory Access

VLIW Very Long Instruction Word

WAN Wide Area Network

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.
Die Stellen der Arbeit, die anderen Quellen im Wortlaut oder dem Sinn nach entnommen wurden, sind durch Angaben der Herkunft kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet.

Ich erkläre ferner, dass ich die vorliegende Arbeit in keinem anderen Prüfungsverfahren als Prüfungsarbeit eingereicht habe oder einreichen werde.

Die eingereichte schriftliche Fassung entspricht der eingereichten elektronischen Fassung.

Ort, Datum Unterschrift