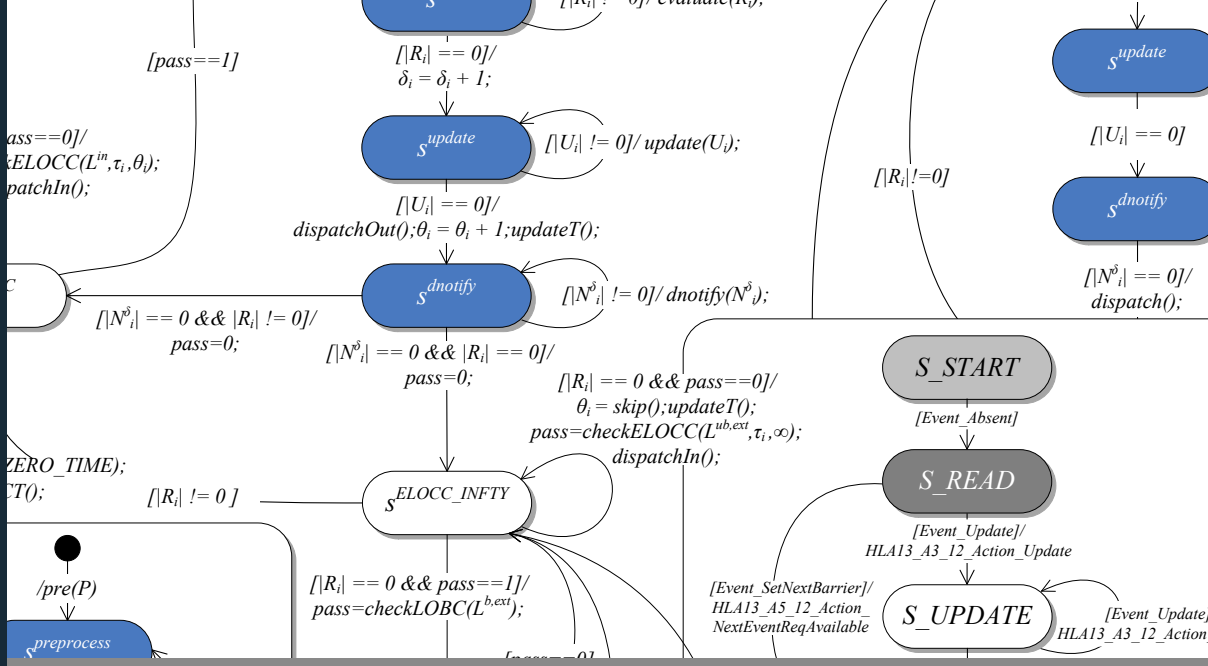


Christoph Roth

Parallele und kooperative Simulation für eingebettete Multiprozessorsysteme

Parallele und kooperative Simulation für eingebettete Multiprozessorsysteme

Zur Erlangung des akademischen Grades eines

DOKTOR-INGENIEURS

von der Fakultät für
Elektrotechnik und Informationstechnik
am Karlsruher Institut für Technologie (KIT)
genehmigte

DISSERTATION

von

Dipl.-Ing. Christoph Roth

geb. in Frankenthal (Pfalz)

Tag der mündlichen Prüfung:
16.12.2014

Hauptreferent: Prof. Dr.-Ing. Dr. h.c. Jürgen Becker
Korreferent: Prof. Dr. rer. nat. Bernhard Bauer

Vorwort

Diese Arbeit entstand während meiner Zeit als wissenschaftlicher Assistent am Institut für Technik der Informationsverarbeitung (ITIV) des Karlsruher Instituts für Technologie (KIT), welche sich nun ihrem Ende neigt. In den vergangenen Jahren habe ich viele wertvolle Erfahrungen sammeln dürfen, sowohl innerhalb meines Tätigkeitsbereichs am ITIV als auch im persönlichen und privaten Umfeld. An dieser Stelle möchte ich mich deswegen bei all jenen Menschen bedanken, die mich während dieses intensiven und bereichernden aber auch anstrengenden und bewegten Lebensabschnitts begleitet haben.

Einen besonderen Dank möchte ich Herrn Prof. Jürgen Becker dafür aussprechen, dass er mir die Möglichkeit eröffnet hat, das von mir gewählte Forschungsthema in einem großartigen Umfeld von hochmotivierten Kollegen zu erarbeiten. Des Weiteren danke ich Herrn Prof. Bernhard Bauer von der Universität Augsburg für die Übernahme des Korreferats sowie den Mitgliedern der Prüfungskommission, den Herren Prof. Hohmann, Prof. Lemmer und Prof. Trommer.

Ohne die Freunde, Kollegen und Studenten am ITIV und das dort existierende ganz spezielle Arbeitsumfeld, das von toller Zusammenarbeit, hoher Motivation, vielen guten Diskussionen und nicht zuletzt viel Humor geprägt war, wäre diese Arbeit nicht entstanden. Dafür möchte ich mich besonders bei meinen unmittelbaren „*Mitbewohnern*“ den „*Guys*“ Harald Bucher und Simon Reder sowie Mahtab Niknahad, dem restlichen „*Kellerkommando*“ und nicht zuletzt Florian Buciuman bedanken. Meinem langjährigen Wegbereiter und -begleiter Oliver Sander möchte ich einen speziellen Dank dafür aussprechen, dass er mir seit meiner Studentenzeit am ITIV mit Rat und Tat zur Seite stand. Dies hat maßgeblich zum Gelingen dieser Arbeit beigetragen.

Meiner gesamten Familie danke herzlich für die große Unterstützung während der vergangenen Jahre abseits des beruflichen Alltags. Speziell meinen Eltern bin ich sehr dankbar dafür, dass sie mir den Weg zu meiner Ausbildung und meiner Promotion erst möglich gemacht haben. Mein allergrößter Dank aber gilt meiner geliebten Frau Lena und meiner Tochter Luise, die nicht enden wollende Geduld hatten, wenn der Arbeitstag schon wieder länger war, das Wochenende dran glauben musste oder ich meine Gedanken mal wieder nicht von den The-

men lösen konnte, die mich gerade beschäftigten. Ich darf mich außerordentlich glücklich schätzen, dass es euch gibt.

Fußgönheim, im Februar 2015
Christoph Roth

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation

Systeme der *Informations- und Kommunikationstechnologie (IKT)* sind heute integraler Bestandteil des täglichen Lebens. Dies ist nicht zuletzt auf den andauernden Trend zur Miniaturisierung im Bereich der Halbleiterentwicklung zurückzuführen [202, 231]. Die Bedeutung der IKT wird auch in Zukunft immer weiter zunehmen, was durch aktuelle Forschungstrends verdeutlicht wird, welche durch Begriffe wie *Ambient Intelligence*, *Ubiquitous Computing* oder *Pervasive Computing* charakterisiert werden können [264, 127, 31]: Zukünftige Applikationen sollen eine gewisse Umgebungsintelligenz (Ambient Intelligence) zur Unterstützung des Menschen im Alltag bereitstellen. Zur Realisierung dieser Applikationen muss Information jederzeit und allgegenwärtig verfügbar sein (Ubiquitous). Dies geschieht unter Ausnutzung und Vernetzung bereits vorhandener Technologien (Pervasive). Zu den Bereichen, die durch die neuen Ansätze im Umfeld der IKT maßgeblich beeinflusst werden, gehören u.a. die Automobilelektronik, die Avionik, der Schienenverkehr, die Telekommunikation, die Automatisierungstechnik oder die Medizintechnik.

Zwei bereits existierende grundlegende Basistechnologien zu Realisierung zukünftiger IKT Systeme bilden eingebettete Systeme und Kommunikationstechnologien [174, 195]. In [248] werden eingebettete Systeme definiert als

> „*heterogene technische Systeme, die sich durch verschiedenartige Komponenten und Interaktionen auszeichnen, die auf einen ganz bestimmten Anwendungsbereich zugeschnitten und die in einem technischen Kontext eingebettet sind*“.

In der Regel müssen sie eine Reihe von Anforderungen wie Echtzeitfähigkeit, Zuverlässigkeit, Robustheit und/oder Effizienz erfüllen. Ein eingebettetes System besteht im Kern typischerweise aus Hardware- und Softwarekomponenten. Aufgrund der Notwendigkeit zur Interaktion mit der physikalischen Umwelt sind eingebettete Systeme mit Sensoren und/oder Aktoren ausgestattet. Kommunikationstechnologien schaffen schließlich die Basis dafür, dass Interaktionen eines eingebetteten Systems mit der Umwelt auch über einen räumlich beschränkten Horizont hinaus möglich werden.

Die Umsetzung der oben genannten neuartigen Konzepte trägt zu einer enormen Steigerung der Vielfalt und Komplexität von eingebetteten Systemen bei. Diese Komplexität äußert sich insbesondere in

1. einer gesteigerten Rechenkomplexität zukünftiger Anwendungen. In Verbindung mit der hohen Anzahl neuer Funktionen resultiert dies in erhöhten Anforderungen an die verfügbare Performanz.
2. einer erhöhten Interaktion von Anwendungen und zugrunde liegenden Systemkomponenten nicht nur mit der heterogenen physischen Umwelt, sondern auch mit einer weitreichenden und vernetzten IKT Infrastruktur.

Wegen der Einkopplung der physikalischen Umwelt in eine umfangreich vernetzte IKT Infrastruktur wird die beschriebene Art des resultierenden Gesamtsystems heutzutage oft auch allgemein unter dem Oberbegriff *Cyber-Physical System (CPS)* [246, 174, 32] oder *Internet of Things* [38, 197] zusammengefasst.

Durch die Komplexitätssteigerung von Funktionen, Anwendungen und Systemen entstehen unmittelbar neue Herausforderungen an Entwicklungsprozesse. Konkrete Herausforderungen im Bereich der Simulation, die aktuell entweder gar nicht oder nicht im notwendigen Maße berücksichtigt werden, seien im Folgenden kurz skizziert.

1.1.1. Beschleunigung durch parallele Simulation

Betrachtet man ausschließlich den Entwicklungsprozess für *Hardware/Software (HW/SW)* Systeme, den Teil eines eingebetteten Systems, der für die Informationsverarbeitung verantwortlich ist, so basiert dieser schon seit den 1980er Jahren auf automatisierten rechnergestützten Entwurfsmethoden. Nur dadurch ist es möglich, solche Systeme in akzeptabler Zeit und mit annehmbaren Kosten zu entwickeln [115, 172, 248].

Die steigenden Anforderungen an die verfügbare Performanz in Verbindung mit der Notwendigkeit zur energetischen Effizienz resultieren nun darin, dass solche eingebetteten HW/SW Systeme immer öfter als *Multiprocessor System-on-Chip (MPSoC)* realisiert werden [56]. MPSoCs integrieren immer mehr homogene oder heterogene Rechenkerne auf einem einzigen Chip. Dies wiederum hat einen wachsenden Umfang von Modellen und Spezifikationen zur Folge, die im Verlauf des Entwicklungsprozesses erstellt werden. Umfangreichere Spezifikationen erzeugen Mehraufwand für die Verifikation und infolgedessen höhere Kosten. Beispielsweise haben stetig wachsende Simulationsmodelle immer längere Laufzeiten einer simulationsbasierten Verifikation zur Folge [180].

Ein oft gewählter Ansatz zur Beschleunigung von MPSoC Simulationen ist es, die Rechenkomplexität von vollständig zyklenakkuraten Simulationsmodellen

durch Abstraktion zu verringern. Dadurch sind beträchtliche Performanzsteigerungen möglich. Ein solche Abstraktion hat jedoch den Nachteil, dass Modelle je nach Abstraktionsgrad nur für bestimmte Anwendungsfälle geeignet sind. Ein zur Abstraktion alternativer Lösungsansatz ist deswegen die parallele Simulation.

Im Bereich der Mehrzweck *Central Processing Units (CPUs)* hat sich der Paradigmenwechsel von einkernigen Prozessoren hin zu Multiprozessoren bereits im Jahr 2004 vollzogen [243]. Die Ursache war das Überschreiten der oft als „*Power Wall*" bezeichneten Barriere (siehe Abb. 1.1). Diese führt dazu, dass Performanzsteigerungen nicht mehr durch eine reine Skalierung der Taktfrequenz, sondern nur noch durch Architekturtechniken wie Hyperthreading, Caching oder eben dem Übergang zu Multiprozessoren erreicht werden können [243]. Aktuell geht der Trend im Mehrzweckbereich bereits in Richtung Manycore Prozessoren, welche mehrere zehn oder gar hundert Rechenkerne integrieren. Ein Beispiel aus der Forschung ist der im Jahr 2009 angekündigte sog. *Single-chip Cloud Computer (SCC)* [144] mit 48 Kernen. Ein seit dem Jahr 2011 kommerziell erhältlicher Manycore Prozessor ist die Intel *Many-Integrated-Core (MIC)* Architektur mit aktuell bis zu 72 Kernen [91].

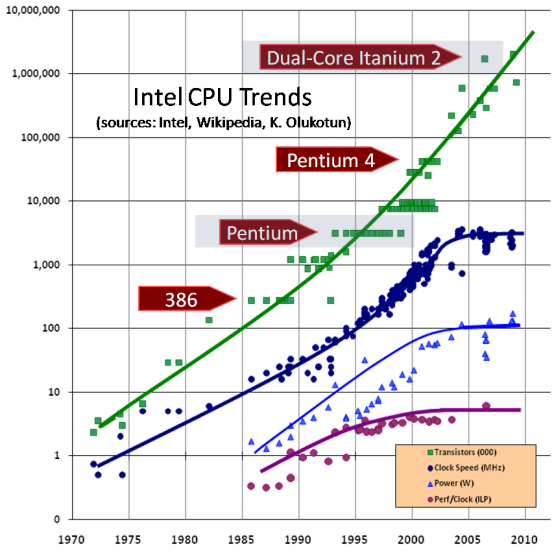


Abbildung 1.1.: Intel CPUs zwischen 1970 und 2010 (Quelle: [243])

Um Multicore oder Manycore CPUs zur Beschleunigung einer MPSoC Simulation nutzen zu können, müssen existierende Simulatoren für die Parallelverarbeitung tauglich gemacht werden. Für einschlägige Simulatoren wie beispielsweise

den SystemC Simulator [27, 1] existiert aktuell noch keine Unterstützung für parallele Ausführung.

Zukünftige Entwicklungsprozesse sollten folglich neue Methoden nutzen, welche die Möglichkeit zur parallelen Simulation komplexer Simulationsmodelle eröffnen. Die Kombination von Parallelisierung und Abstraktion kann ein adäquates Mittel sein, um einen besseren Trade-off zwischen hoher Genauigkeit und hoher Performanz zu erzielen. Aus Gründen der Komplexitätsreduktion sollte der Prozess der Parallelisierung eines Modells zugleich möglichst automatisiert und versteckt vor dem Anwender erfolgen.

1.1.2. Interdisziplinäre Entwicklung durch kooperative Simulation

Durch den Trend in Richtung MPSoCs in Kombination mit dem hohen Vernetzungsgrad lassen sich zukünftige Anwendungen im Bereich der eingebetteten Systeme oft nicht mehr eindeutig bestimmten HW/SW Komponenten zuordnen. Durch die Integration von (vormals vollständig geschlossenen Netzwerken aus) Teilsystemen in eine prinzipiell offene IKT Infrastruktur, entstehen deswegen zusätzliche Herausforderungen bzgl. der Erfüllung von Anforderungen wie Echtzeitfähigkeit, Zuverlässigkeit und Sicherheit.

Um die Erfüllung solcher Anforderungen zu verifizieren, genügt es nicht mehr, Teilsysteme als abgeschlossene Einheiten zu betrachten und erst in einer späten Phase des Entwicklungsprozesses miteinander zu integrieren. Vielmehr muss die Entwicklung eines Teilsystems bereits in frühen Phasen des Entwurfsprozesses und durchgängig bis zur Implementierungsphase unter Berücksichtigung von Randbedingungen erfolgen, die durch andere Teilsysteme erzeugt werden [20]. Nur auf diese Weise können ein Fehlschlagen einer späten Integration sowie teure Iterationen im Entwurfsprozess vermieden werden.

Für die Verifikation bedeutet dies, dass der wechselseitige Einfluss verschiedener Teilsysteme sowie der physikalischen Umwelt so früh wie möglich durch kooperative Simulation (*Co-Simulation*) von disziplinspezifischen Modellen berücksichtigt werden sollte. Als Voraussetzung dafür müssen Schnittstellen zur Kopplung der Modelle oder der zugrundeliegenden Simulationswerkzeuge existieren. Sollen Modelle wiederverwendet werden, so müssen diese Schnittstellen u.U. neu entwickelt oder adaptiert werden. Neben dem hohen Entwicklungsaufwand ist ein weiteres Problem, dass das resultierende heterogene Gesamtsimulationsmodell aufgrund unstrukturierter Komposition ein unerwartetes emergentes Verhalten aufweisen kann [102]. Eine alternative Lösung sind heterogene Simulationswerkzeuge, welche bereits nativ eine Simulation von heterogenen

Modellen unterstützen. Diese Werkzeuge stehen allerdings dem Wiederverwendungsaspekt entgegen.

Zukünftige Entwicklungsprozesse sollten der beschriebenen Problematik daher durch den Einsatz neuer Methoden zur heterogenen Co-Simulation sowie zur Verbesserung der Interoperabilität zwischen existierenden heterogenen Modellen entgegenwirken, welche eine möglichst schnelle aber strukturierte Integration von Simulationsmodellen ermöglichen.

1.2. Zielsetzung der Dissertation

Die vorliegende Dissertation widmet sich der Entwicklung von Lösungsansätzen, um den zuvor skizzierten Herausforderungen im Bereich der Simulation eingebetteter Systeme zu begegnen. Dabei sollen möglichst automatisierte Methoden bereitgestellt werden, die einen simulationsbasierten Verifikationsprozess für eingebettete Multiprozessorsysteme durch Beschleunigung und Interoperabilität unterstützen.

Bei den im Kontext der Simulationsbeschleunigung betrachteten Techniken und Methoden steht die Kompatibilität für zukünftige Manycore Prozessoren im Vordergrund. Dies ist ein Hauptunterscheidungsmerkmal zu anderen existierenden Forschungsansätzen im Bereich der parallelen MPSoC Simulation. Folgende Fragestellungen werden dabei erörtert:

- Wie können zukünftige Manycore Prozessoren wie der Single-chip Cloud Computer für die parallele Simulation und insbesondere die Beschleunigung von zyklenakkuraten und zyklenapproximativen MPSoC Simulationen nutzbar gemacht werden?
- Welche Charakteristika muss ein paralleler Simulator besitzen, um Anforderungen, wie Skalierbarkeit, Erweiterbarkeit, Portierbarkeit, Handhabbarkeit und Kompatibilität zu relevanten Modellierungsstilen zu erfüllen?
- Wie sehen Ansätze für eine automatisierte Werkzeugkette aus, um den Parallelisierungsprozess weitgehend bis vollständig vor dem Anwender zu verstecken?
- Welche Möglichkeiten bestehen, um bei einer MPSoC Simulation eine Performanzsteigerung durch Parallelisierung und Abstraktion zu erreichen aber gleichzeitig den Genauigkeitsverlust gering zu halten?

Um die Herausforderungen bzgl. der Interoperabilität von Simulationsmodellen in zukünftigen Entwicklungsprozessen weiter zu konkretisieren, werden sie in dieser Arbeit am Beispiel der Entwicklung zukünftiger automobiler *elektrisch/-*

elektronischer (E/E) Architekturen betrachtet. Dabei wird auf folgende Kernfragestellungen eingegangen:

- Welche Implikationen von neuartigen Technologien auf zukünftige E/E Architekturen sind zu erwarten? Welche Anforderungen lassen sich daraus für zukünftige Entwicklungsprozesse im Automobilbereich ableiten?
- Wie sieht eine Simulatorarchitektur zur Co-Simulation aus, die Anforderungen wie bessere Interoperabilität, heterogene Modellierung, strukturierte Komposition, Wiederverwendbarkeit und Erweiterbarkeit erfüllt?
- Welche Möglichkeiten bestehen, um den Prozess der Kopplung heterogener Simulationsmodelle zu beschleunigen, die Handhabbarkeit zu verbessern und die Gefahr vor unerwartetem Verhalten zu reduzieren?

Zur Beantwortung der genannten Fragestellungen wurden verschiedene Methoden und Werkzeuge entwickelt, die im Folgenden vorgestellt werden. Dies beinhaltet die Beschreibung und Analyse zugrundeliegender Konzepte sowie die quantitative Bewertung und kritische Diskussion von Grenzen und möglicher Ansätze zur weiteren Optimierung.

1.3. Aufbau der Arbeit

Kapitel 2 gibt einen kurzen Überblick über die Grundlagen dieser Arbeit. Das Kapitel beginnt mit einem Überblick über Entwurfsprozesse für eingebettete Systeme. Anschließend werden Grundbegriffe der Modellbildung und Simulation mit einem speziellen Fokus auf (parallelen) diskreten ereignisbasierten Berechnungsmodellen erläutert. Darauf aufbauend werden einschlägige Sprachen und Werkzeuge, insbesondere SystemC [27] und Ptolemy II [102], vorgestellt. Das Kapitel schließt mit einer Zusammenfassung existierender Techniken im Bereich Prozessorarchitekturen, einer Beschreibung der Architektur des SCC sowie einer Klärung grundlegender Begriffe der Performanzanalyse.

Kapitel 3 beschreibt den Stand von Forschung und Technik. Im Bereich der parallelen Simulation von Multiprozessoren liegt der Schwerpunkt zunächst auf proprietären Forschungsansätzen und bereits existierenden kommerziellen Lösungen. Danach werden Ansätze betrachtet, welche SystemC als Grundlage verwenden. Anschließend wird der Stand von Forschung und Technik im Bereich heterogener Co-Simulation, insbesondere auch im Hinblick auf Methoden zur Verbesserung der Interoperabilität existierender Simulationswerkzeuge, diskutiert.

In Kapitel 4 wird zunächst eine allgemeine Methodik für die parallele SystemC-basierte Simulation von eingebetteten MPSoCs auf Manycore Architekturen her-

geleitet. Anschließend werden Strategien zur Parallelisierung des SystemC Kernels analysiert und klassifiziert. Darauf aufbauend wird die Umsetzung dreier ausgewählter Strategien zur zyklenakkuraten Simulation auf dem sog. *Register Transfer Level (RTL)* beschrieben. Zwei der drei Strategien werden durch eine teil- bzw. vollautomatisierte Werkzeugkette ergänzt. Darauf aufbauend wird eine Modellierungsstrategie auf dem sog. *Transaction Level (TL)* entwickelt, welche es gestattet, die parallele Ausführung durch gezielte Abstraktion zusätzlich zu beschleunigen. Jede der genannten Strategien wird mit Hilfe verschiedener Fallstudien hinsichtlich ihrer Leistungsfähigkeit und der Erfüllung an sie gestellter Anforderungen bewertet.

Kapitel 5 beschreibt einen methodischen Ansatz, welcher eine simulationsbasierte Verifikation anhand interdisziplinärer kooperativer Simulation möglich macht. Die Notwendigkeit dazu wird zunächst am Beispiel des Entwicklungsprozesses automobiler E/E Architekturen verdeutlicht. Aus dem Beispiel werden Anforderungen an eine geeignete Simulationsumgebung für die interdisziplinäre kooperative Simulation abgeleitet. Darauf basierend wird ein Konzept für eine Simulatorarchitektur sowie eine Methode für die Anwendung der Simulatorarchitektur entwickelt. Realisierbarkeit und Anwendbarkeit der Methode werden schließlich anhand einer Werkzeugkette und verschiedener Fallstudien demonstriert.

In Kapitel 6 werden die entwickelten Ansätze in einer abschließenden Betrachtung noch einmal zusammengefasst. Ein kurzer Ausblick auf mögliche Weiterentwicklungen, Optimierungen und zukünftiges Forschungspotential vervollständigen die Arbeit.

2. Grundlagen

2.1. Entwurf eingebetteter elektronischer Systeme

Die kontinuierliche Weiterentwicklung der Halbleitertechnologie ist die Grundlage für die Entwicklung immer komplexerer eingebetteter Systeme. Die Beherrschung dieser stetig steigenden Komplexität stellt eine große Herausforderung während des Entwurfsprozesses dar. Tatsächlich ist zu beobachten, dass die Lücke zwischen den verfügbaren Basistechnologien und der Produktivität der Entwurfsverfahren für digitale Hardware/Software Systeme, die die Basistechnologien ausnutzen, immer größer wird [99]. In [99] wird diese Lücke als *System Design Gap* bezeichnet.

Das stetig wachsende System Design Gap ist auf die Notwendigkeit sog. *Hardware-dependent Software (HdS)* [148, 99] und den mit der Entwicklung von HdS verbundenen zusätzlichen Anstieg im Entwicklungsaufwand zurückzuführen. Im Allgemeinen dient HdS der Abstraktion verfügbarer Hardwareressourcen durch eine zusätzliche Softwareschicht.

Darüber hinaus wird das System Design Gap dadurch verstärkt, dass aktuelle und zukünftige *System-on-Chip (SoC)* Lösungen immer mehr Rechenkerne auf einem einzigen Chip integrieren. Solche Multi- oder Manycore SoCs können von unterschiedlichster Ausprägung sein (homogen, heterogen, generisch, applikationsspezifisch, ...) [180]. Zusätzlich steigt die Anzahl an Anforderungen hinsichtlich unterstützter Applikationen, Funktionen, Zuverlässigkeit und Echtzeitfähigkeit. HdS muss dieser Vielfalt möglicher Hardwarearchitekturen und Funktionen gerecht werden.

Aus dieser Beschreibung wird deutlich, dass ein hoher Bedarf an neuen Ansätzen in Form von neuen Entwurfsmethoden existiert, die zur Verbesserung von Entwicklungsprozessen im Bereich eingebetteter elektronischer Systeme beitragen. Das Ziel einer solchen Entwurfsmethode kann wie folgt formuliert werden (vgl. [107]):

> *„Minimierung der Entwicklungszeit sowie Entwicklungs- und Produktionskosten nach Maßgabe der Anforderungen an Performanz und Funktionalität des Systems“.*

Zur Erreichung dieses Ziels stellt eine Entwurfsmethode typischerweise Werkzeuge in Form von *Electronic Design Automation (EDA)* Tools zur Verfügung, die die einzelnen Schritte der Entwurfsmethode durch Automatisierung unterstützen. Solche automatisierten Verfahren ermöglichen kürzere Entwurfszyklen und reduzieren Entwurfsfehler durch Verbesserung von Verifikations-, Validierungs- und Explorationsprozessen [130]. All diese Teilaspekte kommen wiederum einer Minimierung der Entwicklungszeit zugute. Im Folgenden werden die Grundlagen der heutzutage angewendeten Entwurfsmethoden erläutert.

2.1.1. Klassifikation von Entwurfsmethoden

Unterschiedliche Entwurfsmethoden im Bereich elektronischer Systeme lassen sich anhand des von Gajski und Kuhn im Jahr 1983 entwickelten Y-Diagramms [116] klassifizieren (siehe Abb. 2.1). Das Y-Diagramm besteht aus drei Achsen, die verhaltens-, struktur- und physikorientierte Aspekte eines Systems repräsentieren. Aus der Perspektive des Verhaltens wird ein System als Blackbox betrachtet und anhand der Wirkung von Eingängen auf Ausgänge über der Zeit spezifiziert. Aus struktureller Sicht wird ein System als eine Kombination von Subsystemen und Verbindungen zwischen Subsystemen spezifiziert. Mit dem physikalischen Aspekt ist die Beschreibung der tatsächlichen räumlichen Beschaffenheit gemeint.

Alle drei Achsen sind in unterschiedliche Abstraktionsebenen unterteilt. Diese sind durch konzentrische Kreise dargestellt. Abb. 2.1 beinhaltet die Systemebene, die Prozessorebene, die Logikebene und die Schaltkreisebene. Die Namen leiten sich von den Komponenten ab, die typischerweise auf der entsprechenden Abstraktionsebene im Fokus der Entwicklung stehen. Eine Entwurfsmethode entspricht einem Pfad im Y-Diagramm, der sich immer mehr dem Zentrum nähert. Er beginnt typischerweise bei der Spezifikation des Verhaltens und bewegt sich von dort in Richtung Spezifikation der Geometrie.

2.1.1.1. Synthese

Ein essentieller Teil einer Entwurfsmethode ist die Synthese. Der Begriff der Synthese lässt sich im Y-Diagramm auf jeder Abstraktionsebene als eine „*Konvertierung einer gegebenen Verhaltensspezifikation in eine strukturelle Spezifikation*“ [115] darstellen. Jede Teilkomponente einer strukturellen Spezifikation, kann auf der darunterliegenden Abstraktionsebene wieder aus der Perspektive des Verhaltens der Struktur oder der Geometrie betrachtet werden. Eine Synthese kann manuell erfolgen oder durch automatisierte Syntheseverfahren unterstützt werden. Automatische Verfahren setzen voraus, dass das Verhalten auf Basis sog. Berechnungsmodelle (vgl. Abschnitt 2.2.3) eindeutig spezifiziert ist.

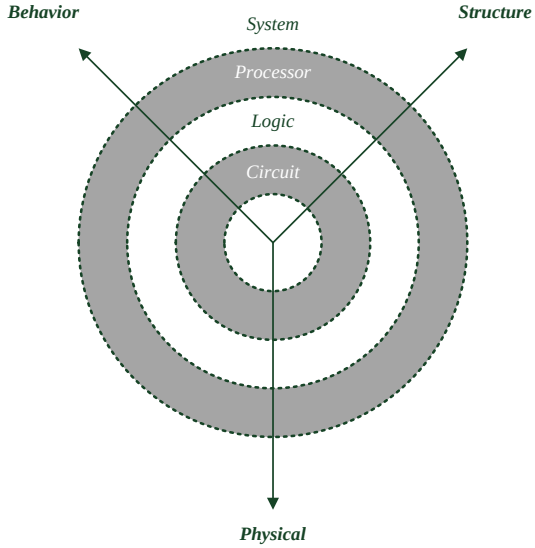


Abbildung 2.1.: Y-Diagramm nach Gajski und Kuhn [115]

Abhängig davon, auf welcher Abstraktionsebene eine Synthese ansetzt, kann sie in unterschiedliche Teilschritte untergliedert werden. Der typische Schritt der sog. Systemsynthese ist die Abbildung (engl. Mapping) einer Verhaltensspezifikation in Form einer Funktion oder Applikation auf eine strukturelle Spezifikation in Form eines beliebigen Netzwerks von Prozessoren. Typische Teilschritte eine Synthese auf den darunterliegenden Ebenen sind I) die Allokation II) die Ablaufplanung und III) die Bindung [248, 115].

2.1.1.2. Beispiele für Entwurfsmethoden

Noch bis in die späten 1990er Jahre begannen Entwurfsmethoden für elektronische Systeme unmittelbar mit der Spezifikation des Verhaltens auf dem RTL. Im Gegensatz dazu lassen sich Entwurfsmethoden heutzutage mit der Umschreibung „*Specify, Explore-and-Refine*“ [115] charakterisieren. Solche Ansätze werden auch als *System-Level Design (SLD)* Methoden bezeichnet. Nur durch die Anhebung der Abstraktionsebene über RTL hinaus auf das sog. *Electronic System Level (ESL)*, ist es möglich, die steigende Komplexität zukünftiger eingebetteter Systeme in der Phase des Entwurfs in den Griff zu bekommen [228]. Gajski nennt und erläutert in [115] drei Basismethodiken, die grundsätzlich auf jeder Abstraktionsebene (ESL, RTL, etc.) angewendet werden können:

1. **Bottom-up Methodik**: In einer Bottom-up Methode existiert auf einer bestimmten Abstraktionsebene eine Bibliothek von Komponenten (inkl. der Spezifikation von Verhalten, Struktur und Geometrie). Aus Komponenten dieser Abstraktionsebene werden sukzessive Komponenten der nächsthöheren Abstraktionsebene erstellt (Komposition) und hinsichtlich der drei Entwurfsaspekte aus Abb. 2.1 spezifiziert. Diese können dann ebenfalls wieder in einer Bibliothek hinterlegt werden. Ein Vorteil ist die klare Trennung von Abstraktionsebenen durch Bibliotheken. der Hauptnachteil ist, dass für die Komposition einer Komponente auf einer höheren Abstraktionsebene alle notwendigen Komponenten auf den niedrigeren Abstraktionsebenen bereits vorhanden sein müssen.
2. **Top-down Methodik**: Eine Top-down Methode beginnt auf einer hohen Abstraktionsebene, üblicherweise bei der Spezifikation des Verhaltens. Die abstrakte Verhaltensspezifikation wird dann sukzessive in Teilkomponenten niedrigerer Abstraktionsebenen zerlegt (Dekomposition) und bzgl. Verhalten, Struktur und letztlich Geometrie detailliert (Verfeinerung). Die Notwendigkeit von Bibliothekskomponenten niedrigerer Abstraktionsebenen wird dadurch vermieden. Umgekehrt ist man während des Entwurfsprozesses auf Schätzungen der Entwurfsqualität auf niedrigeren Abstraktionsebenen angewiesen.
3. **Meet-In-The-Middle Methodik**: Diese kombiniert die beiden zuvor genannten Methoden und nutzt so die Vorteile beider Ansätze. Typischerweise wird auf höheren Abstraktionsebenen eine Spezifikation in einem ein Top-down Verfahren dekomponiert. Auf niedrigeren Abstraktionsebenen werden Komponenten in ein Bottom-up Verfahren komponiert. Oft liegen beispielsweise Komponenten in einer RTL Bibliothek vor, die zuvor in einem Bottom-up Prozess generiert wurden. In einem Top-down Prozess wird dann eine abstrakte Verhaltensspezifikation sukzessive in Richtung einer Architektur verfeinert, die ausschließlich aus Komponenten der RTL Bibliothek besteht. Die Entwurfsqualität wird mit Hilfe von Metriken geschätzt, die aus den Bibliotheken extrahiert werden können.

2.1.1.3. Plattformbasierte Entwurfsmethodik auf Systemebene

Die plattformbasierte Entwurfsmethodik (engl. *Platform-based Design (PBD)*) auf Systemebene ist ein Sonderfall der Meet-In-The-Middle Methodik, welcher stark produktorientiert ist und an spezifische industrielle Anforderungen angepasst werden kann [115][107][120]. PBD auf Systemebene reduziert die Komplexität während des Entwurfsprozesses durch die Separation von Belangen (z.B. Separation von Funktion und Architektur oder Berechnung und Kommunikation)

[163][224] und die Einschränkung des theoretisch unendlich große Entwurfsraum auf eine handhabbare Größe.

Die sog. Systemplattform besteht aus einer Hardware- und einer Softwareplattform. Sie wurde zuvor in einem Bottom-up Prozess entwickelt. Die Hardwareplattform umfasst eine begrenzte Anzahl an Hardwarekomponenten, die bis zu einem gewissen Grad (Typ, Anzahl, Verbindung, etc.) konfigurierbar sind. Die Softwareplattform erfüllt den Zweck der weiter oben bereits erwähnten HdS und abstrahiert von der darunterliegenden Hardware. Typische Komponenten der Softwareplattform sind z.B. ein *Real-time Operating System (RTOS)* oder Treibersoftware. Für größtmögliche Flexibilität ist sie meist in Form einer Schichtenarchitektur realisiert. Die Funktionalität der Systemplattform wird in Form eines *Application Programming Interface (API)* genannt *Plattform API* zur Verfügung gestellt (vgl. [163][230]).

In einer PBD Methodik auf Systemebene wird, ausgehend von einer abstrakten Verhaltensspezifikation (Application Instance), die Implementierung in einem iterativen Prozess entwickelt. Der essentielle Schritt ist die Abbildung der Verhaltensspezifikation auf eine bestimmte Plattforminstanz (Platform Instance). Diese ist eine Konfiguration aus der Menge aller möglichen Konfigurationen der Systemplattform (Architectural Space). Die Kosten einer Entwurfsentscheidung basierend auf einer Abbildung werden anhand von Kostenmodellen geschätzt und bewertet. Im Rahmen des Abbildungsprozesses wird die Spezifikation schrittweise in Richtung Implementierung verfeinert. Die Exploration findet insgesamt auf Abstraktionsebenen statt, die sich irgendwo zwischen vollständig abstrakter Spezifikation und finaler Implementierung bewegen.

Bis zu einer gewissen Ebene entspricht der Prozess der Abbildung und Verfeinerung einer semi-automatischen Synthese. Dabei spielt der Einsatz von Simulationen zur Kostenschätzung oft eine dominante Rolle. Auch die Korrektheit eines durchgeführten Syntheseschrittes bzw. die Erfüllung gegebener funktionaler und nicht-funktionaler Anforderungen muss verifiziert werden. Der iterative Prozess endet auf einer gewissen Ebene. Ab diesem Punkt werden typischerweise vollständig automatische Syntheseverfahren für die Generierung der Hardware- und Softwareplattform verwendet. Abb. 2.2 illustriert den beschriebenen typischen Entwurfsfluss. Eine Verifikation erfolgt für gewöhnlich nach jedem Syntheseschritt, insbesondere auch nach Erreichen der finalen Implementierung.

2.1.2. Validierung und Verifikation

In einem kompletten Entwicklungsprozess kann im Allgemeinen zwischen dem Prozess der Validierung und dem Prozess der Verifikation unterschieden werden. Der *IEEE* Standard 1012-1012 [28] definiert Validierung wie folgt:

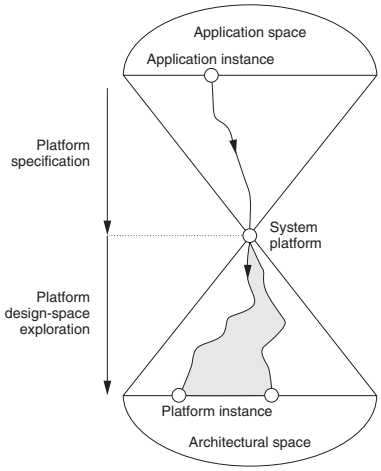


Abbildung 2.2.: Plattformbasierter Entwurf (Quelle: [230])

> *„The process of evaluating a system or component during or at the end of the development to determine whether it satisfies specified requirements. The process of providing evidence that the system, software, or hardware and its associated products ... solve the right problem ..., and satisfy intended use and user needs.“*

Bei der Validierung wird die Erfüllung von Anforderungen überprüft, die der Anwender oder Kunde an das zu entwickelnde System stellt. Es wird sicherstellt, dass überhaupt das richtige System entwickelt wird. Verifikation wird in [28] als

> *„process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase“*

definiert. Oft wird die Verifikation auch als der Prozess umschrieben, der sicherstellt, dass ein System oder ein Teilsystem entsprechend einer gegebenen Spezifikation umgesetzt wird, die zu Beginn eines Teilprozesses im Entwicklungsprozess definiert wurde. Im Vergleich zur Validierung ist die Verifikation daher meist ein interner Prozess [25], der nicht unmittelbar in Verbindung zum Anwender steht. Die Spezifikation ergibt sich dabei indirekt aus den Anforderungen des Anwenders. Aus dieser Perspektive kann die Validierung als der Sonderfall einer Verifikation betrachtet werden, bei dem eine Spezifikation eines Systems hinsichtlich der Erfüllung von Anforderungen des Anwenders verifiziert wird.

2.1.2.1. Methoden zur Verifikation

Entsprechend [130] können drei grundlegende Ansätze zur Verifikation unterschieden werden, formale Verifikationsmethoden, simulative Verifikationsmethoden und prototypische Implementierungen.

- **Formale Verifikationsmethoden**: Formale Verifikationsmethoden basieren auf mathematischen Beweisen und setzen, wie die Synthese, unter Verwendung von Berechnungsmodellen (vgl. Abschnitt 2.2.3) eindeutig definierte Verhaltensspezifikationen voraus. Formale Methoden sind vollständig bzgl. der Abdeckung des Zustandsraumes einer zu verifizierenden Spezifikation. Dazu muss die Spezifikation selbst ebenfalls vollständig sein, was die Anwendbarkeit formaler Methoden auf kleine (vollständig spezifizierte) Teilsysteme limitiert [130]. Komplexe Systeme, die u.U. nicht-deterministischen Störungen unterliegen und deswegen nicht vollständig formal beschrieben werden können, lassen sich beispielsweise nur schwer oder gar nicht formal verifizieren.
- **Simulative Verifikationsmethoden**: Diese basieren auf ausführbaren Simulationsmodellen, deren Verhalten auch durch Berechnungsmodelle (vgl. Abschnitt 2.2.2) restriktiert ist. Während der Ausführung wird ein Simulationsmodell dann mit einer Menge von zuvor innerhalb eines Testfalls spezifizierten Mustern (Testpatterns) angeregt. Die Verifikation erfolgt durch Vergleich der Ausgabe des Simulationsmodells mit der erwarteten Ausgabe. Dieser Ansatz resultiert in der Regel in einer unvollständigen Abdeckung des Zustandsraumes, da es aufgrund der Systemkomplexität normalerweise nicht möglich ist, alle relevanten Testfälle zu simulieren. Ursachen sind entweder schlicht die Unmöglichkeit, alle relevanten Testfälle überhaupt identifizieren zu können oder ein zu hoher Zeitaufwand für die Simulation bei hohem Detailgrad. Anstelle eines vollständigen Korrektheitsbeweises ist deswegen nur eine Falsifikation [130] (d.h. ein selektiver Ausschluss von Fehlern) möglich. Da die Spezifikation nicht vollständig sein muss, können im Unterschied zur formalen Verifikation allerdings auch komplexe Systeme simuliert werden.
- **Prototypische Implementierungen**: Diese können den Geschwindigkeitsnachteil bei der Ausführung von detaillierten Simulationsmodellen ausgleichen. Allerdings ist der Aufwand zur Erstellung einer prototypischen Implementierung mit dem Aufwand für eine tatsächliche Realisierung vergleichbar, wodurch prototypische Implementierungen im Allgemeinen nur sehr spät im Entwicklungsprozess verwendet werden. Darüber hinaus sind die Möglichkeiten zum Debuggen und Testen beschränkt.

In Abb. 2.3 ist der Unterschied zwischen formalen und simulativen Verifikationsmethoden visuell dargestellt. Beide Methoden können als eine Abtastung eines

sog. Ausgaberaumes angesehen werden. Punkte im Ausgaberaum repräsentieren alle möglichen Systemausgaben, die als Antwort auf Systemeingaben aus einem Eingaberaum erzeugt werden können.

Bei simulativer Verifikation entsprechen einzelne Punkte im Eingaberaum Testpatterns, die durch Testfälle erzeugt werden. Ein einzelner Punkt im Ausgaberaum wird erst dann abgetastet, wenn das zugehörige Testpattern durch einen passenden Testfall generiert wird. Existiert kein passender Testfall, so findet auch keine Abtastung statt.

Bei einer Beweisführung im Rahmen einer formalen Verifikation wird hingegen eine Menge von Eingaben auf einmal berücksichtigt. Dadurch wird in der Regel eine zusammenhängende Menge von Punkten im Ausgaberaum auf einmal abgetastet.

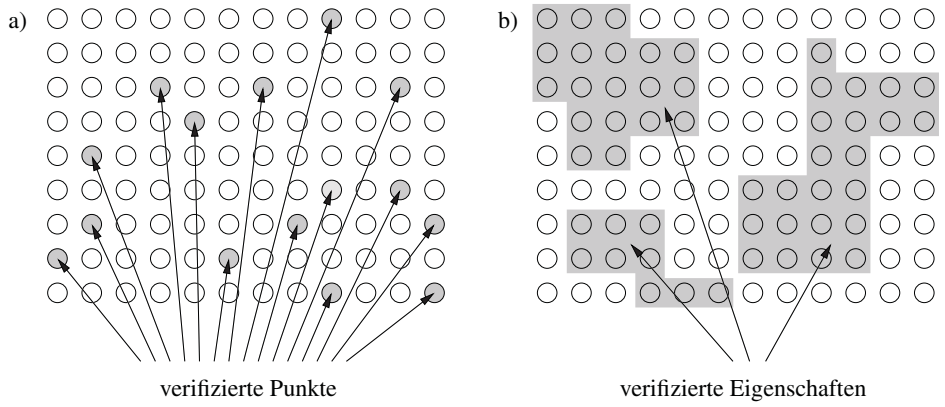


Abbildung 2.3.: Abtastung des Ausgaberaums bei a) simulativer und b) formaler Verifikation (Quelle: [130])

2.2. Modellbildung und Simulation

2.2.1. Modell und Modellbildung

Die Grundlage der im vorigen Abschnitt beschriebenen Entwurfs- und Verifikationsmethoden ist die Existenz von Modellen. Nach Stachoviak [240] besitzt ein Modell drei Hauptmerkmale, das Abbildungsmerkmal, das Verkürzungsmerkmal und das pragmatische Merkmal:

Ein Modell ist grundsätzlich eine natürliche oder künstliche Abbildung oder Repräsentation eines Originals. Dieses Original kann real existieren oder ein fiktives bzw. hypothetisches Original sein. Die Repräsentation des Originals ist „verkürzt“, d.h. sie erfasst nicht alle Attribute des Originals, sondern nur die, die dem Ersteller des Modells als relevant erscheinen. Ein Modell ist pragmatisch, da es aufgrund seiner Verkürzungseigenschaft und der damit verbundenen Abstraktion einem bestimmten Original nicht mehr eindeutig zugeordnet werden kann. Zusammengefasst ist ein Modell damit eine vereinfachte Darstellung eines Originals, die bestimmte, als relevant betrachtete Aspekte, möglichst akkurat wiedergeben soll.

Unter dem Begriff der Modellbildung oder Modellierung versteht man den Prozess der Erstellung eines Modells [191]. Das Ziel ist dabei, eine Repräsentation zu entwickeln, die aus Gründen der Komplexitätsreduktion so weit wie möglich vom Original abstrahiert und nur die für eine bestimmte Analyse wesentlichen Attribute beinhaltet.

Übertragen auf den hier betrachteten Bereich der eingebetteten Systeme bedeutet Modellbildung die (verkürzte) Spezifikation aller der zur Entwicklung eines eingebetteten Systems notwendigen Aspekte. Ein Modell kann beispielsweise mehrere oder nur einen der in Abb. 2.1 dargestellten Entwurfsaspekte abdecken. Die Vernachlässigung oder zusätzliche Einbeziehung eines Entwurfsaspekts hat eine weiteren Verlust oder Gewinn an Genauigkeit zur Folge. Je weiter der Entwurfsprozess fortgeschritten ist, desto detaillierter und näher an der finalen Implementierung ist im Allgemeinen das Modell. Die Aufgabe des Entwicklers ist es, auf Basis existierender Anforderungen zu entscheiden, welcher Grad an Genauigkeit für eine Spezifikation während einer bestimmten Phase des Entwurfs notwendig ist und welche Details bei der Modellbildung vernachlässigt werden können.

2.2.2. Simulation

In der VDI Richtlinie 3633 wird der Begriff der Simulation definiert als das

> „*Nachbilden eines Systems mit seinen dynamischen Prozessen in einem experimentierfähigen Modell, um zu Erkenntnissen zu gelangen, die auf die Wirklichkeit übertragbar sind*“.

Die Voraussetzung für eine Simulation ist also die Eigenschaft der Experimentierfähigkeit des Modells und die Möglichkeit, das Verhalten eines (originalen) physikalischen Systems zu imitieren. In einer Computersimulation geschieht diese Nachbildung des Verhaltens durch ein *Computer-aided Engineering (CAE)* Werkzeug genannt Simulator.

Ein Simulator basiert auf einer Modellierungssprache, welche ein syntaktisches und semantisches Regelwerk zur Spezifikation und Ausführung von Simulationsmodellen zur Verfügung stellt. Neben Syntax (wie wird ein Modell repräsentiert, mit welchen Symbolen) und Semantik (was bedeutet ein Modell) ist eine Modellierungssprache im Allgemeinen auch durch Pragmatik (was bedeutet ein Modell in einem konkreten Kontext und wie wird es verwendet) definiert [217].

Neben der Festlegung der statischen Bedeutung von syntaktischen Elementen (*statische Semantik*) beinhaltet das semantische Regelwerk einer ausführbaren Modellierungssprache wie der eines Simulators insbesondere eine Beschreibung des erlaubten Verhaltens (*dynamische Semantik*) [128, 129]. Dieser Teil des semantischen Regelwerks wird auch als *Berechnungsmodell* (siehe Abschnitt 2.2.3) bezeichnet und ist für alle Simulationsmodelle gültig, welche in der Modellierungssprache beschrieben werden können. Abb. 2.4 zeigt die für einen Simulator typische Grundstruktur.

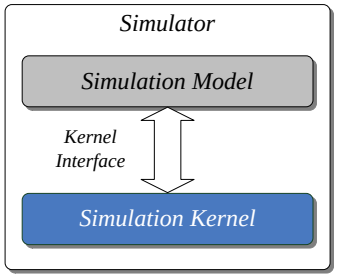


Abbildung 2.4.: Grundstruktur eines Simulators

Ein Simulationsmodell wird üblicherweise durch Instanziierung syntaktischer Basiskonstrukte (z.B. Prozesse, Kanäle oder Module) erzeugt. Diese sind in Form einer Bibliothek hinterlegt. Sie bilden den Ausgangspunkt für eine Spezialisierung im Sinne einer verhaltens- und/oder strukturorientierten Modellierung. Die resultierende Spezifikation repräsentiert zunächst eine statische Abbildung eines physikalischen Systems. Erst die Ausführung des Modells durch einen Simulationskernel erzeugt eine dynamische Abbildung. Der Simulationskernel implementiert dazu ein bestimmtes Berechnungsmodell. Kernel und Modell interagieren über eine Schnittstelle miteinander. Über diese Schnittstelle steuert der Kernel die Ausführung des Simulationsmodells entsprechend dem implementierten Berechnungsmodell.

Der Vorteil der Trennung in Simulationsmodell und -kernel liegt in der Flexibilität: Während das Modell eine spezifische Beschreibung des zu analysierenden Systems enthält, implementiert der Kernel sämtliche modellübergreifenden Aspekte. Grundsätzlich ist die Implementierung des Kernels unabhängig von

dem konkreten System, das durch das Simulationsmodell repräsentiert wird. Allerdings schränkt das Berechnungsmodell des Kernels dessen Anwendbarkeit auf eine bestimmte Klasse von Simulationsmodellen ein.

2.2.3. Berechnungsmodelle

Der Begriff des Berechnungsmodells (engl. *Model of Computation (MoC)*) hat seine Wurzeln in der theoretischen Informatik und hier speziell in der Erforschung der Berechenbarkeit von mathematischen Funktionen [106]. In den 1930er Jahren wurde die Berechenbarkeit von Funktionen erstmals anhand von abstrakten Berechnungsmodellen untersucht. Beispiele für solche Berechnungsmodelle sind der von Alonzo Church eingeführte Lambda-Kalkül [87] oder die von Alan Turing entwickelte Turingmaschine [256]. Letztere ist ein abstraktes Rechnermodell, das die Arbeitsweise eines Computers anhand der schrittweisen Ausführung eines Automaten beschreibt.

In Anlehnung an den Begriff des Entwurfsmusters, wie er von Gamma et al. in [117] im Kontext objektorientierter Programmiersprachen verwendet wird, bilden Berechnungsmodelle nach Lee et al. [178] Entwurfsmuster für Interaktionen zwischen Komponenten eines Systems. Die Regeln eines Berechnungsmodells schränken das erlaubte Verhalten eines Modells auf eine gültige Teilmenge aller möglichen Verhaltensweisen ein. Neben der Simulation werden Berechnungsmodelle auch zur formalen Spezifikation der Interaktion von Komponenten eines zu entwickelnden Systems genutzt und sind elementarer Bestandteil automatisierter Syntheseverfahren (vgl. Abschnitte 2.1.1.1 und 2.1.2.1).

2.2.3.1. Klassifikation von Berechnungsmodellen

Verschiedene Berechnungsmodelle unterscheiden sich dahingehend, wie gut oder wie schlecht sich bestimmte Charakteristika eines Systems erfassen und analysieren lassen. Dies beinhaltet Eigenschaften wie Parallelität, Ausführungsreihenfolge von Operationen, Scheduling, Synchronisation, etc. In [217] werden Regeln, welche ein bestimmtes Berechnungsmodell spezifizieren, in drei Kategorien eingeteilt:

1. Regeln, die spezifizieren was eine Komponente ist,
2. Regeln, die den Kommunikationsmechanismus zwischen Komponenten spezifizieren und
3. Regeln, die den Ausführungsmechanismus von Komponenten spezifizieren.

Ein Simulationskernel als die Implementierung eines Berechnungsmodells kontrolliert somit die Art und Weise, wie das Verhalten von Komponenten eines Simulationsmodells und Interaktionen zwischen diesen Komponenten auf einem Computer berechnet werden. Es legt die Gemeinsamkeiten im Verhalten all der Simulationsmodelle fest, die mit dem Kernel ausgeführt werden können.

In Abb. 2.5 ist eine mögliche (informelle) Klassifikation von typischen Berechnungsmodellen wie *State Machines (SM)*, *Discrete Event (DE)* oder *Continuous Time (CT)* dargestellt, die sich aus den genannten Regeln ergibt. Je weiter unten sich ein Berechnungsmodell in der Hierarchie befindet, desto spezieller und umfangreicher sind die Regeln, die es charakterisieren.

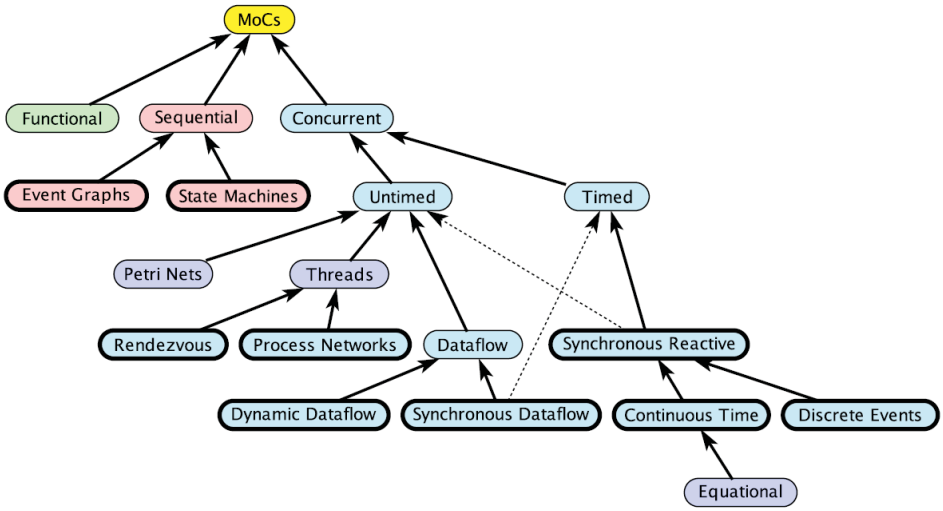


Abbildung 2.5.: Klassifikation von Berechnungsmodellen (Quelle: [217])

In Abb. 2.5 werden Berechnungsmodelle z.B. dahingehend unterschieden, ob die zugrunde liegenden Ausführungs- und Kommunikationsmechanismen die Spezifikation von *gleichzeitiger* oder nur von *sequentieller* Ausführung erlauben. Gleichzeitige Berechnungsmodelle werden weiter in *zeitbasiert* und *nicht zeitbasiert* unterteilt. Die Möglichkeiten zur Spezifikation von Zeit und Gleichzeitigkeit entscheiden darüber, inwieweit bestimmte Reihenfolgen von Aktionen spezifiziert werden können, die während einer Ausführung entstehen dürfen.

Generell lassen sich aus einer solchen Klassifikation Regeln ableiten, wie Berechnungsmodelle miteinander kombiniert werden können [122]. Ein formaler Ansatz zur Klassifikation findet sich z.B. in [175]. Die Klassifikation basiert auf einem denotationellen mathematischen Formalismus namens *Tagged Signal Model*, bei dem das Berechnungsmodell induktiv mit Hilfe der Mengentheorie be-

schrieben wird. Eine andere Möglichkeit zur Klassifikation ist eine operationelle Beschreibungsform wie in [255]. Diese ist näher an der Implementierung und erfasst schrittweise Zustandsänderungen.

2.2.3.2. Der Zeitbegriff in einer Simulation

Im Allgemeinen existieren im Kontext der Simulation unterschiedliche Begriffe und Definitionen von Zeit. Fujimoto unterscheidet in [113] zwischen drei grundlegenden Typen:

- **Physikalische Zeit**: Bezeichnet die Zeit des modellierten physikalischen Systems.
- **Simulationszeit**: Die Simulationszeit ist eine Abstraktion, die vom Simulator bzw. dem Simulationsmodell zur Repräsentation der physikalischen Zeit verwendet wird.
- **Reale Zeit**: Fujimoto nennt diese Zeit *Wallclock Time*. Sie bezeichnet die Zeit, die während der Ausführung eines Simulators vergeht.

Für die Simulationszeit und deren Verhältnis zur physikalischen Zeit gibt Fujimoto folgende allgemeine Definition, die eine lineare Beziehung zwischen Intervallen in der Simulationszeit und der physikalischen Zeit herstellt:

Definition 2.1 (***Simulationszeit***): *Die Simulationszeit ist definiert als eine vollständig geordnete Menge von Werten, wobei jeder Wert einen Zeitpunkt der physikalischen Zeit repräsentiert, die modelliert wird. Des Weiteren gilt für beliebige Simulationszeiten T_1 und T_2, die physikalische Zeiten P_1 und P_2 repräsentieren: Wenn $T_1 < T_2$, dann tritt P_1 vor P_2 auf und $(T_2 - T_1)$ ist identisch zu $(P_2 - P_1) \times K$ mit einer bestimmten Konstanten K. Falls $T_1 < T_2$, dann sagt man, dass T_1* ***vor*** *T_2 auftritt, und wenn $T_1 > T_2$, dann sagt man, dass T_1* ***nach*** *T_2 auftritt.*

In der Literatur existieren weitere speziellere Definitionen für die Simulationszeit und die Modellierung der physikalischen Zeit. Während obige Definition sehr allgemein gehalten ist, legen andere Zeitmodelle, wie das Modell der *Superdense Time* [76][190], den Schwerpunkt auf die Möglichkeit zur Spezifikation einer eindeutigen Definition von Gleichzeitigkeit. Die Superdense Time wird beispielsweise im Ptolemy II Simulator [217, 179] angewendet und ist die Grundlage für eine deterministische Ausführung. Ein ähnlicher Ansatz zur Modellierung von Gleichzeitigkeit existiert auch im SystemC Simulator [27].

Neben der Modellierung der physikalischen Zeit ist die Art und Weise der Repräsentation der Simulationszeit innerhalb eines Berechnungsmodells ebenfalls

ein kritischer Punkt. Die Simulationszeit kann beispielsweise als eine verteilte oder eine globale Variable repräsentiert sein (vgl. Abschnitte 2.2.3.3 und 2.2.3.4). Im Folgenden werden die Grundlagen der für diese Arbeit relevanten Berechnungsmodelle, insbesondere auch im Hinblick auf die Repräsentation der Simulationszeit, kurz erläutert.

2.2.3.3. Discrete Event (DE)

Diskrete ereignisbasierte Berechnungsmodelle dienen zur Simulation zeitdiskreter Systeme. Varianten von DE Berechnungsmodellen sind weit verbreitet und in vielen Werkzeugen zur Modellierung und Simulation von technischen Systemen implementiert. Dies ist der Tatsache geschuldet, dass viele technische Systeme von Natur aus diskret sind oder zumindest diskrete Anteile besitzen.

In einer *Discrete Event Simulation (DES)* wird das Verhalten eines Systems über der Zeit durch Sequenzen von diskreten Ereignissen (Events) modelliert. Ein Ereignis entspricht einem bestimmten Simulationszeitpunkt von unendlich kurzer Dauer, an dem sich der Zustand des modellierten Systems (möglicherweise) ändert. Grundlegende Konzepte eines DE Simulators, bestehend aus Simulationsmodell und Simulationskernel sind (vgl. [173][113]):

- **Zustandsvariablen**: Variablen, die den Zustand des modellierten physikalischen Systems speichern.
- **Globale Zeitvariable**: Eine globale Variable, die die aktuelle Simulationszeit speichert.
- **Globale Ereignisliste**: Eine globale Liste, die die in Zukunft auftretenden Ereignisse inkl. deren Zeitpunkt des Auftretens (Zeitstempel) speichert.
- **Zeitroutine**: Ein Unterprogramm, das das nächste zu verarbeitende Ereignis in der Ereignisliste bestimmt und anschließend die globale Zeitvariable auf den Zeitpunkt dieses Ereignisses setzt.
- **Ereignisroutinen**: Dies sind Unterprogramme in Form von Verhaltensbeschreibungen, die den Systemzustand aktualisieren, wenn ein bestimmtes Ereignis auftritt. Sie erfüllen den Zweck der in Abschnitt 2.2.3.1 im Rahmen der Klassifikation von Berechnungsmodellen erwähnten Komponenten. Die meisten DE Simulatoren implementieren heutzutage Ereignisroutinen, die auf sog. *Prozessen* basieren. Ein Prozess ist eine komplexe Ereignisroutine, die nicht nur das Verhalten in Reaktion auf ein einziges Ereignis, sondern auf eine Menge von Ereignissen implementieren kann.
- **Initialisierungs-/Terminierungsroutinen**: Unterprogramme, die zur Initialisierung und Terminierung der Gesamtsimulation dienen.

- **Hauptroutine**: Ein Unterprogramm, das auf Basis der Zeitroutine das Scheduling von Ereignissen durchführt und durch Aufruf von Ereignisroutinen den Zustandswechsel des Modells initiiert. Die Hauptroutine prüft auch, ob die Simulation beendet werden kann. Dies ist der Fall, wenn die maximale Simulationszeit erreicht ist oder die Ereignisliste keine Ereignisse mehr enthält.

Die Elemente und deren Beziehungen untereinander sind in Abb 2.6 illustriert. Entsprechend der Beschreibung aus Abschnitt 2.2.2 sind die Komponenten aufgeteilt in Modell und Kernel. Die Schnittstelle zwischen beiden ist durch die Beziehungspfeile dargestellt, die den grauen und den blauen Bereich verbinden. Der Ablauf der gesamten Simulationsausführung wird von der Hauptroutine gesteuert. Eine mögliche Variante des DE Berechnungsmodells ist anhand einer entsprechenden Implementierung von Hauptroutine und Zeitroutine in Algorithmus 2.1 dargestellt.

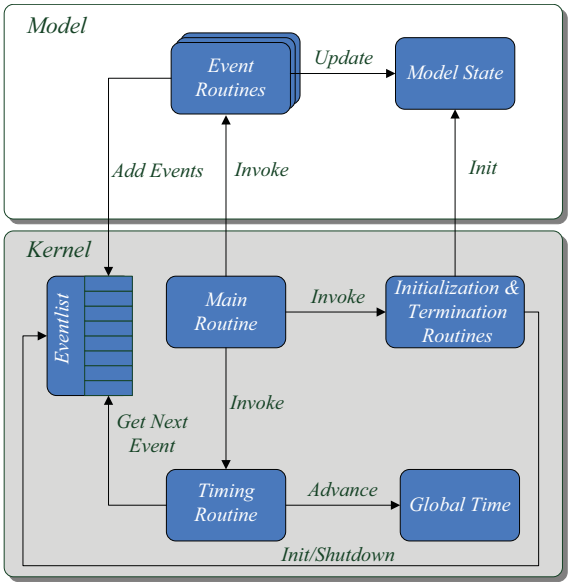


Abbildung 2.6.: Komponenten einer DE Simulation und deren Beziehungen (Quelle: [173], modifiziert)

Nach der Initialisierung durchläuft die Hauptroutine eine Schleife. Mit jeder Iteration der Schleife wird mit Hilfe der Zeitroutine das nächste auszuführende Ereignis bestimmt und durch Aufruf des zugehörigen Prozesses ausgeführt. Das nächste auszuführende Ereignis entspricht dem Ereignis in der Ereignisliste, das den kleinsten Zeitstempel besitzt. Nur, wenn alle Ereignisse mit dem gleichen

Algorithm 2.1

```
1: function MAINPROGRAM()
2:     INITIALIZATIONROUTINE()
3:     while (SimulationClock < MaxTime) do
4:         nextEvent = TIMINGROUTINE()
5:         EventRoutines[nextEvent].EXECUTE()
6:     end while
7:     TERMINATIONROUTINE()
8: end function
9:
10: function TIMINGROUTINE()
11:     nextEvent = EventList.GETNEXTEVENT()
12:     SimulationClock = SimulationClock + nextEvent.GETTIME()
13:     return nextEvent
14: end function
```

Zeitstempel abgearbeitet sind, schreitet die Simulation in der Simulationszeit voran. Die Ausführung terminiert, sobald die maximale Simulationszeit erreicht ist. Abb. 2.7 illustriert hierzu ein Beispiel.

Der Zustand des Systems wird anhand von vier Zustandsvariabeln $S0 - S3$ modelliert. Ein senkrechter Strich entspricht einem Zeitfortschritt, ein blauer Punkt einer Wert- bzw. Zustandsänderung aufgrund eines Ereignisses und dem Aufruf eines Prozesses. Die Anzahl der in einem bestimmten Zeitintervall auftretenden Ereignisse ist direkt vom Modell abhängig. Iterationen durch die Schleife in der Hauptroutine erfolgen nur an markanten Punkten innerhalb der Simulationszeit. Diese sind direkt durch die Ereignisse definiert. Falls über ein längeres Zeitintervall kein Ereignis auftritt, so wird dieses Zeitintervall einfach übersprungen. Dadurch wird unnötiger Overhead vermieden. Umgekehrt macht die Hauptroutine Zum Zeitpunkt „7.5“ aufgrund der drei Ereignisse drei Iterationen, bevor sie zum Zeitpunkt „9“ voranschreitet.

2.2.3.4. Parallel Discrete Event (PDE)

Parallele diskrete ereignisbasierte (engl. *Parallel Discrete Event (PDE)*) Berechnungsmodelle sind eine umfangreiche Klasse von speziellen DE Berechnungsmodellen, deren Hauptmotivation die Beschleunigung einer DE Simulation durch parallele Ausführung ist.

Eine *Parallel Discrete Event Simulation (PDES)* besteht typischerweise aus sog. *logischen Prozessen* (engl. *logical Processes*). Ein logischer Prozess wird, äquivalent zum einem gewöhnlichen Prozess in einer DE Simulation, zur Verhaltensmo-

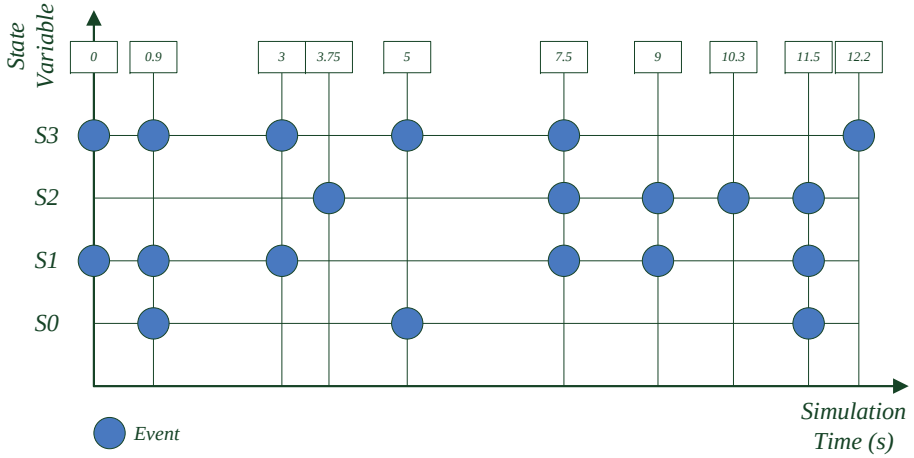


Abbildung 2.7.: Diskrete ereignisbasierte Simulation

dellierung eingesetzt. Entsprechend der Definition von Chandy und Misra [78] bilden logische Prozesse untereinander ein Netzwerk und kommunizieren ausschließlich anhand von Nachrichten über Verbindungen, die oft auch als *logische Verbindungen* (engl. *logical Links*) bezeichnet werden. Logische Verbindungen repräsentieren eine gerichtete Punkt-zu-Punkt Kommunikation auf Basis von *FIFOs*. Über sie können die logischen Prozesse Nachrichten (typischerweise zeitgestempelte Ereignisse) austauschen und sich dabei miteinander zeitlich synchronisieren.

Im Unterschied zu Prozessen in einer DES simuliert ein logischer Prozess häufig ein ganzes Subsystem des modellierten physikalischen Gesamtsystems [108]. Ein logischer Prozess fasst dann mehrere gewöhnliche DE Prozesse zu einem komplexen Prozess zusammen. Er kann dann in erster Instanz als ein erweiterter sequentieller DE Simulator betrachtet werden, der nur auf einen Teil der insgesamt vorhandenen Zustandsvariablen Zugriff hat. In Abb. 2.8 ist beispielhaft ein Prozessnetzwerks bestehend aus vier logischen Prozessen $LP1$ bis $LP4$ illustriert. Für jeden logischen Link ist im Empfänger ein zugehöriger Eingangs-FIFO eingezeichnet. Zudem besitzt jeder logische Prozess eine lokale Ereignisliste, die Teil einer lokalen DES ist.

Damit ein PDE Berechnungsmodell, dessen logische Prozesse ausschließlich anhand von Nachrichten über logische Links kommunizieren, in der Lage ist, korrekte Simulationsergebnisse zu liefern, ist die Einhaltung der sog. *lokalen Kausalitätsbedingung* notwendig. Fujimoto formuliert diese Bedingung in [112] folgendermaßen:

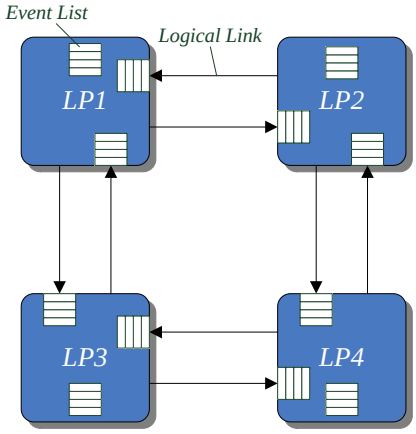


Abbildung 2.8.: Beispiel eines Prozessnetzwerks bestehend aus vier logischen Prozessen

Definition 2.2 *(**Lokale Kausalitätsbedingung**): Eine diskrete ereignisbasierte Simulation, die aus logischen Prozessen besteht, welche ausschließlich durch den Austausch von Nachrichten interagieren, hält die lokale Kausalitätsbedingung dann und nur dann ein, wenn jeder logische Prozess Ereignisse in der Reihenfolge ihrer Zeitstempel verarbeitet.*

Anders ausgedrückt, kann eine PDES im Sinne von Definition 2.2 dann als kausal korrekt bezeichnet werden, wenn deren Ausführung das gleiche Ergebnis liefert, wie eine entsprechende sequentielle DES. Die Entwicklung und Untersuchung von Mechanismen zur Synchronisation der Simulationszeit und zur Einhaltung der lokalen Kausalitätsbedingung bildet einen zentralen Aspekt in der Forschung im Bereich PDES. Dabei definiert das Netzwerk aus logischen Prozessen und logischen Links die Grundstruktur eines parallelen Simulators und bildet einen Rahmen für die Implementierung verschiedener PDE Berechnungsmodelle.

Viele dieser PDE Varianten sind nicht auf eine nachrichtenbasierte Kommunikation und Synchronisation limitiert. Der nachrichtenbasierte Ansatz kann jedoch als ein erstes Hilfsmittel für die Vermeidung von kausalen Fehlern betrachtet werden, die durch eine falsche Zugriffsreihenfolge von mehreren logischen Prozessen auf gleiche Zustandsvariablen entstehen können (vgl. [112]). Die Schwierigkeit bei der Entwicklung eines Synchronisationsverfahrens besteht dann meist darin, dass die Forderung nach Einhaltung der lokalen Kausalitätsbedingung

der Forderung nach Performanz, Effizienz oder Flexibilität oftmals auf den ersten Blick konträr gegenübersteht.

Existierende Synchronisationsalgorithmen zur PDES lassen sich auf unterschiedliche Art und Weise klassifizieren (ein umfangreicher Überblick ist in [149] zu finden):

- Die am weitesten verbreitete Klassifikation ist die Unterscheidung hinsichtlich *konservativer* und *optimistischer* Algorithmen zur Synchronisation. Konservative Algorithmen vermeiden die Verletzung der lokalen Kausalitätsbedingung. Sie garantieren, dass innerhalb eines logischen Prozesses Events immer in der kausal korrekten Reihenfolge verarbeitet werden. Fundamental in konservativen Algorithmen für die Einhaltung der lokalen Kausalitätsbedingung und die Vermeidung von Deadlocks ist eine Größe namens *Lookahead*, welche ein Zeitintervall beschreibt, um die ein logischer Prozess in die Zukunft schauen kann. Beispiele für konservative Algorithmen sind der sog. *Null Message Algorithmus (NMA)* oder *Chandy-Misra-Bryant (CMB)* Algorithmus [78, 69] oder der Bounded Lag Algorithmus von Lubachevsky [187]. [241] oder [207] sind weitere konservative Beispiele. Im Gegensatz zu konservativen Ansätzen erlauben optimistische Algorithmen, die Kausalitätsbedingung zu verletzen. Sie stellen allerdings Mechanismen zur Erkennung von Kausalitätsverletzungen und zur Wiederherstellung früherer valider Systemzustände zur Verfügung. In diese Kategorie gehören beispielsweise der *Time-Warp* Algorithmus von Jefferson et. al [150] oder der *Global Virtual Time* Algorithmus von Mattern [196].
- Eine andere Möglichkeit ist die Einteilung der Synchronisationsalgorithmen in *synchrone* und *asynchrone* Ansätze [44]. Synchrone Algorithmen erzeugen eine regelmäßige globale Synchronisation zwischen logischen Prozessen. Dies resultiert darin, dass die gesamte Simulation an bestimmten Punkten in der Realzeit regelmäßig pausiert und wieder losläuft (siehe Abb. 2.9 oben). Der Abstand dieser globalen Synchronisationspunkte wird häufig durch eine sog. *global Reduction* bestimmt. dazu ist eine globale Sicht über den aktuellen Zustand aller beteiligten logischen Prozesse notwendig. Die Synchronisationspunkte selbst werden typischerweise durch Barriers implementiert.

  Im Unterschied zu synchronen Algorithmen erlauben asynchrone Algorithmen den logischen Prozessen in der Simulationszeit voranzuschreiten, ohne zwangsläufig global zu synchronisieren. Stattdessen wird für einzelne logische Prozesse der Zeitfortschritt separat berechnet (siehe Abb. 2.9 unten). In die Berechnung des Zeitfortschritts eines logischen Prozess p muss ausschließlich der Zustand der logischen Prozesse einbezogen werden, die potentiell das Verhalten von p beeinflussen können. Globale Wartezustände entsprechen daher Deadlocks, die es soweit wie möglich zu ver-

meiden gilt. Falls das Auftreten von Deadlocks nicht verhindert werden kann, beispielsweise im Fall eines Lookahead von Null (*Zero Lookahead*), können diese mit Hilfe von sog. *Deadlock Detection* und *Deadlock Recovery* Mechanismen aufgelöst werden [113]. In [78],[69],[150], [196] oder [43] finden sich Beispiele für asynchrone Ansätze. [241], [187] und [207] sind Beispiele für synchrone Algorithmen.

- Eine dritte oft verwendete Einteilung fokussiert stärker auf die logische Struktur, die sich aus dem Synchronisationsalgorithmus ergibt. Dabei wird zwischen *zentralen* und *dezentralen* Ansätzen unterschieden [44]. Eine globale Reduktion in Kombination mit globalen Barriers wird im einfachsten Fall mit Hilfe eines zentralen Controllers umgesetzt. Dies entspricht einer sog. *Centralized Barrier* [113]. Da der Controller schnell zu einem Hot-Spot in der Kommunikation werden kann, wurden in der Literatur verschiedene dezentrale Alternativen vorgeschlagen, wie z.B. *Tree-* oder *Butterfly-Barriers* [267][68].

  Zentralisierte Ansätze sind dann von Vorteil, wenn für einen Algorithmus eine globale Sichtweise notwendig ist. Viele synchrone Algorithmen benötigen eine solche globale Sichtweise, um einen möglichst großen Abstand der globalen Barriers berechnen zu können. Es existieren auch Beispiele von asynchronen Algorithmen, die auf zentralisierte Strukturen zurückgreifen, beispielsweise wird in [79] ein zentraler Controller zur Erkennung und Auflösung von Deadlocks verwendet.

Prinzip des Null Message Algorithmus

Als Beispiel wird im Folgenden das dem Null Message Algorithmus zugrundeliegende Prinzip erläutert. Ein besonderes Augenmerk gilt dabei der Repräsentation und Synchronisation der Simulationszeit. Während in einer gewöhnlichen DES die Zeit durch eine allen Simulationsprozessen bekannte globale Zeitvariable repräsentiert wird, existiert bei einer PDES auf Basis des NMA in jedem logischen Prozess lp_i eine separate lokale Zeitvariable t_i^{local}. Ein logischer Link l_{ij} von lp_i nach lp_j existiert nur dann, wenn das Teilmodell in lp_i das Teilmodell in lp_j über Ereignisse kausal beeinflussen kann. Jedes Ereignis e, das in Form einer Nachricht über eine logische Verbindung übertragen wird, trägt einen Zeitstempel $ts(e)$. Dieser spezifiziert den (zukünftigen) Simulationszeitpunkt, an dem das Ereignis verarbeitet werden soll.

Unter der Voraussetzung, dass Ereignisse zwischen logischen Prozessen immer in zeitlich aufsteigender Reihenfolge übertragen werden und folglich auch in zeitlich aufsteigender Reihenfolge in den FIFOs der logischen Links gespeichert sind, hält ein logischer Prozess lp_i die kausal richtige Verarbeitungsreihenfolge von Ereignissen ein, wenn er im Rahmen des Schedulings immer genau das Ereignis als das nächste zu verarbeitende Ereignis e^{next} auswählt, das von allen Er-

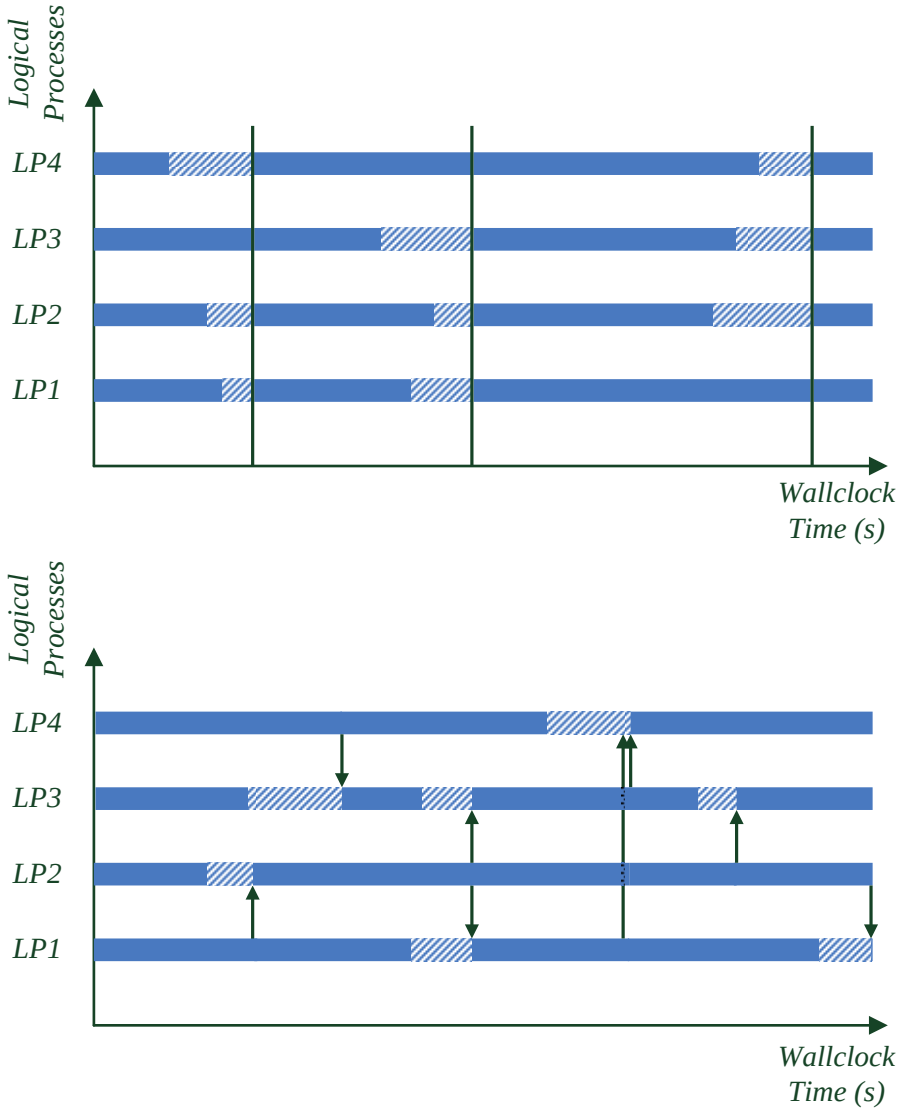


Abbildung 2.9.: Gegenüberstellung von synchronen (oben) und asynchronen (unten) Algorithmen

eignissen E_i^{cur}, die aktuell in den FIFOs und der lokalen Ereignisliste vorhanden sind, den kleinsten Zeitstempel besitzt. Dies bedeutet:

$$e^{next} = e^{min}, \text{ wobei } ts(e^{min}) = \min_{\forall e \in E^{cur}}(ts(e)), \tag{2.1}$$

Nach der Bestimmung von e^{next} wird t_i^{local} auf den Zeitstempel $ts(e^{next})$ gesetzt und e^{next} wird durch Aufruf der zugehörigen Ereignisroutine ausgeführt. Diese Methode funktioniert solange, wie sich in jedem FIFO mindestens ein Ereignis befindet. Ist dies nicht der Fall, so kann der logische Prozess vorhandene Ereignisse nicht verarbeiten, da u.U. noch Ereignisse über momentan leere FIFOs eintreffen können, die früher verarbeitet werden müssten. Der logische Prozess muss dann warten, bis jeder FIFO ein Ereignis enthält. Problematisch wird es insbesondere dann, wenn eine zirkuläre Abhängigkeit zwischen mehreren Prozessen besteht. In diesem Fall kann ein Deadlock genau dann auftreten, wenn logische Prozesse gegenseitig auf die Übertragung eines Ereignisses warten.

Beim NMA werden solche Situation durch den regelmäßigen Versand sog. Null Messages und die Einführung einer Größe namens Lookahead verhindert. Der Lookahead bezeichnet die Latenz einer logischen Verbindung bzgl. der Simulationszeit. Er wird für jede logische Verbindung statisch festgelegt. Der Wert des Lookaheads einer logischen Verbindung muss aus dem Simulationsmodell abgeleitet werden. Der Zeitstempel ts einer jeden Nachricht (auch der von Null Messages) wird von einem Sender lp_i aus der Summe seiner aktuellen lokalen Zeit t_i^{local} und dem Lookahead Δl_{ij} des logischen Links l_{ij} gebildet, über den die Nachricht verschickt werden soll: $ts = t_i^{local} + \Delta l_{ij}$ (eine Alternative ist die Berechnung der Summe im Empfänger). Der Zeitstempel einer Null Message definiert dann eine untere Schranke für den Zeitstempel, den ein in Zukunft über l_{ij} übertragenes Ereignis haben kann. Algorithmus 2.2 illustriert das beschriebene Verfahren durch Erweiterung des Pseudocodes aus Algorithmus 2.1.

2.2.4. Heterogenität

Ein Simulationswerkzeug basiert idealerweise auf einem Berechnungsmodell, welches optimal an die Bedürfnisse des Modellerstellers oder Applikationsentwicklers angepasst ist. In bestimmten Fällen ist es zudem notwendig oder von Vorteil, unterschiedliche Spezifikationen, die unterschiedlichen Berechnungsmodellen folgen, miteinander in Beziehung zu setzen. In diesem Kontext kann man zwischen *vertikaler* und *horizontaler* Heterogenität unterscheiden [136, 184]:

Algorithm 2.2

```
1: function MAINPROGRAM()
2:     INITIALIZATIONROUTINE()
3:     while (SimulationClock < MaxTime) do
4:         WAITFIFOS()
5:         nextEvent = TIMINGROUTINE()
6:         EventRoutines[nextEvent].EXECUTE()
7:         SENDNM()
8:     end while
9:     TERMINATIONROUTINE()
10: end function
11:
12: function TIMINGROUTINE()
13:     nextEvent = GETMINTIMEEVENT()
14:     SimulationClock = SimulationClock + nextEvent.GETTIME()
15:     return nextEvent
16: end function
```

- **Vertikale Heterogenität**: Vertikale Heterogenität wird in [136] als die Möglichkeit beschrieben, Berechnungsmodelle entlang eines Entwicklungsprozesses zu transformieren. Sie ist z.B. innerhalb einer SLD Entwurfsmethodik von großer Bedeutung, bei der Modelle / Spezifikationen unterschiedlicher Abstraktionsebenen Schritt für Schritt durch Synthese und damit verbundener Transformation zugrundeliegender Spezifikationen ineinander überführt werden.
- **Horizontale Heterogenität**: Bezeichnet die Heterogenität zwischen miteinander integrierten Modellen. Beispielsweise sind Modelle von eingebetteten Systemen von Natur aus horizontal heterogen, da deren Teilsysteme oft unterschiedlichen Anwendungsdomänen entstammen und sehr unterschiedliche Charakteristika hinsichtlich des Verhaltens aufweisen. Neben Komponenten aus Hardware oder Software können Teile eines eingebetteten Systems z.B. auch aus mechanischen, hydraulischen oder analogen Komponenten bestehen [102]. Auch die Integration von Teilsystemmodellen ein und derselben Anwendungsdisziplin, die aber auf unterschiedlichen Abstraktionsebenen spezifiziert sind, führt zu horizontaler Heterogenität.

Horizontale Heterogenität kann die Folge vertikaler Heterogenität sein, beispielsweise wenn im Zuge einer vertikalen Verfeinerung Teile eines vormals homogenen Modells durch Teilmodelle niedrigerer Abstraktionsebenen ersetzt werden.

Umgekehrt kann vertikale Heterogenität auch die Folge horizontaler Heterogenität sein (z.B. im Fall von Ptolemy II, siehe Abschnitt 2.3.4).

Die dynamische Kombination von Simulationswerkzeugen, die unterschiedlichen Anwendungsdisziplinen entstammen und verschiedenen Berechnungsmodellen folgen, entspricht einer Co-Simulation. Eine Alternative zur Co-Simulation sind Simulationswerkzeuge basierend auf sog. *formalen Frameworks* (siehe Abschnitt 2.3.3). Mit diesen können heterogene Systeme in ein und derselben Sprache spezifiziert und simuliert werden.

2.3. Sprachen für den Systementwurf

Um die innerhalb von Entwurfsprozessen für elektronische Systeme (vgl. Abschnitt 2.1) typischerweise anfallenden Aufgaben wie Spezifikation, Exploration, Synthese oder Verifikation optimal zu unterstützen, haben sich sowohl in der Industrie als auch im Bereich der Forschung eine ganze Reihe unterschiedlichster Modellierungssprachen und Simulationswerkzeuge herausgebildet. Diese unterscheiden sich stark hinsichtlich der Abdeckung von Anwendungsdomänen und Anwendungsmöglichkeiten.

Im Folgenden werden zunächst Sprachen für den Entwurf von digitalen Hardware- und Softwaresystemen vorgestellt. Anschließend werden Werkzeuge beschrieben, welche es zusätzlich gestatten, Wechselwirkungen zwischen heterogenen Teilsystemen (z.B. Hardware/Software und physikalischer Umwelt) zu spezifizieren. Diese bilden die Grundlage zur Entwicklung heterogener eingebetteter Systeme.

2.3.1. Sprachen zur Modellierung von Hardware und Software

Zur Entwicklung von Software existieren Sprachen wie z.B. C/C++, Java oder C#. Im Bereich der eingebetteten Systeme ist C/C++ die wohl verbreitetste Sprache. Die genannten Sprachen sind allesamt sog. Hochsprachen, d.h. sie abstrahieren stark von der Hardware, auf der ein Programm letztendlich ausgeführt wird. Eine Spezifikation kann deswegen u.U. vollständig unabhängig von der zugrundeliegenden Hardware sein. Die genannten Sprachen eignen sich insbesondere deswegen nicht zur Spezifikation von Hardware, weil sie keine Möglichkeiten zur Beschreibung spezieller Charakteristika von Hardware wie signalbasierte Kommunikation, Zeitverhalten und Parallelität bieten.

Einschlägige Sprachen im Bereich der industriellen Hardwareentwicklung sind z.B. *VHDL* [19] oder Verilog [17]. Diese werden allgemein auch als *Hardware Description Languages (HDLs)* bezeichnet. VHDL und Verilog basieren beide auf ei-

nem DE Berechnungsmodell und bilden die Grundlage zur zyklengenauen Spezifikation und Simulation von diskreten Hardwareschaltungen. Wegen der Limitierung auf Hardware und der Beschränkung des Abstraktionsgrades auf das RTL, eignen sich diese Sprachen allerdings nicht für die Anwendung im Rahmen einer SLD Methodik, welche Hardware und Software einschließt.

Die bekanntesten sog. *System Level Design Languages (SLDLs)* sind SystemC [27] und SpecC [114] [119] [98]. SystemC ist eine Erweiterung von C/C++, SpecC basiert auf ANSI-C. SystemC und SpecC besitzen die notwendigen syntaktischen und semantischen Eigenschaften, um sowohl Hardware als auch Software beschreiben zu können. Wie VHDL und Verilog besitzen beide Sprachspezifikationen als Grundlage ein DE Berechnungsmodell. In beiden Sprachen existiert keine Beschränkung bzgl. *Register Transfer (RT)* Ebene. Deswegen sind diese Sprachen grundsätzlich für die Anwendung innerhalb einer SLD Methodik geeignet. Im Unterschied zu SpecC wird SystemC nicht nur im Bereich der Forschung, sondern auch in der Industrie verwendet. Aufgrund der Relevanz in den nachfolgenden Kapiteln, wird nun eine Einführung in die SystemC Syntax und die für SystemC Modelle gültige dynamische Semantik gegeben.

2.3.2. SystemC

SystemC [27] wurde Ende der 90er Jahre von der *Open SystemC Initiative (OSCI)* entwickelt. Die OSCI ging im Jahr 2011 in die Accelera Systems Initiative [1] über. SystemC ist seit dem Jahr 2005 ein IEEE Standard [27]. SystemC wird aktuell von vielen CAE Werkzeugen einschlägiger EDA Firmen unterstützt. Die Accelera Systems Initiativestellt stellt eine C/C++ Klassenbibliothek frei zur Verfügung, welche die Sprache implementiert. Dies ist der Grund, weshalb SystemC auch im Bereich der Forschung weit verbreitet ist. Abb. 2.10 gibt einen Überblick über die Komponenten der SystemC Bibliothek.

SystemC erlaubt die Spezifikation und Simulation von Hardware und Software. Da die Modellierung von Software größtenteils bereits durch C++ selbst abgedeckt wird, beinhaltet die Bibliothek hauptsächlich Artefakte zur Modellierung von Hardware [55]. Ähnlich zu HDLs wie VHDL oder Verilog kann mit SystemC struktur- und verhaltensorientiert modelliert werden. Im Folgenden werden zunächst die grundlegenden syntaktischen Komponenten allgemein erläutert. Anschließend wird das dem SystemC Kernel zugrundeliegende Berechnungsmodell beschrieben (eine detaillierte Einführung ist z.B. in [55] zu finden).

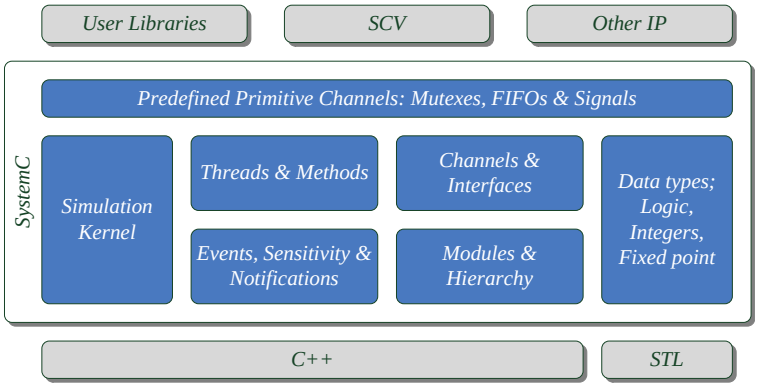


Abbildung 2.10.: Architektur der SystemC Bibliothek (Quelle: [55])

2.3.2.1. Syntaktische Komponenten

Das Grundelement der Syntax von SystemC zur Erzeugung von hierarchischen Strukturen ist das *Modul*. Ein Modul kann *Prozesse*, *Ereignisobjekte*, *Channels*, *Ports*, *Variablen*, *Funktionen* oder beliebige andere Module enthalten. All diese Komponenten sind in der SystemC Bibliothek als C++ Klassen implementiert.

Prozesse dienen zur Spezifikation von Verhalten. Das Scheduling von Prozessen wird im Zusammenspiel von Prozessen und Simulationskernel über Ereignisobjekte gesteuert (vgl. Abschnitt 2.3.2.2). Prozesse können durch Lesen und Schreiben von/auf Channels oder direkt über Variablen und beliebige Funktionsaufrufe kommunizieren. Kommunikation anhand von Channels erfolgt immer über definierte Schnittstellen. Kommunikation über Modulhierarchien hinweg ist über Ports möglich. Ports dienen als Stellvertreter für verbundene Channels. Die Aufteilung in Prozesse / Module und Channels vereinfacht die Separation von Belangen, wie sie bereits in Abschnitt 2.1.1.3 als Grundlage einer plattformbasierten SLD Methodik erwähnt wurde.

Grundsätzlich können zwei Typen von Channels unterschieden werden, *Primitive Channels* und *Hierarchical Channels*. Primitive Channels implementieren das sog. *Evaluate / Update (E/U)* Paradigma, das für die deterministische Simulation von Gleichzeitigkeit grundlegend ist (vgl. Abschnitt 2.3.2.2). Im Gegensatz zu Primitive Channels sind Hierarchical Channels Module und besitzen dementsprechend die gleichen Möglichkeiten zur Implementierung wie ein Modul. Ein Hierarchical Channel kann z.B. die Spezifikation eines komplexen Kommunikationsprotokolls beinhalten.

2.3.2.2. Berechnungsmodell

Das SystemC zugrundeliegende Berechnungsmodell ist eine Variante des DE Berechnungsmodells. Es wird im Folgenden anhand der vier Aspekte Zeitmodell, Prozesse, Kommunikationsmechanismen und Ausführungsmechanismus erläutert.

Zeitmodell

Das Zeitmodell von SystemC besitzt die Besonderheit, dass die Simulationszeit kein Skalar, sondern ein Tupel $t^{\tau,\delta} = (\tau, \delta)$ ist. Neben der eigentlichen Zeit τ existiert eine zweite unabhängige Variable δ, die die Anzahl der sog. *Deltacycles* seit dem Beginn der Simulation zählt. Deltacycles haben per Definition eine Zeitdauer bzgl. τ von Null.

Mit Hilfe von δ ist es möglich, eine partielle Ordnung für SystemC Ereignisse (sog. *Notifications*) zu generieren, die bei gleichem τ eintreten. Notifications können dann unterschiedlichen Deltacycles zugeordnet werden. Für zwei Zeitpunkte $t_1^{\tau,\delta}$ und $t_2^{\tau,\delta}$ lässt sich dabei folgende Ordnungsrelation definieren:

$$t_1^{\tau,\delta} > t_2^{\tau,\delta} \Leftrightarrow \tau_1 > \tau_2 \vee (\tau_1 = \tau_2 \wedge \delta_1 > \delta_2) \tag{2.2}$$

In Anlehnung an die Superdense Time aus dem Ptolemy II (vgl. Abschnitt 2.3.4.2) können Notifications, die bei gleichem τ auftreten, auch als *schwach gleichzeitig* bezeichnet werden. Falls sie auch beim gleichen δ auftreten, so kann man sie als *stark gleichzeitig* bezeichnen. Notifications können entsprechend [27] folgendermaßen klassifiziert werden:

- **Zeitverzögert** (*Timed Notifications*): Das Ereignis hat einen um $\Delta\tau$ verzögerten Eintrittszeitpunkt $\tau := \tau + \Delta\tau$ mit $\Delta\delta = 0$.
- **Deltaverzögert** (*Delta Notifications*): Das Ereignis hat einen um $\Delta\delta = 1$ verzögerten Eintrittszeitpunkt $\delta := \delta + 1$ mit $\Delta\tau = 0$.
- **Keine Verzögerung** (*Immediate Notifications*): Der Eintrittszeitpunkt des Ereignisses ist nicht verzögert, d.h. $\Delta\tau = 0$ und $\Delta\delta = 0$.

Spezifikation von Verhalten durch Prozesse

Die zentrale Komponente zur Verhaltensspezifikation ist der Prozess. Die Ausführung von Prozessen wird vom Simulationskernel anhand eines kooperativen Multitaskings und mit Hilfe von Notifications gesteuert. Notifications werden innerhalb von Prozessen erzeugt. Dies entspricht dem Aufruf der Methode *notify()* auf einem Ereignisobjekt ω aus der Menge aller Ereignisobjekte Ω.

Das Eintreten einer Notification kann wiederum zur Ausführung von Prozessen führen, sofern diese Prozesse *sensitiv* auf die Notification sind. Sensitivität für Notifications kann *dynamisch* mit Hilfe der Funktionen *wait()* oder *next_trigger()* oder *statisch* mit Hilfe des Schlüsselworts *sensitive* spezifiziert werden. Auch gezielte Timeouts können durch Aufruf von *wait()* oder *next_trigger()* auf dem Kernel erzeugt werden.

Wurde die Ausführung eines SystemC Prozesses einmal initiiert, so ist dieser aufgrund des kooperativen Multitaskings danach eigenständig dafür verantwortlich, den Kontext wieder an den Kernelscheduler zurückzugeben. Dabei können zwei Typen von Prozessen unterschieden werden: *SC_METHOD* Prozesse sind simple Callback-Funktionen. *SC_THREAD* Prozesse sind Co-Routinen und besitzen einen Stack. Ein *SC_METHOD* Prozess gibt die Kontrolle durch ein simples *return* an den Kernelscheduler zurück. *SC_THREAD* Prozesse implementieren normalerweise eine Endlosschleife. Sie geben die Kontrolle explizit durch Aufruf von *wait()* an den Kernel zurück.

Kommunikationsmechanismen

Die Semantik des Datenaustauschs zwischen SystemC Prozessen kann entsprechend der Kontrollierbarkeit durch den Kernel klassifiziert werden. Kontrolle durch den Kernel impliziert eine gewisse Restriktivität:

- **Restriktierte Kommunikation**: Diese Art der Kommunikation basiert auf der Verwendung von Primitive Channels wie z.B. Signalen (*sc_signal*) oder Fifos (*sc_fifo*). Signale werden typischerweise in RTL Modellen eingesetzt. Lese- oder Schreibzugriff von einem Prozess auf ein Signal muss mit Hilfe der *read()* oder *write()* Methoden erfolgen. Das E/U Paradigma von Primitive Channels beschränkt die Art und Weise der Weiterleitung von Daten zwischen SystemC Prozessen: Daten, die zu einem bestimmten Simulationszeitpunkt in einen Primitive Channel geschrieben werden, sind immer deltaverzögert am Ausgang sichtbar.

  Die Implementierung eines Primitive Channels *ch* basiert dazu auf den Methoden *request_update()*, *update()*, zwei Datenstrukturen v^{cur} und v^{next} sowie einem Ereignisobjekt ω. Ein Lesezugriff per *read()* resultiert in der Rückgabe des Wertes von v^{cur}, ein Schreibzugriff per *write()* in der Modifikation von v^{next}. Dabei registriert *request_update()* eine Aktualisierungsanfrage im Kernel. *update()* wird vom Kernel aufgerufen, um eine Aktualisierungsanfrage zu bearbeiten. Die Aktualisierung beinhaltet das Setzen von $v^{cur} = v^{next}$ und den Aufruf von *notify()* auf ω, wodurch eine Delta-Notification generiert wird.
- **Nicht-Restriktierte Kommunikation**: Bei dieser Art der Kommunikation wird auf das E/U Paradigma und die Verwendung entsprechender Primi-

tive Channels verzichtet. Der Kernel hat damit keine Kontrolle mehr über die Kommunikation. Die Kommunikation erfolgt vielmehr direkt über Variablen oder Funktionsaufrufe im Modell. Ein Prozess greift u.U. auf beliebige C++ Variablen unmittelbar lesend und schreibend zu. Simulationsergebnisse hängen dann von der Zugriffsreihenfolge auf die Variablen und damit vom Scheduling der SystemC Prozesse ab. Diese Art der Kommunikation wird typischerweise in sog. *Transaction Level Models (TLMs)* verwendet.

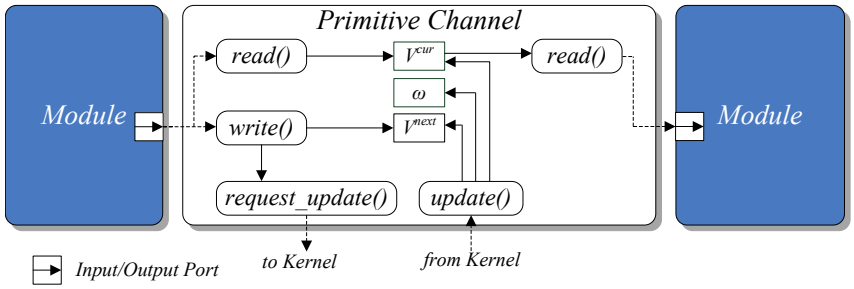


Abbildung 2.11.: Kommunikation über einen Primitive Channel

Ausführungsmechanismus

Der sequentielle SystemC Scheduler durchläuft während der Ausführung die *Elaboration Phase* und die *Simulation Phase*. Die Elaboration Phase dient zur Instanziierung des Modells und zur Herstellung von Modulverbindungen. Zur Erläuterung des Schedulingverfahrens ist sie nicht relevant. Es genügt daher die Betrachtung des iterativen Teils der Simulation Phase. Dieser besteht aus unterschiedlichen Teilphasen namens *Evaluation Phase*, *Update Phase*, *Delta Notification Phase* und *Timed Notification Phase*. Um die Beschreibung des iterativen Ablaufs der Teilphasen zu vereinfachen, werden zunächst folgende Variablen und Mengen definiert:

- *state*: Speichert den aktuellen Basiszustand des Schedulers, wobei *state* nur Werte aus der Menge $S = \{s^{eval}, s^{update}, s^{dnotify}, s^{tnotify}\}$ annehmen kann.
- τ: Speichert den Wert der Simulationszeit im aktuellen Timedcycle. Dabei ist $\tau \in \mathbb{R}^{\geq 0}$.
- δ: Zählt die Anzahl der in der gesamten Simulation bereits ausgeführten Deltacycles, dabei ist $\delta \in \mathbb{N}_0$.
- P: Menge aller SystemC Prozesse p.
- R: Menge der lauffähigen SystemC Prozesse r, dabei gilt: $R \subseteq P$.

- U: Menge der Aktualisierungsanfragen u.
- N^{δ}: Menge der Delta Notifications und Timeouts n^{δ} zum aktuellen Deltazyklus δ.
- N^{τ}: Menge der Timed Notifications und Timeouts n^{τ} zum aktuellen und zu zukünftigen Zeitpunkten τ.

Die Gesamtfunktionsweise des SystemC Schedulers lässt sich anhand folgender grundlegender Basisaktionen beschreiben:

- *get(SET)*: Wähle ein beliebiges Element aus der Menge SET.
- *del(SET1,SET2)*: Lösche alle Elemente in Menge $SET1$ aus Menge $SET2$.
- *run(p)*: Führe Prozess p aus.
- *eval(R)*: Führe folgende Sequenz aus, solange $|R| \neq 0$:
  - *p=get(R); run(p); del(p,R)*;
- *update(U)*: Führe folgende Sequenz aus, solange $|U| \neq 0$:
  - *u=get(U)*; Führe das Update u durch Aufruf von *update()* auf dem zugehörigen Primitive Channel *ch* aus; *del(u,U)*;
- *dnotify(N^{δ})*: Führe folgende Sequenz aus, solange $|N^{\delta}| \neq 0$:
  - n^{δ}=*get*(N^{δ}); $\forall p \in P$: Falls p sensitiv auf n^{δ} ist, dann füge p in R ein; *del*(n^{δ},N^{δ});
- *nextTime()*: Gib die Simulationszeit τ^{next} der frühesten Timed Notification / des frühesten Timeouts zurück.
- *getNext()*: Gib aus der Menge N^{τ} dasjenige n^{τ} mit dem kleinsten Zeitstempel zurück.
- *tnotify(τ,N^{τ})*: Führe folgende Sequenz aus solange *nextTime()*$\equiv \tau$:
  - n^{τ}=*getNext()*; $\forall p \in P$: Falls p sensitiv auf n^{τ} ist, dann füge p in R ein; *del*(n^{τ},N^{τ});

Das SystemC Scheduling ist in Abb. 2.12 anhand einer Zustandsmaschine dargestellt. Die Zustandsmaschine modelliert jede Kernelphase mit einem separaten Zustand. Immediate Notifications sind implizit durch die Rückkopplung im s^{eval} Zustand modelliert. Die Abarbeitung von Delta Notifications entspricht dem Durchlaufen der sog. *inneren Schleife* von $s^{dnotify}$ nach s^{eval}. Die Abarbeitung von Timed Notifications entspricht dem Durchlaufen der *äußeren Schleife* von $s^{tnotify}$ nach s^{eval}.

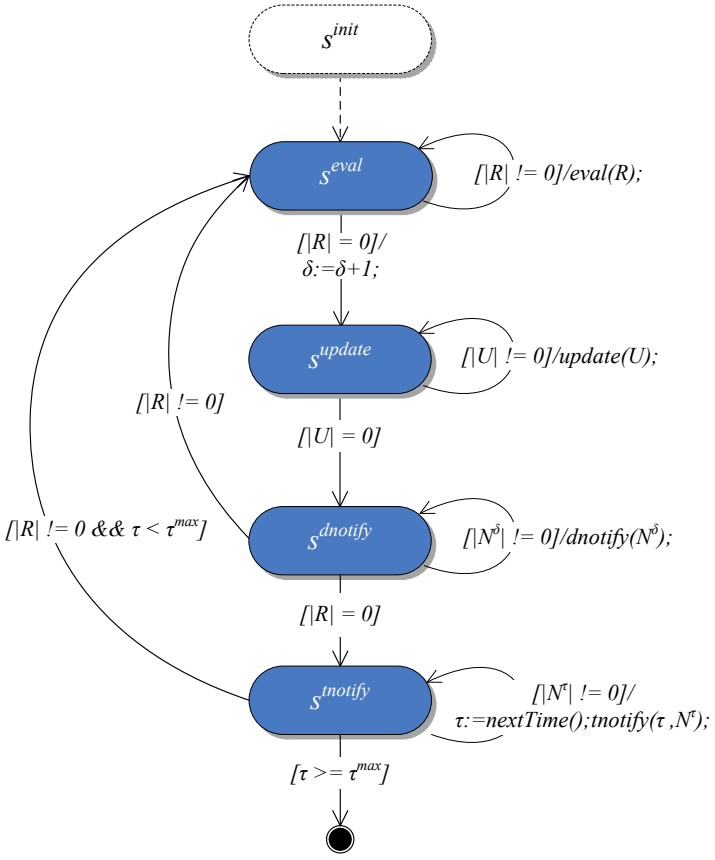


Abbildung 2.12.: Sequentieller SystemC Scheduler

2.3.2.3. Modellierung auf Transaktionsebene

Transaction Level Modeling [74, 100] ist eine Methode zur Anhebung der Abstraktionsebene über das RTL hinaus auf das ESL (vgl. Abschnitt 2.1.1.2). Der Begriff entstammt der Domäne der SLD Sprachen wie SpecC oder SystemC, weshalb TLM insbesondere in Kombination mit diesen Sprachen verbreitet ist [74]. TLM Konzepte bilden dabei die Basis unterschiedlichster SLD Entwurfsmethodiken (z.B. [114, 77, 213, 60]). Ein guter Überblick über existierende akademische SLD Ansätze unter Verwendung von TLM ist in [120] zu finden.

Die grundlegende Idee von TLMs besteht darin, Kommunikation zwischen Komponenten nicht mehr wie in einem RTL Modell mit Hilfe einzelner Signale zu modellieren, sondern mit Hilfe von Transaktionen. Eine Transaktion wird in [72] als

> *„... the longest communication during which the invariant data, as set by a system master, remains valid“*

definiert. Dabei werden direkte Methodenaufrufe verwendet, durch die Transaktionsobjekte zwischen Modulen ausgetauscht werden können (nicht-restriktierte Kommunikation). Neben den zu übertragenden Daten können solche Transaktionsobjekte u.a. einen Zeitstempel enthalten, der die Dauer einer oder mehrerer Phasen einer Transaktion spezifiziert. Da die Kommunikation nicht mehr unter der Kontrolle des Kernels ist, müssen Zeitinformationen aus den Zeitstempeln abgeleitet werden. Der Vorteil des TL gegenüber dem RTL liegt vor allem in der weit besseren Performanz, da Synchronisationsaufwand mit dem Kernel eingespart wird.

Syntaktische Komponenten

Abb. 2.13 illustriert ein typisches TL Modell bestehend aus drei Modulen. Diese können in Initiator und Target Module klassifiziert werden. Das als Interconnect bezeichnete Modul ist Initiator und Target zugleich. Der Modultyp wird durch den Typ des/der Sockets bestimmt, auf das/die ein Modul Zugriff hat.

Die Art und Weise der Modellierung innerhalb eines Moduls ist im SystemC/TLM Standard nicht definiert. Typischerweise werden Initiator mit einem oder mehreren *SC_THREAD* Prozessen modelliert. Target Module können abhängig vom Modellierungsstil (siehe unten) auch rein passiv sein, d.h. sie enthalten keinen Prozess, sondern nur Schnittstellenmethoden.

Kommunikation erfolgt durch direkte Methodenaufrufe auf dem *Vorwärtspfad* oder auf dem *Rückwärtspfad* über Sockets. Ein Initiator Socket stellt einen Port zum Aufruf von Schnittstellenmethoden im Target unter Nutzung des Vorwärtspfades zur Verfügung. Zugleich stellt es eine weitere Schnittstelle bereit, über die

das Target auf dem Rückwärtspfad Schnittstellenmethoden im Initiator aufrufen kann. Ein Target Socket ist das Gegenstück zum Initiator Socket [27].

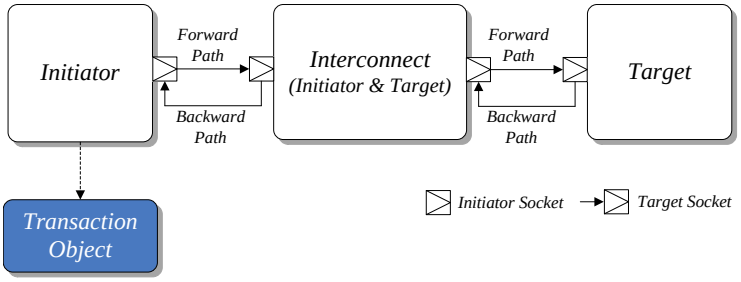


Abbildung 2.13.: Kommunikation zwischen Initiator und Target (Quelle: [27])

Modellierungsstile

Der Begriff des Transaction Levels steht nicht für eine bestimmte Abstraktionsebene mit einem bestimmten Detaillierungsgrad, sondern bezeichnet vielmehr ein Kontinuum unterschiedlicher Abstraktionsgrade. Ein TL Modell kann durch einen Punkt innerhalb dieses Kontinuums charakterisiert werden [100, 72]. Zum aktuellen Zeitpunkt existiert keine allgemein anerkannte Terminologie zur präzisen Definition und Charakterisierung solcher Punkte innerhalb dieses Kontinuums [219]. Mit anderen Worten: Es existiert kein eindeutig spezifiziertes Berechnungsmodell für TL Modelle. Die TLM 2.0 Spezifikation des SystemC Standards [27] unterscheidet allerdings zwischen sog. *Modellierungsstilen* oder *Coding Styles* (Abb. 2.14 mitte), mit deren Hilfe TL Modelle genauer eingeordnet werden können.

Coding Styles beschreiben Alternativen zur Nutzung der vom TLM Standard zur Verfügung gestellten API (vgl. Abb. 2.14 unten). Jeder Coding Style beschreibt eine Reihe von Möglichkeiten, um bis zu einem gewissen Grad zwischen höherer und geringerer (zeitlicher) Genauigkeit zu skalieren. Dadurch lassen sich Modelle realisieren, die für unterschiedliche Anwendungsfälle geeignet sind (Abb. 2.14 oben). Die in [27] beschriebenen Coding Styles sind:

- **Loosely-Timed (LT)**: Dieser Coding Style nutzt das Blocking Transport Interface (*b_transport()* Methode) zur Kommunikation. Transaktionen sind blockierend, d.h. die Methodenaufrufe blockieren solange, bis eine Transaktion abgeschlossen ist. Diese Dauer kann durch einen Zeitstempel annotiert werden. Synchronisation mit dem Kernel oder anderen Prozessen erfolgt durch *wait()* Aufrufe. Mit Hilfe von *Temporal Decoupling* können Prozesse in der Simulationszeit maximal bis zu einem sog. *Global Quantum* vorauseilen, ohne mit anderen Prozessen bzw. dem Kernel zu synchronisieren.

Ähnlich einem asynchronen PDES Algorithmus (siehe Abschnitt 2.2.3.4) besitzen Module zu diesem Zweck eine lokale Zeit, die im Fall von Temporal Decoupling bis zum Erreichen des globalen Quantums inkrementiert wird. Der reduzierte Synchronisationsoverhead resultiert in einer höheren Performanz.

Die Verwendung des LT Coding Styles ist in der Regel mit einem hohen Genauigkeitsverlust verbunden, da durch die blockierenden Methodenaufrufe keine simultanen Transaktionen modelliert werden können. Der Genauigkeitsverlust wird durch Temporal Decoupling weiter erhöht. Aufgrund der geringen zeitlichen Akkuratheit ist der LT Coding Style für die Entwicklung virtueller Plattformen zur Softwareentwicklung und weniger zur Exploration oder Verifikation von Hardware geeignet.

- **Approximately-Timed (AT)**: Dieser Coding Style nutzt das Non-Blocking Transport Interface (*nb_transport_fw()*, *nb_transport_bw()* Methoden). Transaktionen sind in mehrere Phasen eingeteilt. Diese sind durch nicht-blockierende Funktionsaufrufe markiert. Die Phasen spezifizieren die Synchronisationspunkte mit dem Kernel. Dazu muss für jede Phase separat eine Zeitdauer festgelegt werden. Die Phasenaufteilung ermöglicht die Modellierung simultaner Transaktionen. Temporal Decoupling wird in der Regel nicht unterstützt. Durch die höhere zeitliche Genauigkeit, die aus dem AT Coding Style resultiert, ist dieser zur Architekturexploration und Analyse gut geeignet. Die geringe Geschwindigkeit macht AT Modelle weniger geeignet für die Softwareentwicklung.

Eine rein funktionale Spezifikation (*Untimed (UT)* Coding Style) ist ein Sonderfall des LT Coding Styles, bei dem auf eine Modellierung der Zeit grundsätzlich verzichtet wird. Vollständig zyklenakkurate Modellierung ist nicht Teil von TLM 2.0. Ein zyklenakkurater Codingstyle könnte laut SystemC/TLM Standard [27] als Erweiterung des AT Coding Styles definiert werden.

Alternative Klassifikationen von TL Modellen finden sich z.B. in [74] oder [100]. Cai und Gajski charakterisieren TL Modelle in [74] separat hinsichtlich der zeitlichen Genauigkeit von Kommunikation und Berechnung und unterscheiden diesbezgl. zwischen fünf verschiedenen Modellierungsstilen. Donlin differenziert in [100] explizit zwischen RTL Modellen und zyklenakkuraten Modellen. Nach seiner Klassifikation sind zyklenakkurate Modelle Teil des TL.

2.3.3. Sprachen zur Modellierung heterogener eingebetteter Systeme

Um das Verhalten von heterogenen Systemen eindeutig spezifizieren und verifizieren zu können, bedarf es einer Möglichkeit, Kompositionen und Interak-

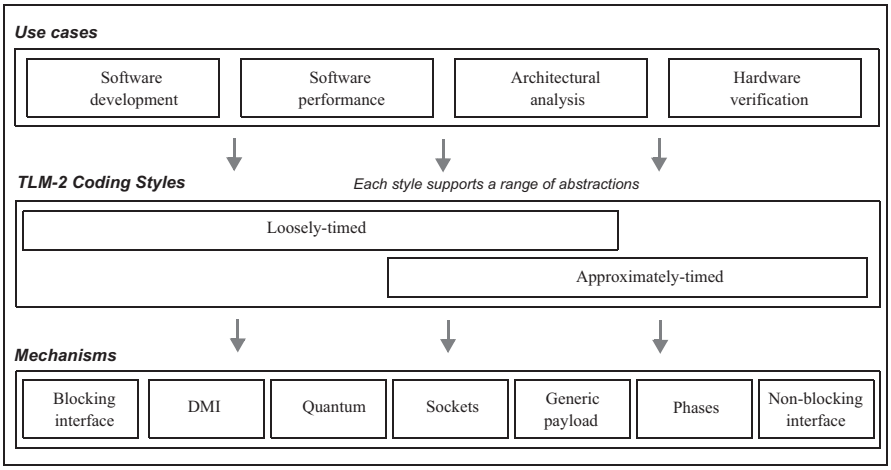


Abbildung 2.14.: Anwendungsfälle, Coding Styles und Mechanismen von TLM 2.0 (Quelle: [27])

tionen zwischen Modellen, die auf unterschiedlichen Berechnungsmodellen basieren, möglichst exakt zu beschreiben. Lee und Vincentelli führen in diesem Zusammenhang in [176] an, dass die Verwendung von nur einem einzigen allgemeingültigen Berechnungsmodell zur vollständigen Beschreibung eines Systems zur Folge hätte, dass Eigenschaften des zu entwickelnden Systems nur noch durch umfangreiche Simulationen und nicht mehr durch formale Methoden überprüfbar wären. Umgekehrt argumentieren Eker et al. in [102], dass eine unorganisierte amorphe Heterogenität von Komponenten die Analyse von Wechselwirkungen zwischen Teilsystemen außerordentlich schwierig gestalten kann, da die Komposition heterogener Modelle u.U. in nicht vorhersehbarem emergentem Verhalten resultiert. Ein solches Verhalten kann sich entsprechend [102] z.B. dann einstellen, wenn eine „brute-force“ Komposition heterogener Simulationsmodelle aufgrund nicht vorhersehbarer Charakteristika verschiedener Berechnungsmodelle ungewollte Wechselwirkungen auslöst. Insbesondere eine Co-Simulation (vgl. Abschnitt 2.2.4) ist anfällig dafür.

Aus den genannten Gründen ist die exakte Beschreibung von Beziehungen zwischen heterogenen Berechnungsmodellen und die Untersuchung von Eigenschaften unterschiedlich heterogener Berechnungsmodelle und deren Interaktionen ein aktuelles Forschungsthema. Aus unterschiedlichen Projekten sind in diesem Zusammenhang bereits eine ganze Reihe sog. formaler Frameworks hervorgegangen. Diese basieren meist auf einem

1. **mathematischen Formalismus**, der sich zur Beschreibung, formalen Verifikation, Synthese (von Teilen) und Simulation eines Modells eignet, das auf Basis heterogener Berechnungsmodelle spezifiziert wurde. Der mathematische Formalismus definiert typischerweise ein einheitliches Metamodell für verschiedene Berechnungsmodelle.
2. **heterogenen Werkzeug** zur *Modellierung und Simulation (M&S)*, das zur experimentellen simulationsbasierten Analyse eines Modells dient. Die dem Werkzeug zugrundeliegende Sprache basiert auf dem mathematischen Formalismus.

Im Unterschied zu SLDL Ansätzen wie z.B. SystemC/TLM, existiert in diesen Frameworks eine eindeutige Definition für verschiedene Berechnungmodelle sowie deren Interaktionen. Beispiele formaler Frameworks sind Ptolemy II [102, 255, 175] und dessen Vorgänger Ptolemy [70], Metro II [95, 94] und dessen Vorgänger Metropolis [45, 175] sowie ForSyDe [226]. Beispiele für heterogene Erweiterungen von SystemC sind HetSC [136] oder die Arbeit von Patel und Shukla in [212]. Aufgrund der Relevanz in Kapitel 5, werden im Folgenden die wichtigsten Grundlagen von Ptolemy II erläutert.

2.3.4. Ptolemy II

Ptolemy II (PtII) [102] ist ein Werkzeug zur Modellierung und Simulation heterogener Systeme in Java. Das PtII zugrundeliegende Konzept zum Management der Heterogenität wird als hierarchische Heterogenität bezeichnet. Diese ist ein spezieller Ansatz zum Management horizontaler Heterogenität (vgl. Abschnitt 2.2.4), bei dem heterogene Teilmodelle nur durch hierarchische Komposition mit der Folge zusätzlicher vertikaler Heterogenität kombiniert werden können.

2.3.4.1. Syntaktische Komponenten

Die abstrakte Syntax von PtII erlaubt (ähnlich wie die SystemC Syntax) eine Strukturierung von Modellen in hierarchisch geclusterte Graphen [185]. Die Knoten eines solchen Graphen bilden sog. *Actors*, die über *Ports*, *Links* und *Relations* miteinander verknüpft sind. Die abstrakte Syntax kann durch unterschiedliche Formen einer konkreten Syntax repräsentiert werden. In PtII existiert eine konkrete visuelle Syntax (siehe Abb. 2.15) sowie ein *XML* Dialekt namens *Modeling Markup Language (MoML)* [177].

Actors können atomare (*B*, *D* und *E* in Abb. 2.15) oder *komposite Actors* (*TopLevel*, *A* und *C*) sein. Am unteren Ende der Hierarchie befinden sich ausschließlich *atomare Actors*. Ein Netzwerk von (kompositen) Actors kann wiederum in einen kompositen Actor eingebettet sein. Ein kompositer Actor erlaubt so die hierarchi-

sche Verschachtelung von beliebigen (atomaren und kompositen) Actors. Jeder komposite Actor besitzt entweder genau ein Director Attribut oder keines. Die Toplevelebene besitzt immer einen Director.

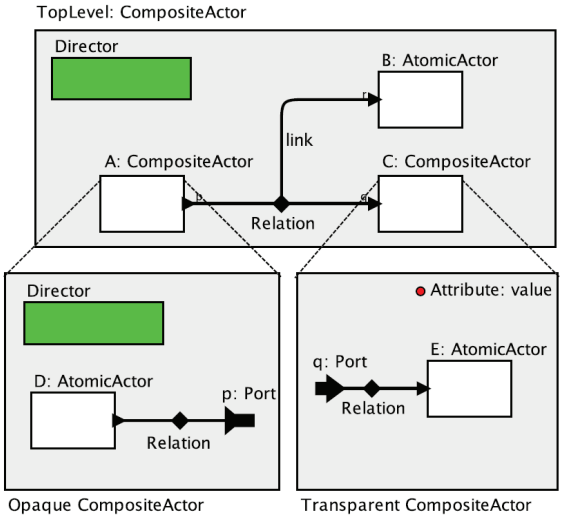


Abbildung 2.15.: Konkrete visuelle Syntax von Ptolemy II (Quelle: [217])

2.3.4.2. Abstrakte Semantik

PtII unterscheidet zwischen abstrakter und konkreter Semantik [217]. Die abstrakte Semantik spezifiziert Regeln zur Ausführung und Kommunikation zwischen Actors, die allgemein gültig sind. Sie wird auch als *Actor Semantics* bezeichnet. Ein Berechnungsmodell entspricht einer konkreten Semantik. Deren Implementierung wird auch als *Domäne* oder *Domain* bezeichnet. Die Existenz einer abstrakten Semantik ist grundlegend für die eindeutige Beschreibung der Interaktion zwischen heterogenen Domänen. Im Folgenden werden die grundlegenden Aspekte der abstrakten PtII Semantik erläutert.

Zeitmodell

Das Zeitmodell von PtII wird als Superdense Time bezeichnet. Dabei wird die Zeit, ähnlich zum Zeitmodell aus Kapitel 4.5, als ein Tupel $t^{\tau,\mu} = (\tau, \mu)$ repräsentiert. τ ist die sog. Modeltime und μ der Microstep. Zwei Zeitstempel (τ_1, μ_1) und (τ_2, μ_2) heißen *schwach gleichzeitig*, wenn $\tau_1 = \tau_2$ und *stark gleichzeitig* wenn zusätzlich $\mu_1 = \mu_2$ gilt. Bzgl. der zeitlichen Dauer ist ein Microstep in PtII damit identisch zu einem Deltacycle in SystemC (vgl. Abschnitt 2.3.2). Das Zeitmodell

ist die Grundlage für den Erhalt von Determinismus, indem die Ausführungsreihenfolge der Actors vollständig geordnet wird.

Auf Grundlage der Superdense Time wird der Zeitverlauf innerhalb eines PtII Modells hierarchisch gesteuert: Typischerweise wird ein Zeitfortschritt nur auf der obersten Hierarchieebene durchgeführt. Untere Ebenen erhalten den aktuellen Zeitwert $t^{\tau,\mu}$ des Modells dann vom höheren Ebenen [217].

Spezifikation von Verhalten durch Actors

Aktors sind prinzipiell ausführbare nebenläufige Komponenten. Sie kommunizieren über Relationen. Interaktionen entsprechen dem Austausch sog. Tokens. Die Art und Weise der Ausführung und Kommunikation ist durch die Domäne in dem kompositen Actors definiert, in der der Actor instanziiert ist. Komposite Actors können als undurchlässig (engl. opaque) oder als transparent definiert werden. Im ersten Fall verhalten sie sich wie eine Blackbox, bei der die interne Spezifikation der Struktur und des Verhaltens nach außen hin nicht sichtbar ist. Im zweiten Fall ist die komplette interne strukturelle Spezifikation nach außen hin sichtbar. Die konkrete Semantik entspricht der des umgebenden kompositen Actors.

Kommunikationsmechanismen

Die Art und Weise der Kommunikation ist in der abstrakten Semantik noch nicht festgelegt. Mit Hilfe unterschiedlicher Typen von sog. Receiver können Ports auf einen für eine bestimmte Domäne geeigneten Kommunikationsmechanismus spezialisiert werden. Dazu wird Senden eines Tokens per Aufruf von *send()* auf dem Ausgangsport an die *put()* Methode des zugehörigen Receivers delegiert (siehe Abb. 2.16). Beispiele unterschiedlicher Kommunikationsmechanismen sind *FIFOs*, *Mailboxen* oder *Queues* [217].

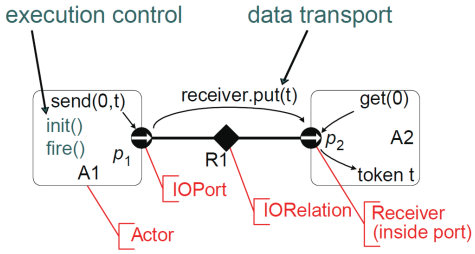


Abbildung 2.16.: Kommunikation in Ptolemy II (Quelle: [217])

Ausführungsmechanismus

Der Ausführungsmechanismus einer Domäne wird innerhalb eines kompositen Actors durch einen Director festgelegt. Das abstrakte Ausführungsschema aller Directors von PtII ist in Abb. 2.17 anhand eines Beispiels dargestellt. Die Ausführung lässt sich, ähnlich wie die Ausführung eines SystemC Modells (vgl. Abschnitt 2.3.2.2), in verschiedene Ausführungsphasen einteilen, die *Initialization Phase*, die *Execution Phase* und die *Wrapup Phase*. Diese Phasen lassen sich wiederum in Teilphasen untergliedern, für die jeder Actor und Director entsprechende Callback-Methoden (Action Methods) besitzt.

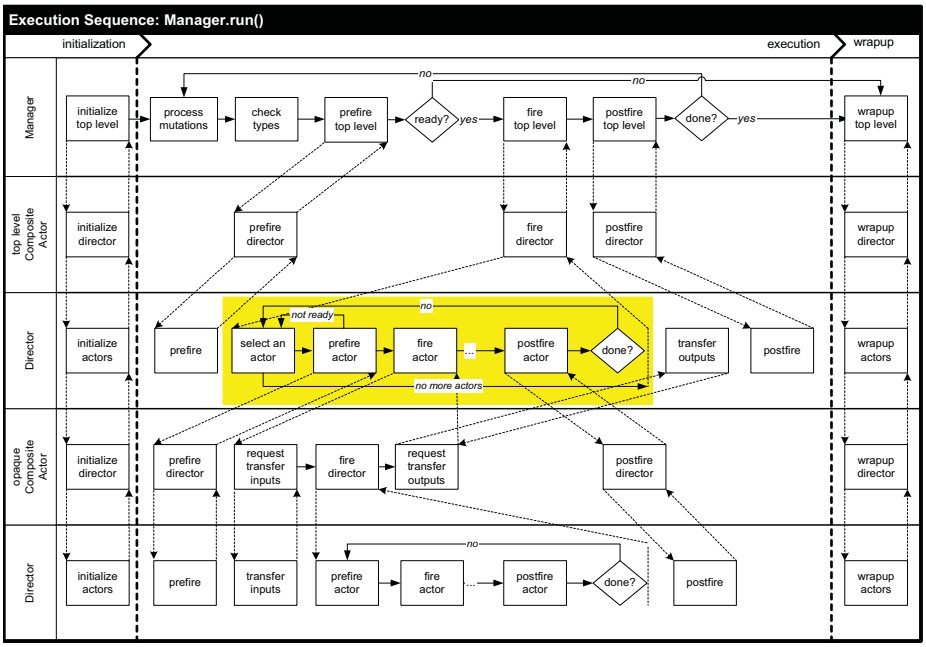


Abbildung 2.17.: Ausführung eines PtII Modells (Quelle: [217])

Die Initialization Phase besteht aus *preinitialize()* und *initialize()*. Deren Aufgabe ist die Initialisierung von Modellparametern und Zuständen. In der Execution Phase wird das Modell durch wiederholte *Iteration* über die Aktionen *prefire()*, *fire()* und *postfire()* ausgeführt. In *prefire()* werden mögliche Vorbedingungen für den Aufruf von *fire()* getestet. Das eigentliche Verhalten eines Actors (lesen von Tokens, Berechnung, Schreiben von Tokens) ist in *fire()* implementiert. In *postfire()* wird der lokale Zustand des Actors aktualisiert. Die Tatsache, dass der Zustand eines Actors erst in *postfire()* aktualisiert wird, ist grundlegend für die Existenz sog. domänenpolymorpher Actors (vgl. [217]). Die Ausführung der Wrapup

Phase erfolgt durch Aufruf der *wrapup()* Aktion. Sie signalisiert das Ende der Ausführung eines Actors oder Directors.

2.3.4.3. Konkrete Semantik

Spezialisierte Domänen können grundsätzlich auf Basis bereits existierender weniger spezieller Domänen entwickelt werden. Ein Ansatz dazu ist es, durch objektorientierte Ableitung von den relevanten Klassen (insbesondere von einer geeigneten Director Klasse) eine neue Variante zu implementieren. Durch den Mechanismus der Ableitung wird sichergestellt, dass die Eigenschaften der ursprünglichen Domäne sowie deren Kompatibilität zur anderen Domänen erhalten bleibt. Wegen ihrer besonderen Relevanz in Kapitel 5 werden im Folgenden einige wichtige Besonderheiten der DE Domäne von PtII kurz erläutert.

2.3.4.4. Spezifika der DE Domäne

Das DE Berechnungsmodell von PtII ist, wie das SystemC Berechnungsmodell, eine Variante des Basisalgorithmus aus Abschnitt 2.2.3.3). Details zur Implementierung sind in [67] zu finden. In der DE Domäne kommunizieren Actors über Events. Ein Event ist eine Kombination aus einem Token und einem Tag. Das Token speichert Daten und der Tag speichert einen Zeitstempel entsprechend dem Zeitmodell aus Abschnitt 2.3.4.2.

Der DE Director verwaltet eine globale Eventqueue. Das Auftreten eines Ereignisses ist äquivalent zum Empfang eines Tokens über einen mit dem DE Receiver spezialisierten Port (vgl. Abschnitt 2.2.3.3). Daneben gibt es sog. Pure Events. Mit deren Hilfe kann ein Actor zu einem späteren Zeitpunkt erneut gefeuert werden, ohne dass dies mit dem Empfang eines Tokens verbunden ist.

In einer SystemC RTL Simulation genügt ein Zweiertupel bestehend aus Simulationszeit und Deltacycle, um deterministische Simulationsergebnisse zu erzielen. Der Grund ist das Evaluate/Update Paradigma: Dieses sorgt dafür, dass alle in einem Deltacycle n generierten Wertänderungen eines Signals grundsätzlich erst im Deltacycle $n + 1$ (gleichzeitig) sichtbar werden.

Würde man in PtII DE ausschließlich Actors mit einer minimalen Verzögerung von einem Microstep verwenden, so genügte ebenfalls die Berücksichtigung von Model Time und Microstep bei der Sortierung von Ereignissen. In einem PtII Modell existieren jedoch meist auch Actors, die weder eine Verzögerung bzgl. τ noch eine Verzögerung bzgl. μ erzeugen. Dadurch kann es passieren, dass das Simulationsergebnis bei Existenz mehrerer Ereignisse mit gleichem τ und gleichem μ von der Reihenfolge abhängt, in der Actors gefeuert werden. Um dennoch deterministische Simulationsergebnisse garantieren zu können, wird in

der DE Domäne zusätzlich der Level von Actors berücksichtigt, wodurch eine vollständige Ordnung mit einer eindeutigen Ausführungsreihenfolge von Actors entsteht. Das DE Berechnungsmodell erweitert dazu das Superdense Time Zeitmodell zu einem Dreiertupel (τ, μ, λ). Dabei spezifiziert λ den Level.

Der Level von Actors wird statisch durch eine topologische Sortierung der Actors bestimmt, die Teil des Actor-Graphen $G_A(A, R)$ im betrachteten DE Modell sind. Eine Sortierung ist nur dann möglich, wenn die Topologie einen gerichteten azyklischen Graphen (engl. *Directed Acyclic Graph (DAG)*) aufspannt [67, 152]. Existiert der DAG eines DE Modells, dann ist der Level eines Actors durch dessen Position innerhalb des DAG definiert. Dabei hat ein sog. Upstream Actor (ein Actor, der sich näher an der Wurzel des DAG befindet) einen kleineren Level als ein Downstream Actor. Ein DAG existiert nicht, wenn innerhalb der Topologie verzögerungsfreie Zyklen zwischen Actors existieren. Ein solches Modell ist in PtII grundsätzlich nicht ausführbar. PtII verweigert in diesem Fall die Ausführung. Die Ausführbarkeit kann hergestellt werden, indem verzögerungsfreie Zyklen mit Verzögerungsgliedern von mindestens einem Microstep in verzögerungsbehaftete Zyklen umgewandelt werden.

2.4. Prozessorarchitekturen und Parallelität

2.4.1. Taxonomie für Prozessorarchitekturen

Eine der ersten Klassifikationen für Prozessorarchitekturen stammt aus dem Jahr 1966 von Michael J. Flynn [109]. Flynn klassifiziert Computersysteme entsprechend der Matrix aus Abb. 2.18. Die Klassifikation leitet sich von der Anzahl der innerhalb einer Architektur vorhandenen parallelen Kontroll- und Datenflüsse ab. Im Falle mehrerer paralleler Kontroll- und/oder Datenflüsse existieren entsprechend mehrere parallele Recheneinheiten zu deren Verarbeitung.

- **Single Instruction Single Data (SISD)**: Das SISD Paradigma ist die Grundlage traditioneller Prozessorarchitekturen, die entweder entsprechend einer Von-Neumann oder Harvard-Architektur aufgebaut sind. Es wird weder Parallelität im Datenstrom noch im Befehlsstrom genutzt. Es existiert nur ein einziger Datenstrom als Eingang in eine zentrale Recheneinheit. Während eines Taktzyklus wird von der zentralen Recheneinheit nur ein einziger Befehl ausgeführt.
- **Single Instruction Multiple Data (SIMD)**: Alle Recheneinheiten führen in jedem Takt den gleichen Befehl aus. Dabei kann jede Recheneinheit auf unterschiedlichen Daten arbeiten. Die Synchronität der Befehlsausführung

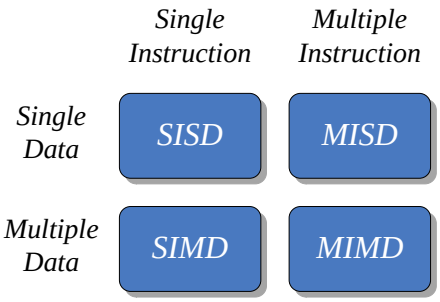


Abbildung 2.18.: Klassifikation von Computerarchitekturen nach Flynn

ist die Basis für Determinismus. Aktuelle Grafikprozessoren arbeiten nach diesem Prinzip.

- **Multiple Instruction Single Data (MISD)**: Jede Recheneinheit führt andere Befehle auf einem für alle Recheneinheiten gleichen Datenstrom aus. Bisher existieren wenige Beispiele für Architekturen, die diesem Typ zugeordnet werden können.
- **Multiple Instruction Multiple Data (MIMD)**: Jede Recheneinheit führt einen eigenen Befehlsstrom aus und arbeitet zugleich auf einem separaten Datenstrom. Abhängig von der Art der Implementierung arbeiten MIMD Architekturen synchron oder asynchron. Wenn keine entsprechenden Vorkehrungen im Sinne der Synchronisation und Koordination paralleler Recheneinheiten getroffen werden, ist eine korrekte Funktion nicht garantiert. MIMD Architekturen sind die am weitesten verbreiteten parallelen Architekturen.

2.4.2. Klassifikation von Parallelität

Mit den genannten Klassen von Prozessorarchitekturen können verschiedene Arten von Parallelität innerhalb einer Applikation mehr oder weniger effizient ausgenutzt werden. Hennessy und Patterson unterscheiden in [133] zwei grundlegende Klassen von Parallelität innerhalb von Applikationen, *Data-Level Parallelism (DLP)* und *Task-Level Parallelism (TLP)*.

DLP entsteht durch voneinander unabhängige Datenelemente, die aufgrund ihrer Unabhängigkeit zur gleichen Zeit verarbeitet werden können. Im Gegensatz dazu spricht man von TLP im Kontext von Funktionen, die aufgrund ihrer funktionalen Unabhängigkeit weitgehend parallel ausgeführt werden können. DLP und TLP lassen sich in einem bestimmten betrachteten Kontext weiter verfei-

nern. Hennessy und Patterson nennen orthogonal zur Klassifikation von Flynn mehrere mehr hardwarespezifische Typen von Parallelität, die sich von DLP und TLP ableiten. Zu den hier relevanten gehören:

- **Instruction Level Parallelism (ILP)**: Diese Art der Parallelität nutzt DLP zur Extraktion von feinkörniger Parallelität innerhalb eines Befehlsstroms. Entsprechende Techniken hierzu finden sich in Mikroarchitekturen von Prozessoren wieder. Beispiele sind Pipelining, spekulative Befehlsausführung oder *Hardwareseitiges Multithreading (HMT)* bzw. *Hyper-Threading* [101].
- **Thread-Level Parallelism (THLP)**: THLP nutzt die in einer Applikation vorhandene DLP oder TLP zur parallelen Ausführung der Applikation verteilt auf mehrere grobkörnige Funktionen, die Thread oder Prozess genannt werden.

Die in dieser Arbeit für die Simulation eingesetzten Zielplattformen lassen sich als MIMD Architekturen klassifizieren und implementieren Techniken zur Nutzung von ILP und THLP. Der Fokus der entwickelten Softwaretechniken liegt hingegen ausschließlich auf der Verbesserung des THLP.

2.4.2.1. Parallelität auf Threadebene

Ein Thread oder Prozess ist ein separater Befehlsstrom, der auf einem oder mehreren Datenströmen arbeitet. Die Ausnutzung von Parallelität auf Threadebene setzt voraus, dass eine Applikation in mehrere Threads zerlegbar ist. Je nach Applikation können diese dann mehr oder weniger unabhängig voneinander abgearbeitet werden. Der Grad der Unabhängigkeit steht in direkter Relation zur vorhandenen Parallelität. Durch spezielle Synchronisationsmechanismen muss die parallele Ausführung gezielt limitiert werden, um Datenabhängigkeiten korrekt aufzulösen und funktionale Korrektheit herzustellen.

Techniken zur Ausnutzung von Parallelität auf Threadebene basieren typischerweise auf MIMD Architekturen, da diese eine Infrastruktur für mehrere parallele Befehls- und Datenströme bereitstellen. Betrachtet man die Implementierung von MIMD Architekturen etwas genauer, so werden im Allgemeinen mehrere Recheneinheiten und Speichermodule über ein Verbindungsnetzwerk miteinander gekoppelt. MIMD Architekturen können in die Klasse der *Distributed-Memory (DM)* Architekturen sowie der *Shared Memory (SHM)* Architekturen unterteilt werden [247, 103] (siehe Abb. 2.19). Erstere werden auch als Multiprozessoren oder Multicore Prozessoren, letztere als Multicomputer oder Distributed Memory Multiprozessoren bezeichnet [247, 139, 211].

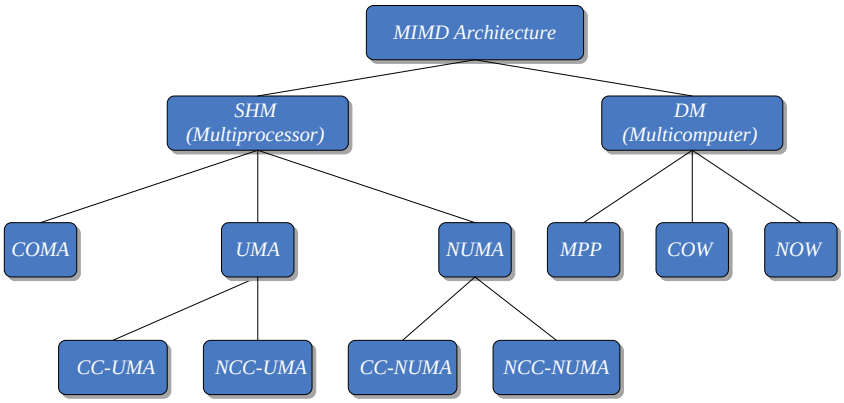


Abbildung 2.19.: Verfeinerte Klassifikation von MIMD Architekturen (nach [247])

2.4.2.2. Distributed-Memory MIMD Architekturen

In einem DM System ist jede Recheneinheit mit einem lokal zugreifbaren Speicher ausgestattet. Kommunikation zwischen Recheneinheiten wird über den Austausch von Nachrichten realisiert. Das Verbindungsnetzwerk zum Austausch der Nachrichten kann auf unterschiedlichste Weise realisiert sein, beispielsweise als als Computernetzwerk, das geografisch verteilte Rechner miteinander verbindet. In die Klasse der DM Architekturen gehören u.a. sog. *Massively Parallel Processors (MPP)*, *Clusters of Workstations (COW)* und *Networks of Workstations (NOW)*.

Zum Nachrichtenaustausch muss dem Programmierer eine spezielle API zur Verfügung gestellt werden. Eine der bekanntesten APIs ist der *Message Passing Interface (MPI)* Standard [110]. Kommunikation und Synchronisation erfolgt über definierte Methodenaufrufe. Mit deren Hilfe können unterschiedliche sog. *Kommunikationsmuster* wie *One-Sided Communication*, *Two-Sided Communication* oder *Collective Communication* zwischen parallelen Prozessen realisiert werden. Kommunikationsmuster sind durch Syntax und Semantik des Datenaustauschs zwischen einzelnen oder einer Gruppe von beteiligten MPI Prozessen definiert[1].

[1] Aus dem Blickwinkel der Beschreibung aus Abschnitt 2.2.3 definiert der MPI Standard damit ein (sehr allgemeines) Berechnungsmodell zur verteilten Ausführung.

2.4.2.3. Shared-Memory MIMD Architekturen

In einer SHM Architektur kommunizieren mehrere Recheneinheiten über einen globalen Speicherbereich und einen gemeinsamen logischen Adressraum. Dieser ist allen Recheneinheiten über normale Speicheradressierung zugänglich. Der globale Speicher kann aus mehreren Speicherelementen bestehen, die mit den Recheneinheiten über ein mehr oder weniger komplexes Netzwerk verbunden sind. Im einfachsten Fall ist das Verbindungsnetzwerk ein Bus. Komplexere Verbindungsnetzwerke bestehen aus mehreren hierarchisch organisierten Bussen, einer Crossbar oder einem *Network-on-Chip (NoC)* [50].

Jede Recheneinheit hat typischerweise die gleichen Zugriffsrechte auf den globalen Speicher. Bei zugleich identischen Zugriffszeiten spricht man auch von einem *Unified Memory Access (UMA)* System. Im Gegensatz dazu spricht man von einem *Non-Unified Memory Access (NUMA)* System, wenn sich die Zugriffszeiten von verschiedenen Recheneinheiten auf den Speicher unterscheiden. NUMA Architekturen werden auch als *Distributed Shared Memory (DSM)* Architekturen bezeichnet [208]. Tatsächlich sind sie eine Hybridlösung, die DM Prinzipien mit SHM Prinzipien verbindet. Abhängig davon, ob ein sog. Cachekohärenzprotokoll in Hardware implementiert ist oder nicht, können UMA und NUMA Prozessoren in weitere Unterkategorien eingeteilt werden[139]:

- **Cache-Coherent / Non-Cache-Coherent UMA (CC/NCC-UMA)**
- **Cache-Coherent / Non-Cache-Coherent NUMA (CC/NCC-NUMA)**

Cachekohärenz bedeutet, dass Prozessoren eine gemeinsame Sicht auf bestimmte Stellen im Speicher haben müssen. Sie wird durch das Cachekohärenzprotokoll (z.B. *MESI* [133]) sichergestellt. Ist kein Cachekohärenzprotokoll implementiert, muss die Kohärenz softwareseitig sichergestellt werden. Eine *Cache Only Memory Access (COMA)* Architektur ist ein Sonderfall einer CC-NUMA Architektur, bei dem jeglicher Shared Memory im System ein Cache ist.

Architekturtechniken zur Synchronisation

Die korrekte Funktion einer parallelen Applikationen basiert meist auf der Existenz einer partiellen Ordnung von Interaktionen bzw. Speicherzugriffen von parallelen Threads [236]. Eine solche partielle Ordnung kann durch eine gezielte Synchronisation zwischen Threads und sog. wechselseitigen Ausschluss [37] von Zugriffen auf ein und dieselbe Speicherstelle hergestellt werden. Im Bereich der Multiprozessoren existieren unterschiedliche Basislösungen in Form spezieller sog. *Read Modify Write (RMW)* Instruktionen, die als Ausgangspunkt für komplexere Synchronisationsmethoden genutzt werden können. RMW Instruktionen sind eine Klasse von atomaren Befehlen, die es ermöglichen, gleichzeitig

von einer Speicheradresse zu lesen und auf sie zu schreiben. Beispiele für RMW Instruktionen sind:

- **Test-and-Set (TNS)**: Eine TNS Instruktion erlaubt das Schreiben eines Wertes auf eine Speicheradresse bei gleichzeitigem Lesen des alten Wertes. Eine mögliche Anwendung ist die Bereitstellung einer Critical Section.
- **Compare-and-Set (CAS)**: Eine CAS Instruktion vergleicht den Inhalt einer Speicheradresse mit einem gegebenen Wert. Wenn beide übereinstimmen, dann wird der gegebene Wert an die Speicheradresse geschrieben. Anhand eines boolschen Rückgabewertes erkennt ein Prozess, ob das Schreiben erfolgreich war.

2.4.3. Single-chip Cloud Computer

Der Single-chip Cloud Computer [46] ist ein von Intel Labs entwickelter experimenteller Prozessor, der als Forschungsplattform im Kontext der Programmierung von paralleler Software für zukünftige Manycore Architekturen dient. Als Manycore werden Prozessoren bezeichnet, die viel mehr Kerne besitzen, als dies in traditionellen Architekturen üblich ist.

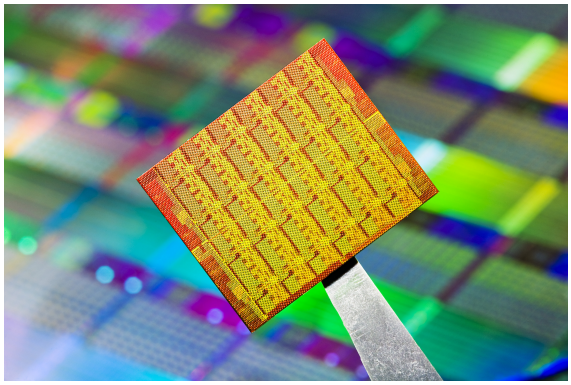


Abbildung 2.20.: Wafer des SCC (Quelle: www.intel.com)

Die Architektur des SCC kann als NCC-NUMA klassifiziert werden. Ein im Vergleich zu herkömmlichen cachekohärenten SHM Architekturen neuartiges Feature ist die Unterstützung von schneller On-Chip Kommunikation, basierend auf nicht cachekohärentem SHM. Es wurde bewusst auf eine Hardwareimplementierung eines Cachekohärenzprotokolls zu Gunsten einer in Software verwalteten Cachekohärenz verzichtet.

Abb. 2.21 gibt einen Überblick über die Gesamtarchitektur des SCC anhand eines Blockdiagramms. Er besteht aus 24 Tiles. Diese sind über ein zweidimensionales 6x4 Mesh NoC mit 256 GB/s Bisektionsbandbreite. Jedes Tile besteht wiederum aus zwei P54C Pentium^{TM} Kernen, 16 kB L1 Cache jeweils für Daten und Befehle pro Kern, einen für Daten und Befehle gemeinsamen 256 kB großen L2 Cache pro Kern eine *Mesh Interface Unit (MIU)*, zwei TNS Register und einen 16 kB SRAM basierten sog. *Message Passing Buffer (MPB)*. Die MIU verbindet jedes Tile mit einem Router, der einen XY-Routingalgorithmus implementiert [225]. Der Router wiederum integriert das Tile in das NoC. Der SCC ist ausgestattet mit vier integrierten DDR3 Memorycontrollern (IMCs), über die bis zu 64 GB Hauptspeicher adressiert werden kann. Zusätzlich existiert ein 8 Byte breites bidirektionales Highspeed *I/O* Interface, das für die Off-Chip-Kommunikation genutzt wird [92].

Die Kerne des SCC lassen sich entweder mit jeweils sparatem Linux *Operating System (OS)* (einem modifizierten 2.6.16 Linuxkernel) oder im sog. „BareMetalC" Modus (ohne OS) betreiben [198]. In dieser Arbeit wurden die Kerne des SCC grundsätzlich mit OS betrieben.

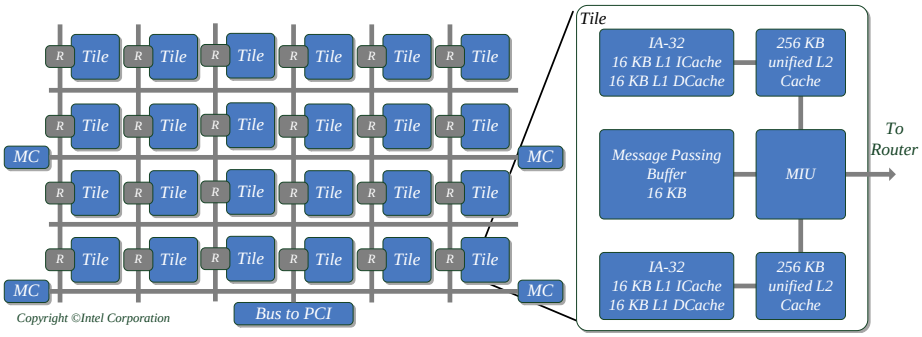


Abbildung 2.21.: SCC Architektur Blockdiagramm

2.4.3.1. Flexible Speicherarchitektur

Der SCC besteht aus mehreren unterschiedlichen Speicherbereichen. Sie wird durch den externen DDR Speicher, sowie den durch die MPB bereitgestellten On-Chip Speicher aufgespannt. Dabei können sowohl der DDR3 als auch der MPB Speicherbereich als Private Memory oder als Shared Memory deklariert werden. Durch Nutzung von *Lookuptables (LUTs)* kann der komplette Speicherbereich flexibel auf unterschiedliche Adressräume abgebildet werden. Zugriffe auf externen DDR3 oder MPB Speicher erfolgen immer über die MIU[198].

2.4.3.2. Prozessorkerne

Die P54C Kerne des SCC sind eine erweiterte Version von 32 bit Pentium Kernen der zweiten Generation [235]. Sie unterstützen keine Out-of-Order Ausführung von Befehlen. Die ursprünglichen 8 kB L1 Daten- und Befehlscaches wurden durch 16 kB 4-Wege satzassoziative Caches mit Write-Through und Write-Back Unterstützung ausgetauscht. Wie jede x86 Architektur basiert das Speichermanagement auch bei den SCC Kernen auf einem Pagetable Ansatz. Physikalische Speicheradressen eines Kerns werden mit Hilfe der Pagetable in systemweite Adressen übersetzt. Dies geschieht mit einer Granularität einer Page von 4 kB.

Für jede Page kann anhand spezieller Konfigurationsbits separat festgelegt werden, ob der Speicherzugriff mit oder ohne Cacheunterstützung erfolgen soll [198]. Die Wahl hängt davon ab, ob ein Speicherbereich ausschließlich als Private Memory oder gemeinsam mit anderen Kernen als Shared Memory verwendet werden soll. Falls nur Private Memory verwendet wird, so kann die vollständige L1 und L2 Cacheunterstützung aktiviert werden. Daten werden mit einer Granularität von 32 bit zwischen Caches und Speicher bewegt. Eine Kohärenzsicherung ist nicht notwendig. Falls Shared Memory verwendet wird, so muss die Kohärenz zwischen dem Speicher und den Caches der Kerne softwareseitig sichergestellt werden. Dazu existieren folgende Alternativen:

1. **Cached Mode (CM)**: Vollständige Aktivierung der L1 und L2 Cacheunterstützung. Sowohl L1 als auch L2 Cache werden zum Lesen und/oder Schreiben von Daten verwendet. Die einzige Möglichkeit, um Konsistenz herzustellen ist die Ausführung teurer L2 Cache Flushes.
2. **Uncached Mode (UCM)**: Vollständige Deaktivierung der L1 und L2 Cacheunterstützung: Weder L1 noch L2 Cache werden zum Lesen und/oder Schreiben von Daten verwendet. Speicherzugriffe werden direkt auf die Zielsystemadresse gemappt. In diesem Modus ist ein Speicherzugriff mit einer Granularität von 1, 2, 4 oder 8 Byte möglich.
3. **Gemischter Modus**: Verwendung von L1 und L2 Cacheunterstützung für Private Memory Zugriffe und den sog. *Message Passing Memory Type (MPMT)* in Verbindung mit unterschiedlichen Konfigurationen des L1 Cache für Shared Memory Zugriffe: Daten, die in der Pagetable als MPMT markiert sind, werden bei einem Schreibzugriff zunächst in einem sog. *Write-Combine Buffer (WCB)* zwischengepuffert und erst an die Zieladresse geschrieben, wenn der WCB eine volle Cacheline (32 Byte) enthält. Der L2 Cache wird bei MPMT Zugriffen grundsätzlich umgangen. Die Benutzung von L2 Cache in Verbindung mit der in Software verwalteten Cachekohärenz ist nicht möglich. Zur Nutzung des MPMT existieren folgende Varianten:
    - **MPMT + L1CM**: MPMT mit L1 Cacheunterstützung. Um dies zu ermöglichen, wurde der Befehlssatz des Pentium um einen neuen Be-

fehl namens `CL1INVMB` erweitert. In Verbindung mit dem MPMT ist dieser Befehl die Grundlage für die in Software verwaltete Cachekohärenz, für den Fall, dass Teile des MPB oder externen Speichers als SHM genutzt werden. Wenn der `CL1INVMB` Befehl ausgeführt wird, werden alle als MPMT markierten L1 Cachezeilen innerhalb eines Taktes invalidiert (ungültig gemacht). Darauffolgende Lese- oder Schreibzugriffe auf MPMT Zeilen resultieren in einem garantierten Cache Miss. Bei einem Lesezugriff wird die jeweilige Zeile direkt aus dem SHM geholt. Bei einem Schreibzugriff werden Daten zunächst im WCB zwischengespeichert, bis eine Cachezeile voll ist und dann erst in den Shared Memory geschrieben.

- **MPMT + L1UCM**: MPMT ohne L1 Cacheunterstützung. In diesem Fall durchlaufen Uncached Schreibzugriffe zusätzlich den WCB. Diese können so beschleunigt werden, da sie im Gegensatz zu Lesezugriffen nicht mit einer Granularität von 1, 2, 4 oder 8 Byte erfolgen müssen, sondern mit bis zu 32 Byte Breite erfolgen können.

Anhand der flexiblen Speicherarchitektur können unterschiedliche Programmiermodelle experimentell untersucht werden. Die von Intel zur Verfügung gestellte *Rapidly Communicating Cores Environment (RCCE)* Softwarebibliothek [198, 15] implementiert bereits grundlegende Mechanismen zur Umsetzung von verteilten Anwendungen auf Basis von Shared Memory oder Message Passing. Aktuell nutzt RCCE den nicht cachekohärenten MPB für ein synchrones On-Chip Message Passing, kann aber auch als Ausgangspunkt für die Entwicklung neuer Protokolle verwendet werden. Alternative Lösungen zum synchronen Message Passing Protokoll von RCCE sind z.B. in [144] oder in Anhang A zu finden.

2.4.4. Performanzanalyse

Zur Bewertung und zum Vergleich paralleler Algorithmen existieren verschiedene Leistungsmaße und Metriken. Der Nutzen einzelner Leistungsmaße ist abhängig vom betrachteten Anwendungsfall und dem Ziel der Leistungsbewertung. Im Rahmen dieser Arbeit ist das Primärziel der Parallelverarbeitung, eine Steigerung der Performanz im Sinne einer höheren Ausführungsgeschwindigkeit bzw. einer geringeren Simulationslaufzeit im Vergleich zur sequentiellen Ausführung zu erreichen. Ein weiteres damit eng verknüpftes Ziel ist die Erreichung einer guten Skalierbarkeit auf einer bestimmten Zielplattform.

Im Allgemeinen gehören Laufzeit und Skalierbarkeit zu den wichtigsten Kriterien bei der Bewertung von parallelen Algorithmen und Systemen [111]. Die im Folgenden beschriebenen Leistungsmetriken zielen daher auf eine Bewertung

der Leistungsfähigkeit im Hinblick auf Laufzeit und Skalierbarkeit. Die Beschreibungen basieren auf [111][49][133].

2.4.4.1. Sequentielle Laufzeit

Im Kontext von Prozessoren und Mikroarchitekturen wird die Laufzeit oft in Abhängigkeit der notwendigen Taktzyklen für die Abarbeitung eines Algorithmus oder Programms definiert [133], d.h.

$$T_s = CC \times t_{cycle}. \tag{2.3}$$

Dabei ist T_s die Laufzeit, CC die Anzahl notwendiger Taktzyklen (engl. *Cycle Count*) zur Abarbeitung eines Programms und t_{cycle} die Taktzykluszeit. Ein alternativer Ansatz ist es, für die Bestimmung der Laufzeit das Zeitintervall, vom Beginn der Berechnung bis zu deren Beendigung zu messen. Diese Methode wurde in dieser Arbeit angewendet.

2.4.4.2. Parallele Laufzeit

Die Definition der Laufzeit über Gleichung 2.3 eignet sich für eine Performanzanalyse von einkernigen Prozessoren. Im parallelen Fall gibt es allerdings viele weitere zu berücksichtigende Parameter. Die *Laufzeit eines parallelen Programms* T_p auf einer bestimmten Zielplattform ist u.a. eine Funktion der Problemgröße N, der Anzahl der Prozessoren P und der Anzahl an Tasks U [111]:

$$T_p = f(N, P, U, ...) \tag{2.4}$$

Ein möglicher Ansatz zur näherungsweisen Bestimmung von T_p ist es, die Laufzeit als das Zeitintervall, vom Beginn der Berechnung eines parallelen Programms bis zu deren Beendigung zu definieren. Der Beginn der Berechnung ist dabei spezifiziert als der Zeitpunkt, an dem der erste Prozessor mit der Abarbeitung startet. Das Ende der Berechnung ist der Zeitpunkt, an dem der letzte Prozessor mit der Abarbeitung endet [111]. Diese Vorgehensweise wurde auch in dieser Arbeit zur Bestimmung von T_p verwendet.

2.4.4.3. Beschleunigung

Die *Beschleunigung* (engl. *Speedup*) bezeichnet den Faktor, um den die Laufzeit bei paralleler Ausführung gegenüber der sequentiellen Ausführung reduziert wird. Sie ist der Quotient der sequentiellen und parallelen Laufzeiten für ein gegebenes Problem [111, 49]:

$$S_p = \frac{T_s}{T_p} \tag{2.5}$$

Dabei ist T_s die Laufzeit der sequentiellen Implementierung auf einem einzigen Prozessor und T_p die Laufzeit der parallelen Implementierung auf P Prozessoren. Wenn $S_p = P$, dann spricht man von einer linearen Beschleunigung. Wenn $S_p > P$, so spricht man von einer superlinearen Beschleunigung. Letztere kommt in der Realität äußerst selten vor. In den meisten Fällen ist die Beschleunigung durch die Anzahl P der Prozessoren nach oben beschränkt, d.h. $Sp \leq P$.

Ein Grund für eine superlineare Beschleunigung können Cacheeffekte sein: Durch einen größeren Parallelisierungsgrad ist im Gesamtsystem mehr Cache verfügbar. Dadurch können mehr Daten im Cache anstelle des langsameren Hauptspeichers vorgehalten werden. Als Folge wird die parallele Laufzeit im Vergleich zur sequentiellen Laufzeit noch einmal zusätzlich reduziert.

2.4.4.4. Kosten, Overhead und Effizienz

Mit den *Kosten eines parallelen Programms* wird die von den Prozessoren bei der Problemlösung durchgeführte Arbeit gemessen [49]. Die Kosten können geschrieben werden als

$$C_p = T_p \times P \tag{2.6}$$

Dabei gilt ein paralleles Programm als kostenoptimal [49], wenn die einzelnen Prozesse der parallelen Implementierung insgesamt die gleiche Anzahl an Operationen ausführen wie die sequentielle Implementierung, d.h. wenn

$$C_p = T_s. \tag{2.7}$$

Der *Overhead* entspricht der Differenz zwischen den Kosten des parallelen und denen des sequentiellen Programms:

$$H_p = C_p - T_s = P \times T_p - T_s \tag{2.8}$$

Im kostenoptimalen Fall gilt $H_p = 0$. Die *Effizienz* ist ein Maß für die zusätzlich, durch die Parallelisierung erzeugte Last einzelner Prozessoren. Sie gibt die Verarbeitungsgeschwindigkeit im Vergleich zu einer sequentiellen Implementierung bezogen auf einen einzelnen Prozessor an. Sie sagt aus, wie nah ein bestimmter paralleler Algorithmus an das Optimum (maximal möglicher Speedup) herankommt. Die Effizienz kann aus der Beschleunigung durch Normierung auf die Anzahl P der Prozessoren wie folgt berechnet werden:

$$E_p = \frac{S_p}{P} = \frac{T_s}{P \times T_p} = \frac{T_s}{C_p} \tag{2.9}$$

Es können folgende drei Fälle unterschieden werden [49, 35]:

- $E_p < 1$: Die Parallelisierung ist *suboptimal* bzgl. ihrer Kosten. Dies ist in der Praxis der Normallfall. Es liegt eine Beschleunigung unterhalb einer linearen Beschleunigung vor.
- $E_p > 1$: Die Parallelisierung ist *kostenoptimal*. Es liegt eine lineare Beschleunigung im Vergleich zur sequentiellen Implementierung vor.
- $E_p > 1$: Die Parallelisierung ist *besser als kostenoptimal*. Es liegt eine *superlineare Beschleunigung* vor.

2.4.4.5. Amdahlsches Gesetz

Das sog. *Amdahlsche Gesetz* geht zurück auf eine Veröffentlichung von Gene M. Amdahl aus dem Jahre 1967 [36]. Es erfasst die oft gültige Tatsache, dass Algorithmen aus einem parallelisierbaren und einem nicht zu vernachlässigenden rein sequentiellen Anteil bestehen. Der sequentielle Anteil ist nach Amdahl für die Limitierung der Beschleunigung durch parallele Ausführung verantwortlich. Über den sequentiellen Anteil kann eine theoretische Obergrenze der erzielbaren Beschleunigung für einen bestimmten parallelen Algorithmus geschätzt werden. Das Gesetz kann geschrieben werden als:

$$S_p = \frac{T_s}{T_p} = \frac{T_s}{T_s \times (f_s + (\frac{1-f_s}{P}))} = \frac{1}{f_s + \frac{f_p}{P}}. \tag{2.10}$$

Dabei gilt:

- S_p ist die Beschleunigung,
- f_s ist der sequentielle Anteil des Algorithmus,
- f_p ist der parallelisierbare Anteil des Algorithmus,
- P ist die Anzahl der Prozessoren,
- $f_s + f_p = 1$ mit $0 \leq f_{s/p} \leq 1$.

Im Idealfall ist $S_p = P$ (mit $f_s = 0$ und $f_p = 1$). In der Realität ist $1 \leq S_p \leq P$. S_p ist dabei durch den sequentiellen Anteil f_s beschränkt, da dieser durch Parallelisierung nicht verringert werden kann (für den Grenzfall von $P \to \infty$ ist $S_p = \frac{1}{f_s}$).

Als Beispiel ist in Abb. 2.22 der Speedup in Abhängigkeit der Anzahl vorhandener Prozessorkerne aufgetragen. Wenn der sequentielle Anteil in einem Programm groß ist, dann tritt eine Sättigung des erreichbaren Speedups schon bei einer kleinen Anzahl von Kernen ein. Im Beispiel ist zu sehen, dass bei Ausführung einer Applikation auf 16 Kernen schon ein sequentieller Anteil von lediglich 6.7% in einem maximalen theoretischen Speedup von nur acht resultiert.

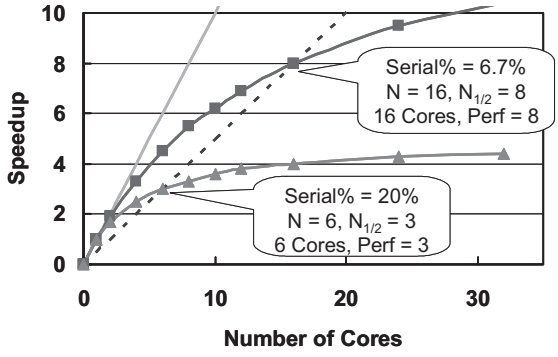


Abbildung 2.22.: Limitierung der Beschleunigung nach Amdahl (Quelle: [63])

3. Stand von Forschung und Technik

3.1. Parallele Simulation von Multiprozessoren

Da eingebettete Systeme immer öfter als MPSoC realisiert werden, nimmt die Größe und Komplexität entsprechender Simulationsmodelle immer weiter zu. Dies verursacht einen stetigen Anstieg der Simulationslaufzeiten. Durch Abstraktion können Simulationen zwar beschleunigt werden, die Abstraktion geht aber auf Kosten der Genauigkeit. Die parallele Simulation ist daher eine Möglichkeit, um die stetig steigende Diskrepanz zwischen Performanz und verfügbarer/notwendiger Genauigkeit zu verringern [180].

Im Folgenden wird zunächst ein allgemeiner Überblick über existierende Ansätze zur parallelen MPSoC Simulation gegeben und diese entsprechend ihrem Anwendungsbereich eingeordnet. Anschließend werden SystemC spezifische Forschungsansätze im Detail klassifiziert.

3.1.1. Anwendungsbereiche

In Abb. 2.14 aus Abschnitt 2.3.2.3 werden unterschiedliche Anwendungsfälle für die Simulation mit SystemC/TLM genannt: Softwareentwicklung, Performanzanalyse von Software, Architekturanalyse und Verifikation von Hardware. Die notwendige zeitliche Genauigkeit eines Simulators wird durch den Anwendungsfall bestimmt. Das Schema eignet sich daher gut, um einen Überblick über existierende (auch SystemC unabhängige) Werkzeuge zur parallelen MPSoC Simulation zu geben und diese initial zu klassifizieren.

Für die Softwareentwicklung steht weniger die zeitliche Genauigkeit als vielmehr die Möglichkeit im Vordergrund, die funktionale Korrektheit von Software zu verifizieren. Erste Performanzabschätzungen sollten dennoch möglich sein. Zu den bereits existierenden parallelen Simulatoren, die hierfür geeignet sind, gehören z.B. Parallel SimOS [169] oder Graphite [201]. Ein kommerzielles Beispiel ist Simics [104]. Neben abstrakter Modellierung von Hardwarekompo-

nenten und speziellen Techniken wie Dynamic Binary Translation [10] wird die Simulation typischerweise durch starke temporäre Entkopplung von parallelen Threads um mehrere hundert bis tausend Taktzyklen, zusätzlich beschleunigt. Dies hat eine enorme Reduktion des Synchronisationsaufwands gegenüber zyklengenauer Simulation zur Folge und macht es möglich, sogar komplette Betriebssysteme in akzeptabler Zeit auf einem MPSoC Modell zu booten.

Slacksim [81], Darsim/Hornet [183][182], gem5/GEMS [194, 54][1] oder SystemC spezifische Ansätze wie *Transaction Level Modeling with Distributed Time (TLM-DT)* [199] und die Arbeit von Yi et al. [268] sind Beispiele, welche besser für eine detailliertere Performanzanalyse geeignet sind. Hardwarekomponenten werden akkurater modelliert als in den zuvor genannten Beispielen, so dass bestimmte architekturspezifische Effekte wie z.B. Zugriffszeiten auf den Speicher genauer wiedergegeben werden können. Dies führt dazu, dass diese Simulatoren um mehrere Größenordnungen langsamer sind. Temporäre Entkopplung ist möglich, wird aber für eine detaillierte Performanzanalyse typischerweise auf wenige zehn oder hundert Taktzyklen beschränkt. Ein quantitative Analyse für den gem5 Simulator [54] ist in [73] zu finden.

Für den Anwendungsfall der Hardwareverifikation muss eine zyklenakkurate Simulation möglich sein, z.B. um Timingfehler und Hardwarebugs erkennen zu können. Arbeiten, welche dies explizit als Fokus haben, sind beispielsweise [215, 260, 105, 232, 206, 188, 181]. [215] basiert auf einem proprietären Simulator namens LSE. Die Arbeiten in [206, 188, 181] basieren auf VHDL bzw. Verilog und sind damit auf zyklenakkurate oder subzyklenakkurate Simulation (auf der Ebene von Gates) beschränkt. Die Ansätze in [215, 260, 105, 232] basieren auf SystemC.

Der Fokus der vorliegenden Arbeit ist die zyklenakkurate und -approximative parallele Simulation von MPSoC Modellen, die sich für Performanzanalysen und Hardwareverifikation eignet. Dazu wurde SystemC als Grundlage gewählt. Im Folgenden werden existierende Forschungsansätze zur Parallelisierung von SystemC weiter vertieft betrachtet.

3.1.2. Forschungsansätze basierend auf SystemC

Die aktuelle SystemC Referenzimplementierung der Accelera Systems Initiative [1] arbeitet rein sequentiell und implementiert ausschließlich ein kooperatives Mulitasking zur Simulation von paralleler Hardware- oder Software (vgl. Abschnitt 2.3.2.2). Daher ist die Untersuchung von Möglichkeiten zur Parallelisierung des SystemC Schedulers schon seit einiger Zeit in den Fokus der Forschung geraten. Existierende Ansätze lassen sich neben dem Modellierungsstil orthogo-

[1] Laut [118] existiert eine initiale parallele Implementierung von gem5.

nal hinsichtlich Zielplattformen, Synchronisationsverfahren und Kernelpartitionierung klassifizieren.

3.1.2.1. Modellierungsstile und Zielapplikationen

Die Unterstützung unterschiedlicher Modellierungsstile ist die Voraussetzung zur Modellbildung auf verschiedenen Abstraktionsebenen. Die Arbeiten in [93], [86], [90], [205] und [260] unterstützen nur Kommunikation basierend auf dem Evaluate/Update Paradigma. Damit ist die Anwendbarkeit auf RTL und ähnliche Modellierungstile beschränkt. Umgekehrt unterstützen die Arbeiten in [214] und [199] nur transaktionsbasierte Kommunikation und sind daher nur für die Simulation auf Transaktionsebene, nicht aber auf Registertransferebene geeignet. [105], [232] und [237] sind die einzigen unter den genannten Arbeiten, die sowohl die sowohl RTL Modelle, als auch TL Modelle unterstützen. Außer [205] und [237] fokussieren alle genannten Beispiele explizit auch auf die Simulation vollständiger SoCs. In [232] und [199] liegt der Schwerpunkt explizit auf der Simulation von MPSoCs.

3.1.2.2. Zielplattformen

Als Zielplattformen zur parallelen SystemC Simulation kommen in der Literatur insbesondere COWs, cachekohärente SHM Multiprozessoren oder Grafikprozessoren (engl. *Graphics Processing Units (GPUs)*) zum Einsatz (vgl. Abschnitt 2.4). [93], [86], [90] oder [214] sind Beispiele für Arbeiten, die auf die parallele Ausführung von SystemC Modellen auf einem Workstation Cluster abzielen. Sie sind grundsätzlich geeignet für Architekturen, die nur über einen verteilten und keinen gemeinsam nutzbaren Speicher verfügen. Dies impliziert typischerweise eine statische Modellpartitionierung, eine verteilte Ausführung innerhalb mehrerer Betriebssystemprozesse, sowie eine nachrichtenbasierte Kommunikation.

Die in [105], [199] und [232] beschriebenen Ansätze sind spezialisiert für die Ausführung auf cachekohärenten SHM Multiprozessoren. Dabei wird die Simulation nicht auf Betriebssystemprozesse, sondern auf Threads aufgeteilt, die einen gemeinsamen virtuellen Adressraum besitzen. Dies hat auf cachekohärenten SHM Maschinen den Vorteil eines sehr geringen Kommunikationsoverheads. Wegen der Abhängigkeit von einem gemeinsamen virtuellen Adressraum sind diese Verfahren normalerweise nicht auf Architekturen mit ausschließlich verteiltem Speicher anwendbar.

[205], [260]) oder [237] beschreiben sog. *General Purpose Computation on Graphics Processing Unit (GPGPU)* Ansätze zur Nutzung von GPUs für die SystemC Beschleunigung. Während die ersten beiden Arbeiten nur eine homogene GPU

Plattform betrachten, steht bei [237] eine heterogene Plattform bestehend aus cachekohärentem SHM Multiprozessor und GPU im Fokus. Einen aktuellen Überblick über verschiedene GPU Ansätze gibt [53].

3.1.2.3. Synchronisationsverfahren

Eine allgemeine Klassifikation von Synchronisationsalgorithmen zur PDES in konservativ/optimistisch, synchron/asynchron, oder zentral/dezentral wurde bereits in Abschnitt 2.2.3.4 gegeben. Existierende Ansätze zur SystemC Parallelisierung lassen darin wie folgt einordnen:

- **Konservativ versus optimistisch**: Alle bekannten Ansätze können in die Klasse der konservativen Strategien eingeordnet werden, da sie versuchen, kausale Abhängigkeiten zwischen Ereignissen, soweit sie durch die Modellspezifikation definiert sind, soweit wie möglich nicht zu verletzen.
- **Synchron versus asynchron**: [105],[232] und [237] sind Beispiele für vollständig synchrone Verfahren: Die Simulation startet und stoppt phasenweise an globalen Barrieren. In allen drei Fällen erfolgt die Synchronisation bis zu einer Granularität von Deltazyklen, was einen hohen Overhead zur Folge hat.

  [93, 86, 90, 199] können als asynchrone Verfahren klassifiziert werden. [93], [86] und [90] nutzen globale Synchronisation zur Auflösung von Deadlocks. In [199] wird auf globale Synchronisation auf Kosten einer Limitierung der verwendbaren Modellierungsstile verzichtet.

  [214] kann als eine Mischung aus einem synchronen und einem asynchronen Ansatz betrachtet werden: Logische Prozesse synchronisieren nur lokal mit ihren jeweiligen Nachbarn. In dieser Hinsicht ist das Verfahren asynchron. Diese Synchronisation efolgt allerdings durch explizite Handshakes in global definierten, regelmäßigen Abständen und abwechselnd mit den Ausführungsphasen, was typisch für synchrone Verfahren ist.
- **Zentral versus dezentral**: [105], [232], [205], [86], [260] und [237] basieren auf einem zentralisierten Synchronisationsverfahren. In allen Fällen existiert ein Master, der die Synchronisation zentral steuert. Auch die GPU basierten Ansätze in [205], [260]) und [237] basieren auf einem zentralisierten Verfahren. Diese ergibt sich zwangsläufig daraus, dass eine GPU innerhalb eines PCs eine I/O Ressource ist, die phasenweise vom Host mit neuen Daten (zu evaluierenden SystemC Prozessen) versorgt werden muss, um diese in SIMD Manier (vgl. Abschnitt 2.4.2) zu verarbeiten. In [237] wird ein zentralisiertes Synchronisationsverfahren auf einem Multiprozessor mit einem zentralisierten Verfahren für eine GPU kombiniert. Die Synchronisati-

on zwischen beiden erfolgt mit einer Granularität von bis zu einem Deltazyklus.

Die Ansätze in [93], [86], [90], [214] und [199] können als dezentral klassifiziert werden. In [214] und [199] synchronisieren logische Prozesse ausschließlich mit benachbarten Prozessen. In [90] wird ein zentralisiertes Verfahren durch globale Synchronisation vermieden.

3.1.2.4. Kernelpartitionierung

Unter dem Begriff der Kernelpartitionierung ist zu verstehen, in welcher Art und Weise die Phasen des sequentiellen SystemC Schedulings (vgl. Abschnitt 2.3.2.2) auf parallele Prozesse verteilt werden. Dabei kann man zwischen einer *asymmetrischen* und einer *symmetrischen* Partitionierung unterscheiden: In einem asymmetrischen Kernel implementieren parallele Prozesse jeweils unterschiedliche Phasen des sequentiellen Schedulings. U.U. werden nicht alle Phasen repliziert. In einem symmetrischen Kernel implementieren alle Prozesse alle Phasen. Dabei werden alle Kernelphasen repliziert (siehe auch Abschnitt 4.2.4).

[105, 232, 205, 260, 237] sind allesamt Beispiele für asymmetrische Kernelpartitionierung. Der zugrundeliegende Mechanismus wird auch als *Master/Worker Schema* bezeichnet (vgl. [49]). Der Master führt alle Phasen des SystemC Schedulings aus, die Worker nur die Evaluation Phase. Dabei werden zu evaluierende SystemC Prozessen vom Master auf die Worker verteilt. Der Vorteil des Verfahrens ist die globale Sicht des Masters auf die Simulation und die Möglichkeit zu einfachen Umsetzung einer zentral gesteuerten dynamischen Lastverteilung.

Das Master/Worker Schema eignet sich besonders zur Nutzung auf cacheköhärenten SHM Multiprozessoren, da hier eine dynamische Lastverteilung ohne die Notwendigkeit einer aufwändigen Umverteilung von Daten erfolgen kann. In allen Fällen ist entweder ein Thread oder Prozess auf einem Multiprozessor der Master. In [105] und [232] sind Worker durch andere Threads gegeben. In [205] und [260] können alle Threads auf der GPU als Worker betrachtet werden. In [237] existieren sowohl Workerthreads auf dem Multiprozessor als auch auf der GPU.

In [93], [86], [90] und [214] wird eine symmetrische Kernelpartitionierung verwendet. D.h. alle vorhandenen parallelen Prozesse führen jeweils alle Kernelphasen aus. Ein ganz anderer Ansatz wird in [199] verfolgt: Hier wird zugunsten der Performanz auf die Implementierung des Evaluate/Update Paradigmas und damit auf die Implementierung der Update Phase vollständig verzichtet.

3.1.3. SystemC Front-Ends

Die bisher im Kontext der parallelen SystemC Simulation genannten Arbeiten bieten eine unterschiedlich umfangreiche Werkzeugunterstützung, um Simulationsmodelle und -kernel für eine parallele Simulation zu präparieren. In [93], [86], [90], [214] oder [199] existiert keinerlei Werkzeugunterstützung zur automatisierten Präparation von Modell und Kernel. Die Partitionierung muss manuell erfolgen. Außer in [199] ist sie beschränkt auf grobgranulare Module der obersten Hierarchieebene eines Modells. [105] und [232] sind aufgrund des Master/Worker Schemas und der Beschränkung auf cachekohärente SHM Multiprozessoren prinzipiell nicht auf eine statische Modellpartitionierung angewiesen. Im Kontext von SystemC existieren damit aktuell nur in Verbindung mit GPUs Arbeiten, bei denen verschiedene automatisierte Optimierungsschritte vor der eigentlichen Simulationslaufzeit vorgenommen werden (z.B. [205] und [260]). Dies kann Schritte wie eine automatisierte Modellpartitionierung und -transformation oder Kernelkonfiguration beinhalten.

Entsprechende Werkzeugketten basieren auf einer automatischen Extraktion einer formalen abstrakten Repräsentation (AR) oder vollständigen Zwischenrepräsentation (engl. *Intermediate Representation (IR)*) anhand eines sog. SystemC Front-Ends [192]. Eine solche Modellrepräsentation enthält alle für eine Präparation notwendigen struktur- und verhaltensorientierten Informationen. Das Front-End bietet Funktionen, die das Parsen und das Generieren einer IR unterstützen. Zu den bekanntesten freien SystemC Front-Ends gehören Pinapa [204], PinaVM [193], Scoot [57], SystemCXML [52] oder KaSCPar [7]. HIFSuite [59] ist ein Beispiel für ein kommerzielles Softwarepaket. Mögliche allgemeine Anwendungsfälle für SystemC Front-Ends sind formale Verifikation, Synthese, optimierte Kompilierung und Simulation, Debugging oder Visualisierung [193].

SystemC Front-Ends können in rein statische und hybride Ansätze klassifiziert werden. Während erstere die IR ausschließlich durch statische Codeanalyse extrahieren, nutzen letztere auch die Möglichkeit, SystemC Code mit dem Kernel auszuführen und auf diese Art zusätzliche Informationen zu gewinnen. Beispielsweise ist es schwer bis gar nicht möglich, dynamisch instanziierte Architekturen vollständig statisch zu erkennen, da entsprechende Informationen nur zur Laufzeit zur Verfügung stehen [192]. Zu den rein statischen Front-Ends gehören Scoot [57], SystemCXML [52] und KaSCPar [7], zu den Front-Ends, die zusätzlich dynamische Ausführung nutzen, gehören Pinapa [204] und PinaVM [193].

Die Methoden zum Parsen von SystemC Code unterscheiden sich stark. Die Spannweite reicht vom Parsen mit Hilfe von Doxygen [52][2] bis hin zur Nutzung von C/C++ Compiler-Suites wie GCC [57, 204, 5] oder *Low Level Virtual Machine (LLVM)* [186, 193]. PinaVM [193] basiert z.B. auf LLVM und nutzt die LLVM

IR sowohl zur statischen Extraktion von Verhaltensbeschreibungen, als auch zur dynamischen Extraktion der Modellstruktur anhand von *Just-in-Time Compilation (JITC)*. Ein erst kürzlich veröffentlichtes rein statisches SystemC Front-End ist SystemC-clang [158], welches als Plug-in für das LLVM Front-End clang [88] implementiert ist. Keines der genannten SystemC Front-Ends nutzt die Analyse zur Extraktion von Informationen zur Performanzabschätzung.

3.1.4. Forschungsansätze basierend auf SpecC

Auch im Bereich von SpecC existieren einige Forschungsansatze zur parallelen Simulation, aufgrund der geringen Verbreitung von SpecC allerdings nicht im gleichen Umfang wie im Fall von SystemC. Zu erwähnen wären hier die Arbeiten von Chen et al. in [84, 85] sowie Chen und Dömer in [82] und [83]. Der Ansatz in [84] kann als weitgehend äquivalent zu dem SystemC-basierten voll synchronen Master/Worker Ansatz aus [232] eingeordnet werden, welcher speziell für cachekohärente SHM Architekturen entwickelt wurde. In beiden Fällen werden TLM und RTL Modelle unterstützt

Die Methode in [82] dient zur parallelen Simulation von TL Modellen und kann ähnlich wie der SystemC/TLM Simulator in [214] klassifiziert werden. Im Unterschied zu [214] ist der Synchronisationsmechanismus in [84] vollständig asynchron.

Die Ansätze in [85] und [83] sind als Erweiterungen des zentralisierten Ansatzes aus [84] zu verstehen. Auf Basis einer statischen compilerbasierten Modellanalyse wird der parallele SpecC Kernel so konfiguriert, dass Prozesse ohne Kausalitätsverletzungen vom Master bis zu einem gewissen Grad außerhalb der durch die Zeitstempel definierten Reihenfolge auf die Worker verteilt werden können. Das synchrone Verfahren aus [84] wird damit zu einem asynchronen Verfahren erweitert.

3.2. Interdisziplinäre Co-Simulation

In verschiedenen Anwendungsdisziplinen wie Informatik, Elektrotechnik oder Mechatronik haben sich unterschiedliche Modellierungsformalismen etabliert, die sich durch unterschiedliche Berechnungsmodelle charakterisieren lassen. Um Wechselwirkungen zwischen Anwendungsdisziplinen zu untersuchen, wird typischerweise Co-Simulation eingesetzt. Die Heterogenität der Berechnungsmodelle resultiert in Heterogenität in der Co-Simulation (vgl. Abschnitt 2.2.4). Das Management dieser Heterogenität ist eine Herausforderung, die aktuell von vielen Entwurfsmethodiken und Sprachen noch nicht hinreichend unterstützt wird

[176]. Nachfolgend wird zunächst ein allgemeiner Überblick über existierende Forschungsarbeiten zur heterogenen Co-Simulation gegeben. Anschließend werden einschlägige Interoperabilitätsstandards genauer betrachtet.

3.2.1. Allgemeiner Überblick

Existierende Ansätze zur heterogenen Simulation können allgemein in *formale* und *kopplungsbasierte* Ansätze eingeteilt werden. Formale heterogene Frameworks aus dem Bereich der eingebetteten Systeme wurden bereits in Abschnitt 2.3.3 erwähnt. Zu diesen gehören z.B. Ptolemy II [102], Metro II [95] oder ForSyDe [226]. Diese Werkzeuge basieren auf einer Sprache, welche es erlaubt, Wechselwirkungen zwischen beliebigen heterogenen Berechnungsmodellen mathematisch eindeutig zu beschreiben.

Neben diesen vollständig heterogenen Frameworks existieren verschiedene Ansätze, welche auf einer festen Kombination einiger weniger Berechnungsmodelle basieren. Dazu gehören Formalismen wie Hybrid Automata [134] oder Giotto [135]. Beispiele aus dem Bereich der Simulationswerkzeuge sind VHDL-AMS [18] und SystemC-AMS [1] oder Simulink/Stateflow [8]. Alle drei kombinieren diskrete mit kontinuierlichen Berechungsmodellen. Eine solche Simulation wird auch als hybrid bezeichnet [179].

Alternativ existieren in der Literatur Arbeiten, welche Simulatorkopplungen zur Untersuchung von Wechselwirkungen zwischen Anwendungsdisziplinen nutzen (z.B. [64, 242, 131, 200]). Dabei werden disziplinspezifische Simulatoren oder Modelle über eine zusätzliche Schnittstelle zur Co-Simulation miteinander gekoppelt. Im Unterschied zu den formalen Methoden ermöglicht dies die einfache Wiederverwendung bereits existierender disziplinspezifischer Modelle.

Technische Lösungen zur Kopplung sind vielfältig und reichen von Ansätzen auf Basis von TCP/IP [265] bis hin zur Integration per Shared Memory [164]. Unabhängig von der technischen Umsetzung sind bei kopplungsbasierten Ansätzen zwei Architekturtrends zu beobachten: Im ersten Fall dienen formale Frameworks wie Ptolemy II [102] oder ForSyDe [226] als zentraler Koordinator für eine Co-Simulation. Zu diesen Arbeiten gehören [184], [39] oder [40]. Simulatoren kommunizieren über das formale Framework, welches heterogene Simulatoren miteinander kombiniert und die kooperative Ausführung steuert. In zweiten Fall bildet eine Middleware die Grundlage (z.B. [65, 242, 170]). Simulatoren müssen dann eine Schnittstelle implementieren und dafür sorgen, dass das Berechnungsmodell der Simulationswerkzeugs in das Berechnungsmodell der Middleware übersetzt wird und umgekehrt. Für diesen Fall existieren bereits Standards wie die High Level Architecture (HLA) [22] oder das Functional Mock-up Interface (FMI) [29].

Alle genannten technischen und architekturspezifischen Lösungen sind Ansätze zur Herstellung besserer Interoperabilität. Im Folgenden wird der Begriff der Interoperabilität daher genauer erläutert.

3.2.2. Interoperabilität

Tolk und Muguira definieren in [251] und [250] den Begriff der Interoperabilität anhand des sog. *Levels of Conceptual Interoperability Model (LCIM)*. Das LCIM wird stetig weiterentwickelt und setzt sich aktuell aus sechs Ebenen zusammen [262] (siehe Abb. 3.1). Die Ebenen des LCIM entsprechen unterschiedlichen Abstraktionsschichten, welche es ermöglichen, den Begriff der Interoperabilität differenzierter zu beschreiben und das Problem der Herstellung von Interoperabilität in mehrere Teilprobleme zu zerlegen. Auf den untersten beiden Ebenen liegt der Fokus auf der technischen Infrastruktur zum Datenaustausch, auf den Ebenen 2 bis 4 auf der Implementierung des Simulationssystems und auf den Ebenen 5 und 6 auf der Modellierung.

Entsprechend Abb. 3.1 lässt sich Interoperabilität aus Sicht der Implementierung in *syntaktische Interoperabilität* (es gibt ein gemeinsames Verständnis über die Struktur von Daten), *semantische Interoperabilität* (es gibt ein gemeinsames Verständnis über die Bedeutung von Daten) und *pragmatische Interoperabilität* (es gibt ein gemeinsames Verständnis über den Kontext, in dem Daten verwendet werden) aufteilen. Aufgrund ihrer Relevanz in dieser Arbeit, wird im Folgenden näher auf die syntaktische und die semantische Interoperabilität eingegangen, wobei insbesondere zwischen statischer und dynamischer Semantik unterschieden wird (vgl. Abschnitt 2.2.2).

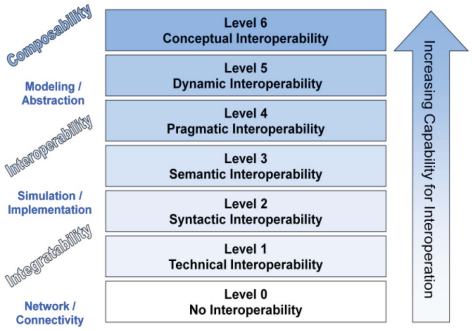


Abbildung 3.1.: Levels of Conceptual Interoperability Model (Quelle: [262])

Während für die Herstellung syntaktischer sowie statischer semantischer Interoperabiliät in der Literatur bereits diverse Lösungsansätze existieren (z.B. [154,

131, 242]), wird die dynamische semantische Interoperabilität zwischen Berechnungsmodellen oft nicht im notwendigen Maße berücksichtigt [176, 252]. Diese Tatsache ist verantwortlich für eines der Kernprobleme kopplungsbasierter Simulationsansätze: Nicht dokumentierte oder unklare Semantik, insbesondere die dynamische Semantik von gekoppelten Simulationswerkzeugen, hat amorphe Heterogenität zur Folge (vgl. Abschnitt 2.2.4), was die Kopplung enorm erschweren kann.

Vergleicht man standardisierte Middlewarelösungen wie die *High Level Architecture (HLA)* [22] oder das *Functional Mock-up Interface (FMI)* [29] mit einem formalen Framework wie PtII [102], so bieten beide eine ausgereifte statische semantische Interoperabilität. Eine Middleware bietet typischerweise die Möglichkeit zur verteilten Ausführung. Ist sie standardisiert, so bietet dies zudem den Vorteil einer größeren Verbreitung sowie u.U. bereits vordefinierte Dienste, die nicht von Grund auf neu implementiert werden müssen. Die Kopplung über ein formales Framework hingegen bietet im Allgemeinen den Vorteil einer besseren dynamischen semantischen Interoperabilität, da die Heterogenität durch das formale Framework koordiniert wird. Im Folgenden sollen die zwei aktuell am weitesten Verbreiteten Standards zur Herstellung von Interoperabilität zwischen Simulationswerkzeugen näher betrachtet werden.

3.2.3. Standardisierung

3.2.3.1. Functional Mock-up Interface

Das Functional Mock-up Interface ist ein werkzeugunabhängiger Standard [29], welcher Schnittstellen zum Austausch und zur Co-Simulation von dynamischen Modellen definiert. Das FMI ist dabei auf Modelle spezialisiert, die durch kontinuierliche und diskrete Differentialgleichungen beschrieben werden können. Abb. 3.2 skizziert das dem FMI zugrundeliegende Konzept anhand eines Beispiels aus der Automobilindustrie.

Die Entwicklung von FMI wurde von der Daimler AG mit dem Ziel initiiert, den Austausch von Simulationsmodellen zwischen Zulieferern und Erstausrüstern (*Original Equipment Manufacturers (OEMs)*) zu verbessern. Das FMI wird stetig weiterentwickelt. Die aktuelle Version ist FMI 2.0 vom Juli 2014 [29]. Das FMI Konsortium besteht aktuell aus 16 Partnern aus Industrie und Forschung. Der FMI Standard ist in drei Teile untergliedert:

1. **Gemeinsame Konzepte**: Hier werden allgemeingültige Prinzipien des FMI spezifiziert. Das grundlegende Konzept ist die sog. *Functional Mock-up Unit (FMU)*. Eine FMU ist äquivalent zu einem austauschbaren Modell oder einem Wrapper für einen sog. Slave zur Co-Simulation. Sie besteht aus einem

dynamischen Modell in Form von C Code und einer XML Datei, die Metainformation der FMU wie Definitionen von Variablen, Eingängen, Ausgängen oder Zuständen beinhaltet.

2. **FMI für den Modellaustausch**: In diesem Teil des Standards werden die notwendigen Prinzipien zum Austausch von FMUs zwischen Simulationswerkzeugen definiert. Dies beinhaltet die Definition von Methoden in Form von C Code sowie die Definition gültiger Interaktionen mit der FMU in Form einer Zustandsmaschine. Dabei wird vorausgesetzt, dass Werkzeuge C Code einer dynamischen Modellbeschreibung generieren können. Die FMU kann dann von anderen Werkzeugen integriert werden. Je nach Größe können Modelle entweder in einem Simulator oder auf einer eingebetteten Plattform (z.B. Microcontroller) ausgeführt werden.
3. **FMI für die Co-Simulation**: In diesem Teil werden die Schnittstellen einer FMU spezifiziert, die für eine Kopplung verschiedener Simulationswerkzeuge notwendig sind. Die FMU dient als Wrapper für einen Slave und verbindet diesen mit einem Master. Die Spezifikation beinhaltet die Definition von Methoden in Form von C Code sowie die Definition gültiger Interaktionen mit der FMU in Form einer Zustandsmaschine. Datenaustausch und Synchronisation zwischen Slaves werden durch den Master gesteuert. Der Datenaustausch erfolgt an diskreten Punkten. Dazwischen werden gekoppelte Simulatoren unabhängig voneinander ausgeführt. Der exakte Algorithmus des Masters ist nicht Teil der FMI Spezifikation.

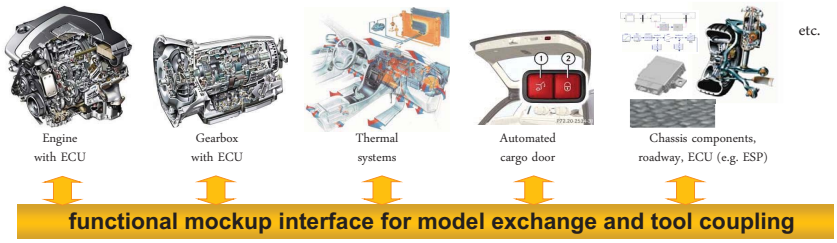


Abbildung 3.2.: Konzept des Functional Mock-up Interface (FMI) (Quelle: [29])

Insgesamt spezifiziert der FMI Standard damit eine für einen Modellaustausch oder eine Co-Simulation von zeitkontinuierlichen oder zeitdiskreten Systemen geeignete Semantik der Schnittstelle einer FMU. Dadurch, dass der Master nicht spezifiziert ist, existieren viele Freiheitsgrade für die Umsetzung verschiedener Algorithmen zur Co-Simulation. Da der Standard noch sehr jung ist und sich aktuell noch in der Entwicklung befindet, werden diskrete ereignisbasierte Modelle laut [42] nur beschränkt unterstützt. Bis auf eine Absichtserklärung schließt der Standard bisher keine Lösungsansätze für eine verteilte Ausführung mit ein.

3.2.3.2. High Level Architecture

Die HLA ist eine generische Softwarearchitektur für verteilte Simulation. Sie wurde zunächst durch das *Defense Modeling and Simulation Office (DMSO)* für das U.S. *Department of Defense (DoD)* spezifiziert [257] und ist seit dem Jahr 2000 ein IEEE Standard [22]. Die HLA ist keine Implementierung einer solchen Architektur. Das Anwendungsfeld ist ursprünglich der Bereich militärischer Trainingssimulationen. In HLA Terminologie entspricht eine verteilte Simulation einer *Federation* [23]. Eine Federation ist ein Zusammenschluss von *Federates*. Prinzipiell entspricht ein Federate einem Simulator. Federates sind über eine sog. *Runtime Infrastructure (RTI)* miteinander gekoppelt. Die Schnittstellen zur RTI werden durch sog. *Ambassador* bereitgestellt (vgl. Abb. 3.3). Der HLA Standard besteht aus vier Dokumenten. Diese definieren

- Regeln, welche die Verantwortlichkeiten von HLA Federates und Federations zur Sicherstellung einer konsistenten Implementierung definieren [22].
- Dienste und Schnittstellen zur RTI. Diese Dienste werden von interagierenden Simulationen genutzt, um einen koordinierten Austausch von Informationen in einer verteilten Federation zu ermöglichen [23].
- ein *Object Model Template (OMT)*. Dies beinhaltet Format und Syntax (aber nicht Inhalt, d.h. statische Semantik) eines HLA *Object Models (OM)* [24]. Das OM spezifiziert die Eigenschaften der Daten, die in einer Federation ausgetauscht werden können.
- Empfohlene Prozesse und Vorgehensweisen, die von Anwendern der HLA beim Entwurf und der Ausführung von Federations befolgt werden sollten [26].

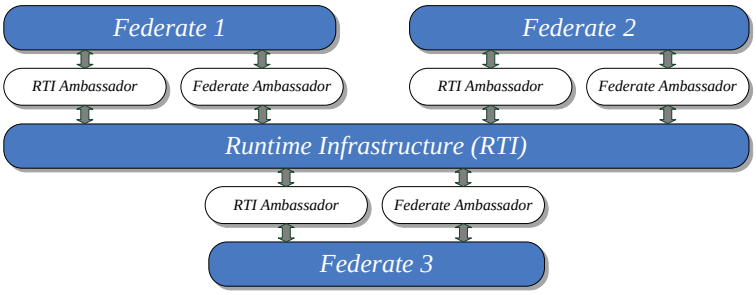


Abbildung 3.3.: High Level Architecture (HLA)

Die durch die Ambassador zu implementierenden HLA Schnittstellen sind in [23] in sieben Kategorien unterteilt: *Federation Management (FM)*, *Declaration Management (DM)*, *Object Management (OM)*, *Ownership Management (OWM)*, *Time*

Management (TM), *Data Distribution Management (DDM)* und Support Services. Ein detaillierteter Überblick ist in Anhand B zu finden.

Ein konkretes Object Model, welches auf dem OMT basiert, definiert die in einer Federation existierenden Typen in Form von sog. Objektklassen (*Object Classes*) und Interaktionsklassen (*Interaction Classes*) sowie deren Attributen (*Attributes*), Parametern (*Parameters*) und Datentypen (*Datatypes*). Zu Beginn oder während einer Simulation werden dann Instanzen von Object Classes (*Object Instances*) und Interaction Classes (*Interaction Instances*) erzeugt. Im Allgemeinen wird zwischen einem *Federation Object Model (FOM)* und einem *Simulation Object Model (SOM)* unterschieden. Während das FOM alle Eigenschaften einer gesamten Federation beschreibt, beinhaltet das SOM nur die Beschreibung von Eigenschaften eines einzelnen Federates.

Ein Teil der Informationen, die das FOM einer Federation beinhaltet, muss mit der sog. *Federation Object Model Document Data (FDD)* Datei (im ursprünglichen DoD Standard [257] heißt diese Datei *Federation Execution Data (FED)*) für eine Ausführung explizit spezifiziert werden. Dies beinhaltet u.a. Namen und Struktur von Object Classes, Interaction Classes, Attributes und Parameters sowie Publish und Subscribe Beziehungen auf den Klassen und deren Attributen/Parametern. Die FDD Datei beinhaltet bestenfalls das komplette FOM, aber nicht notwendigerweise.

Durch die Möglichkeit das FOM einer Federation anhand des OMT vollständig zu spezifizieren [24], wird die Spezifikation der Art und Weise der Repräsentation von Daten (statische Semantik) explizit zur Aufgabe des Erstellers der Federation. Die Spezifikation der dynamischen Semantik der Kommunikation ist hingegen nicht Teil des FOM und damit nicht explizit Aufgabe des Erstellers. Aufgrund der umfangreichen Schnittstellendefinition der HLA existiert daher im Allgemeinen eine Menge an Freiheitsgraden für die Implementierung der dynamischen Semantik des Datenaustauschs.

Eine Hilfestellung für die Entwicklung von HLA Federations bietet der *Distributed Simulation Engineering and Execution Process (DSEEP)* [26]. Der DSEEP erstreckt sich über sieben Schritte von der Vorgehensweise bei der Ziel- und Konzeptanalyse einer Simulation bis hin zur Vorgehensweise bei der Evaluation von Simulationsergebnissen (siehe Abb. 3.4). Der DSEEP ist in erster Linie als ein Leitfaden zu verstehen, der auf einen speziellen Anwendungsfall angepasst werden kann.

Im DSEEP [26] wird eine Spezifikation, welche die zur Laufzeit ausgetauschten Daten definiert um ein bestimmtes Simulationsziel zu erreichen, als *Simulation Data Exchange Model (SDEM)* bezeichnet. Dies beinhaltet laut [26] Klassenbeziehungen, Datenstrukturen, Parameter und andere relevante Information. Es ist nicht exakt spezifiziert, wie ein solches SDEM auszusehen hat. Im Zusammenhang mit dem SDEM werden sog. *Base Object Model (BOM)* Spezifikationen

[125, 126] als mögliche Elemente eines SDEM erwähnt. Ein BOM fokussiert auf die abstrakten konzeptorientierten Ebenen fünf und sechs des LCIM (vgl. [262]) und nicht auf die implementierungsorientierte Ebene der semantischen Interoperabilität, welche Fokus dieser Arbeit ist.

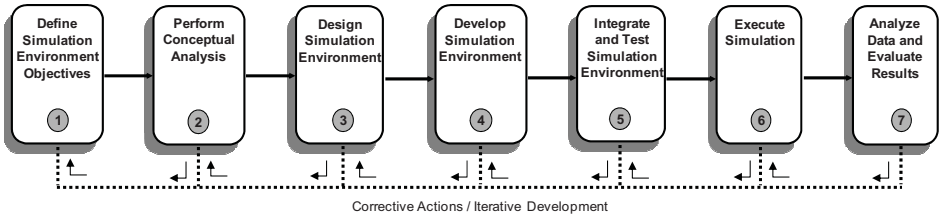


Abbildung 3.4.: Distributed Simulation Engineering and Execution Process (Quelle: [26])

4. Parallele SystemC Simulation für Multiprozessoren

Der Paradigmenwechsel von einer immer weiteren Erhöhung der Taktfrequenz in Richtung Multiprozessoren macht eine Parallelisierung existierender Simulationswerkzeuge unausweichlich, um das stetig größer werdende System Design Gap zu verringern (vgl. Abschnitt 2.1). Bei einer parallelen Simulation ist die Beschleunigung zum einen durch den Grad an Parallelität limitiert, der aus dem Simulator und dem ausgeführten Simulationsmodell extrahiert werden kann. Zum anderen hat die Ausführungsplattform einen großen Einfluss auf den erzielbaren Gewinn. Eine zusätzliche Anhebung der Abstraktionsebene kann u.U. zu einer starken Ausführungsbeschleunigung beitragen, ist aber meist mit einem Verlust an Genauigkeit verbunden.

Innerhalb dieses Spannungsfeldes sollen im folgenden Kapitel Möglichkeiten zur parallelen SystemC-basierten Simulation von eingebetteten MPSoCs untersucht werden. Der Schwerpunkt liegt dabei auf Modellen, deren zeitliche Genauigkeit bis auf Zyklenebene skaliert werden kann und die für Anwendungsfälle wie Hardwareverifikation und Debugging geeignet sind. Als Ausführungsplattform dient in erster Linie der Single-chip Cloud Computer, dessen Architektur als Blaupause für zukünftige Manycore Chips angesehen werden kann.

4.1. Allgemeine Anforderungen

Aus dem angestrebten Anwendungsfall und den beschriebenen Zielen ergeben sich folgende allgemeine Anforderungen für den Entwurf des parallelen Simulationsverfahrens:

- **Geschwindigkeit und Skalierbarkeit**: Die Parallelisierungsstrategie soll die Laufzeit von MPSoC Simulationen verkürzen und damit zu höherer Produktivität beitragen. Das Verfahren soll skalierbar sein und die Ausführung kleinerer und größerer MPSoC Modelle beschleunigen.
- **Kompatibilität zu relevanten Modellierungsstilen**: Für eine zyklengenaue Simulation im Kontext der Hardwareverifikation muss mindestens das RTL

Subset von SystemC unterstützt werden. Um auch die Performanzvorteile abstrakterer Simulationen nutzen zu können, sollte die Möglichkeit bestehen, zyklenapproximativ zu modellieren.

- **Erweiterbarkeit**: Der zu entwickelnde Simulator sollte anpassbar und erweiterbar an neue und zukünftige Anforderungen sein. Diese kann z.B. neue Modellierungsstile, Synchronisationsverfahren, oder Zielplattformen betreffen.
- **Kompatibilität zum SCC und Portierbarkeit**: Das Verfahren soll sich für die Ausführung auf dem SCC eignen. Die Portierbarkeit auf Multicore und zukünftige Manycore CPUs soll möglich sein.
- **Handhabbarkeit**: Die zur Parallelisierung notwendigen Schritte sollen nach Möglichkeit vor dem Anwender verborgen bleiben und automatisiert erfolgen.

4.2. Konzept und Methodik

4.2.1. Perspektiven und Abstraktion

In Analogie zur plattformbasierten Entwurfsmethodik auf Systemebene aus Abschnitt 2.1.1 wird das zu entwickelnde parallele Simulationssystem im Folgenden auf der Systemebene betrachtet. Um auf dieser Ebene unterschiedliche Aspekte des Gesamtsystems genauer beschreiben zu können, ist eine weitere Unterteilung in Zwischenebenen sinnvoll (vgl. [115]). Im Allgemeinen ist dann eine Betrachtung des Simulationssystems aus einer horizontalen und einer vertikalen Perspektive möglich. Die Betrachtung aus einer horizontalen Perspektive entspricht der Betrachtung auf einer bestimmten Abstraktionsebene. Die Betrachtung aus einer vertikalen Perspektive entspricht der Fokussierung auf Beziehungen zwischen Abstraktionsebenen.

Will man in einer Software eine Trennung von Abstraktionsebenen aus Gründen der Erweiterbarkeit und der Komplexitätsreduktion dauerhaft beibehalten, so ist die Verwendung einer Schichtenarchitektur eine mögliche Lösung. Dabei wird die Software in vertikal angeordnete Schichten aufgeteilt, von denen jede eine begrenzte Anzahl an Funktionen erfüllt. Existieren eindeutig definierte Schnittstellen, so können unterschiedliche Implementierungen leicht ausgetauscht werden.

Der theoretische Zusammenhang zwischen Abstraktion und Schichtenbildung im Kontext von Kommunikationssystemen wurde von Herzberg und Broy in [137] und [138] erläutert. Demnach besteht jede Schicht aus Komponenten und

sog. komplexen Konnektoren, die die Komponenten verbinden[1]. Die Verfeinerung einer Schicht entspricht der Verfeinerung von komplexen Konnektoren dieser Schicht mit Komponenten und Konnektoren der darunterliegenden Schicht. Das wohl bekannteste Beispiel einer Schichtenarchitektur aus dem Bereich der Kommunikationssysteme ist das *ISO/OSI Referenzmodell* (OSI-RM) [270, 147]. Wie bereits in Abschnitt 2.1 dargelegt, ist die Schichtenbildung auch ein fundamentales Prinzip beim plattformbasierten Entwurf.

Im Rahmen dieser Arbeit soll das Prinzip der Schichtenbildung einerseits als Hilfsmittel für die Erläuterung der Prinzipien des entwickelten parallelen Simulators eingesetzt werden. Andererseits diente die Schichtenbildung als Fundament für eine strukturierte Umsetzung im Hinblick auf die oben genannten Anforderungen.

4.2.2. Architekturmodell für eine parallele Simulation

Für die Beschreibung der Verfahren in den weiteren Abschnitten sowie als Basis für die Implementierung, dient das folgende Schichtenmodell als Grundlage:

Definition 4.1 *(**Parallele Simulation**): Eine parallele Simulation ist ein Tupel* $\mathcal{S} = (S^i)_{i\in 4,\ldots,0} = (\mathcal{M}, \mathcal{K}, \mathcal{L}, \mathcal{B}, \mathcal{A})$. *Dabei gilt:*

- $\mathcal{M}$ *ist die Modellebene,*
- $\mathcal{K}$ *ist die Kernelebene,*
- $\mathcal{L}$ *ist die logische Ebene,*
- $\mathcal{B}$ *ist die Basisdienstebene,*
- $\mathcal{A}$ *ist die Ausführungsplattformebene.*

Die Ebenen $\mathcal{K}$, $\mathcal{L}$ *und* $\mathcal{B}$ *repräsentieren den parallelen Simulator PS:* $PS = \{\mathcal{K}, \mathcal{L}, \mathcal{B}\}$.

Definition 4.1 wird im Folgenden auch als Referenzmodell für die parallele SystemC Simulation bezeichnet. Im Unterschied zum LCIM aus Abschnitt 3.2.2 fokussiert das Referenzmodell aus Definition 4.1 ausschließlich auf Implementierungsaspekte. Die Ebenen haben folgende Bedeutung:

- **Modellebene**: Die Modellebene bildet die Schnittstelle zum Anwender eines Simulators. Sie wird im Folgenden auch einfach als *Simulationsmodell* bezeichnet. Üblicherweise wird das Simulationsmodell durch Instanzbildung von Komponenten- und Konnektorprototypen aus einer Klassenbi-

[1] Nach [138] sind komplexe Konnektoren ebenfalls Komponenten, wenn auch eine spezielle Form.

bliothek, deren Verknüpfung und Implementierung erstellt. Die Implementierung beinhaltet typischerweise die Spezialisierung der Prototypen auf eine bestimmte Funktion und damit die finale Festlegung der dynamischen Semantik.

- **Kernelebene**: Die Kernelebene implementiert die Syntax und Semantik einer bestimmten Modellierungssprache zur Beschreibung von Simulationsmodellen (in diesem Fall SystemC). Sie stellt zudem eine Infrastruktur in Form unterschiedlich partitionierter Kernelkomponenten und Kernelkonnektoren (siehe Abschnitt 4.2.4.4) zur Ausführung eines Simulationsmodells zur Verfügung.
- **Logische Ebene**: Die logische Ebene ist die Infrastruktur zur Ausführung und Synchronisation von Kernelkomponenten. Auf der logischen Ebene werden Komponenten und Konnektoren als logische Prozesse und logische Links bezeichnet. Die logische Ebene implementiert ein bestimmtes Synchronisationsverfahren und versteckt deren konkrete Funktion vor der Kernelebene. Diese Separation ist insbesondere für komplexe (z.B. topologieabhängige Verfahren) von Vorteil.
- **Basisdienstebene**: Die Basisdienstebene ist die unterste Ebene des parallelen Simulators *PS*. Die konkrete Ausprägung ist von der Ausführungsplattform abhängig. Auf einem Multiprozessor entsprechen Komponenten und Konnektoren z.B. Prozessen des Betriebssystems und APIs zur Interprozesskommunikation. Typische Aufgaben dieser Ebene sind die Initialisierung, Einbindung und Bereitstellung plattformspezifischer Bibliotheken, welche verschiedene Typen grundlegender Synchronisations- und Transportmechanismen für den verlustfreien Datenaustausch zur Verfügung stellen.
- **Ausführungsplattformebene**: Hier sind alle Komponenten vereint, welche nicht Teil des parallelen Simulators *PS* sind. Diese Ebene repräsentiert die Infrastruktur in Form von Hardware und Software, zur Ausführung des parallelen Simulators. Komponenten können Prozessorkerne inkl. eines darauf ausgeführten Betriebssystems oder ganze Workstations sein. Zu den Konnektoren gehören beispielsweise On-Chip Netzwerke oder *Local Area Networks (LANs)* sowie zugehörige Software APIs.

4.2.3. Simulationssynthese

In Abschnitt 2.1 wurde der Begriff der Synthese im Kontext eines Entwurfsprozesses für eingebettete Systeme als der Prozess der *„Konvertierung einer gegebenen Verhaltensspezifikation in eine strukturelle Spezifikation“* [115] eingeführt. Dabei wurde das Mapping als der essentielle Schritt der Systemsynthese identifiziert.

Diese Sichtweise der Systemsynthese kann 1:1 auf den Prozess der Präparation einer Simulation für die parallele Ausführung übertragen werden. Letzterer ist dabei mit einem plattformbasierten Entwurfsprozess auf Systemebene vergleichbar (Abschnitt 2.1.1.3). Eine *Simulationssynthese* beschreibt demnach den Prozess der (automatisierten) Abbildung eines Simulationsmodells $\mathcal{M}$ auf eine Ausführungsplattform $\mathcal{A}$ mit Hilfe eines parallelen Simulators $\mathcal{PS}$. Liegt der Fokus auf der simulationsbasierten Verifikation eines MPSoC Modells, so lässt sich die Simulationssynthese entsprechend Abb. 4.1 in zwei Schritte gliedern:

- I) Abbildung der Applikation $\mathcal{AP}$ auf das (MPSoC) Simulationsmodell $\mathcal{M}$.
- II) Abbildung des Simulationsmodells $\mathcal{M}$ auf die Ausführungsplattform $\mathcal{A}$ mit Hilfe des parallelen Simulators PS.

4.2.3.1. Abbildung der Applikation auf das Simulationsmodell

Die Abbildung der Applikation auf das Simulationsmodell findet noch außerhalb des Referenzmodells aus Definition 4.1 statt. Dieser Schritt ist typischerweise Teil der Verifikation in einem plattformbasierten Entwicklungsprozess für MPSoCs [75, 261, 151, 145, 189]. Da die Applikation das Verhalten des Simulationsmodells beeinflusst, hat sie auch Einfluss auf die Performanz, wenn das Modell auf einer bestimmten Ausführungsplattform simuliert wird. Im Folgenden wird beispielhaft das *Hermes Multiprocessor System (HeMPS)* [75] und der zugehörige Entwurfsfluss erläutert, da HeMPS in den weiteren Kapiteln als Grundlage zur experimentellen Evaluation dient.

HeMPS (siehe Abb. 4.2) ist ein homogenes MPSoC, das aus einer konfigurierbaren Anzahl von *Processing Elements (PE)* basierend auf einem *MIPS* Prozessor namens Plasma [220] besteht. Diese sind über ein Mesh NoC namens Hermes [203] miteinander verbunden. Jedes PE ist über ein *Network Interface (NI)* an einen dedizierten *Router (RO)* des NoCs angebunden. Die Router vollziehen ein XY Routing in Kombination mit Round Robin Scheduling und Wormhole Switching. HeMPS hat eine Distributed Memory Architektur. Datentransfers basieren auf einem Message Passing Protokoll. Die Plasmakerne führen ein RTOS aus, welches Multithreading unterstützt. Applikationen für HeMPS können als Prozessnetzwerk [153] spezifiziert werden. Prozesse/Tasks werden in einem *Task Repository (TP)* abgelegt. Nur der Master hat Zugriff auf das TP. In der Initialisierungsphase verteilt er die Tasks an die Slaves, bevor diese mit der Ausführung der Applikation beginnen.

Der grundlegende Ablauf des Entwurfsflusses von HeMPS (siehe Abb. 4.3) ist typisch für viele MPSoC Entwurfsmethoden [75, 261, 151, 145, 189]. Er beginnt mit der Spezifikation der Softwareapplikation(en) und der Konfiguration der Zielplattform, bestehend aus Hardware- und Softwarekomponenten. Im Rahmen

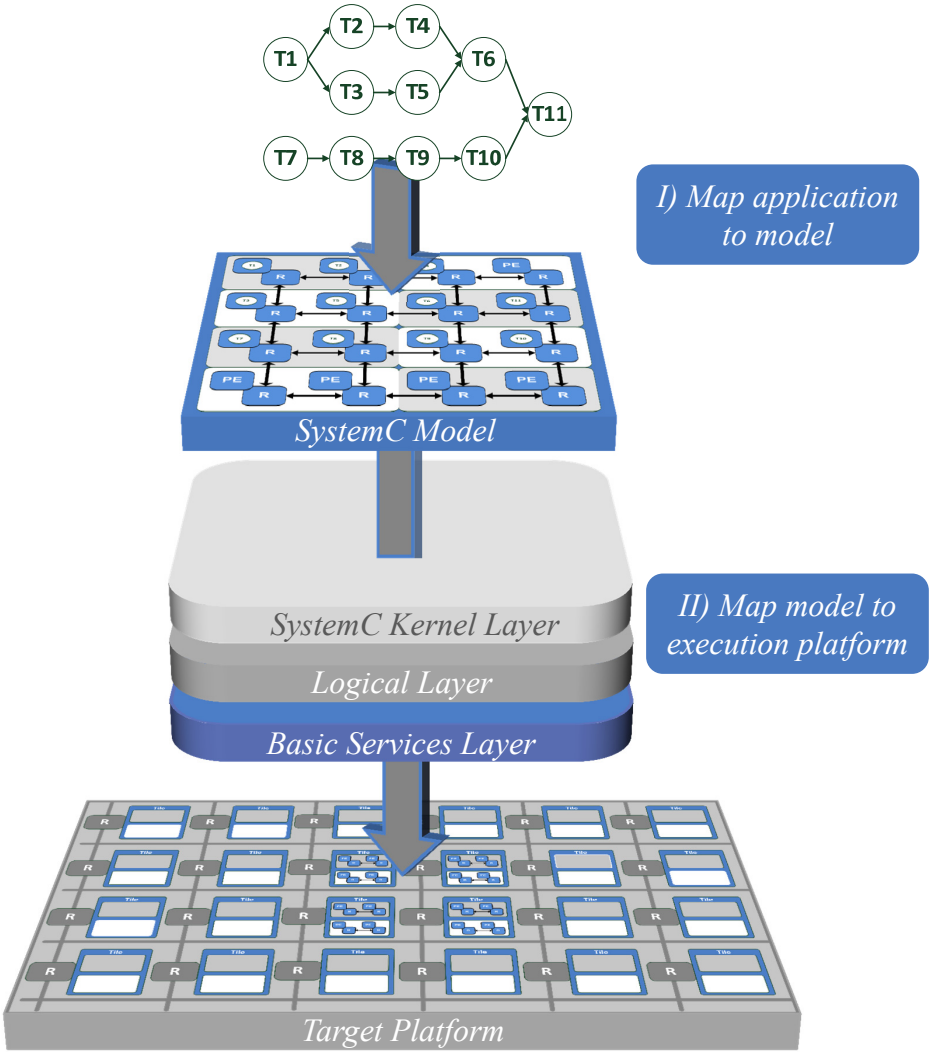


Abbildung 4.1.: Synthese einer parallelen MPSoC Simulation

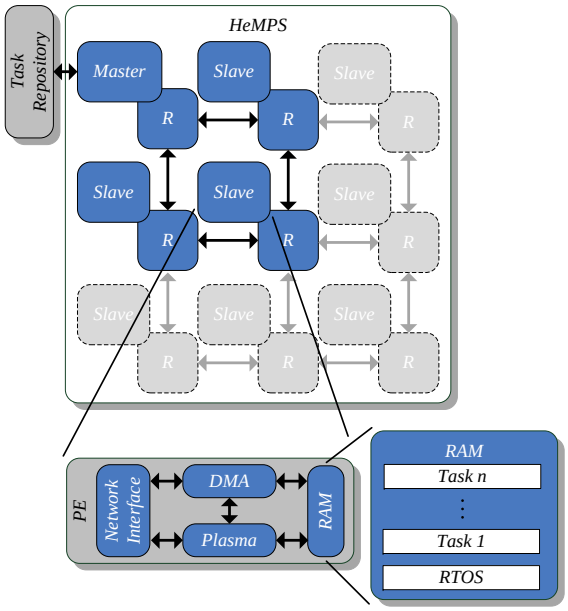


Abbildung 4.2.: Hermes Multiprocessor System (HeMPS) [75]

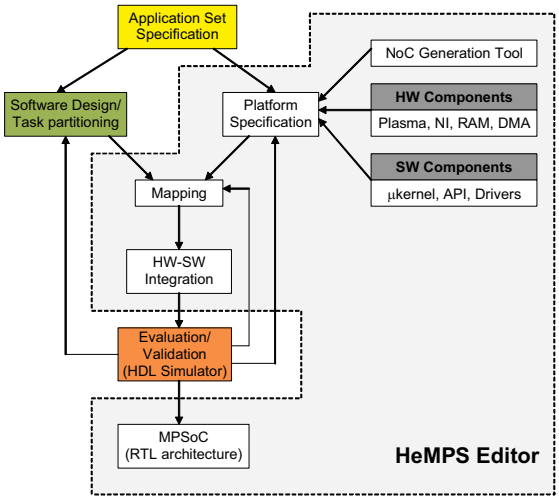


Abbildung 4.3.: HeMPS Entwurfsfluss (Quelle: [75])

des Mappingschritts werden die Tasks der Applikation(en) auf eine bestimmte Plattformkonfiguration abgebildet. Daraus wird ein Simulationsmodell $\mathcal{M}$ generiert und zu Verifikationszwecken ausgeführt. Dieser Prozess wird solange iteriert, bis eine gültige Abbildung gefunden ist. Der beschriebene Prozess wird durch eine grafische Benutzeroberfläche (HeMPS Editor) und diverse andere Softwaretools unterstützt[2].

4.2.3.2. Abbildung des Simulationsmodells auf die Ausführungsplattform

Dieser (wie prinzipiell auch der vorangegangene) Teilschritt lässt sich vereinfachend auf ein *Graphpartitonierungsproblem* [152][248] zurückführen. Dazu können Komponentennetzwerke, die mit Hilfe der Ebenen des Referenzmodells realisiert sind, als Graph $G^n(P^n, V^n)$ modelliert werden, der für jede realisierte Komponente einen Knoten in P^n enthält. Zwischen p_a^n und p_b^n existiert eine Kante $v_{ab}^n \in V^n$, wenn mindestens eine gerichtete Kommunikationsbeziehung zwischen p_a^n und p_b^n existiert.

Mit der vereinfachenden Annahme, dass alle Graphen unterhalb der Modellebene identisch sind, kann die Partitionierung eines Modells als Abbildung von Knoten und Kanten des Graphen G^4 des Modells auf Knoten und Kanten eines beliebigen der vier darunterliegenden Graphen modelliert wird. Die Abbildung erfolgt im Allg. unter bestimmten Randbedingungen, die bei der Partitionierung eingehalten werden müssen (z.B. möglichst optimaler Lastausgleich oder geringer Kommunikationsaufwand) [49]. Zur Schätzung entstehender Kosten müssen Kostenfunktionen entwickelt werden, die als Entscheidungsgrundlage für die Art und Weise der Abbildung dienen.

Erfolgt die Abbildung *statisch* und ändert sich diese nicht mehr zur Laufzeit, so kann die Erstellung einer Kostenfunktion mit Hilfe von *Profiling* [120][124] vor der eigentlichen Simulationslaufzeit unterstützt werden. Falls die Abbildung *dynamisch* zur Laufzeit geändert werden kann, so müssen Performanzwerte während der Ausführung beobachtet und unmittelbar in eine geeignete Lastverteilung umgesetzt werden. Dies entspricht in etwa den Aufgaben eines Regelkreises [49]. Eine statische Abbildung hat den Vorteil, dass die räumliche Datenverfügbarkeit dauerhaft eingeschränkt werden kann (siehe auch Abschnitt 4.2.4.5). Dies reduziert u.U. den Kommunikations- und Synchronisationsaufwand. Statische Abbildung hat den Nachteil, dass mögliche dynamisch auftretende Ungleichgewichte in der Auslastung nicht automatisch ausgeglichen werden können. Bei dynamischer Abbildung liegt der genau entgegengesetzte Fall vor.

[2] Die Originalversion von HeMPS basiert auf einer VHDL RTL Beschreibung. Nur für den Plasmakern existiert zusätzlich ein zyklenapproximativer Instruktionssatzsimulator in SystemC. Daher wurden im Rahmen dieser Arbeit alle existierenden VHDL Teile manuell in äquivalente SystemC RTL Beschreibungen übersetzt und der HeMPS Editor entsprechend angepasst.

Formulierung des Partitionierungsproblems für SystemC

Da SystemC unterschiedliche Modellierungsstile und Abstraktionsgrade unterstützt (vgl. Abschnitt 2.3.2), die unterschiedliche syntaktische Elemente verwenden, ist der Graph G^4 der Modellebene vom verwendeten Modellierungsstil abhängig. In reinen RTL oder ähnlichen Modellen macht es Sinn, Instanzen von Prozessen als Modellkomponenten und Instanzen von *sc_signal* als Modellkonnektoren zu interpretieren. Vernachlässigt man die hierarchische Struktur, so kann ein reines RTL Modell dann als ein Prozess-Signalgraph $G_{PS}(P, S)$ modelliert werden:

Definition 4.2 *(**Prozess-Signal Graph**): Ein **Prozess-Signal Graph** $G_{PS}(P, S)$ ist ein gerichteter Graph mit Knoten $p \in P$ und Kanten $s \in S$. Jeder Knoten repräsentiert genau einen SystemC Prozess. Zwei Knoten p_x und p_y sind durch eine Kante s_{xy} verbunden, wenn die Prozesse, die durch die Knoten p_x und p_y repräsentiert werden, durch mindestens ein Signal verbunden sind, wobei der durch p_x repräsentierte Prozess auf das Signal schreibt und der durch p_y repräsentierte Prozess von diesem Signal ausschließlich liest.*

In TL Modellen ist es hingegen sinnvoller, SystemC Module als Modellkomponenten und Socket- oder Transportverbindungen als Modellkonnektoren zu interpretieren. Ein reines TL Modell kann dann z.B. als ein Modul-Transport Graph $G_{MT}(M, T)$ modelliert werden:

Definition 4.3 *(**Modul-Transport Graph**): Ein **Modul-Transport Graph** $G_{MT}(M, T)$ ist ein gerichteter Graph mit Knoten $m \in M$ und Kanten $t \in T$. Jeder Knoten repräsentiert genau ein SystemC Modul. Zwei Knoten m_x und m_y sind durch eine Kante t_{xy} verbunden, wenn das durch m_x repräsentierte Modul anhand einer Transportmethode den Zustand in dem durch m_y repräsentierten Modul modifizieren kann[3].*

Ein gemischtes RTL/TL Modell, das sowohl RTL als auch TL Anteile enthält, kann durch zwei Teilgraphen G_{PS} und G_{MT} beschrieben werden. Beide Teilgraphen sind dann durch Knoten aus einer Menge A verbunden, wobei $P \cap M = A$. Ein Knoten $a \in A$ repräsentiert ein gemischtes Modul, das sowohl über Signale als auch Sockets kommuniziert. Ein Modul aus A repräsentiert damit ein TL Modul und zugleich einen durch das Modul aggregierten Prozess (eine Menge von Prozessen), der nicht weiter zerlegt werden kann. Darauf basierend lässt sich das Partitionierungsproblem für ein SystemC Modell wie folgt formulieren:

[3] Auf welche Art und Weise diese Modifikation exakt geschieht, ist absichtlich nicht spezifiziert. Diese kann z.B. durch das Schreiben eines Parameters innerhalb einer Transportmethode erfolgen oder umgekehrt, durch Rückgabe eines Wertes bei Verlassen der Methode.

Definition 4.4 *(**SystemC Partitionierungsproblem**): Bei der Partitionierung eines SystemC Modells ist eine Graphenpartitionierung gesucht, die den Graphen $G_{PS} \cup G_{MT}$ partitioniert. Eine Partitionierung beinhaltet die Abbildung von Knoten des Graphen $G_{PS} \cup G_{MT}$ auf Knoten eines beliebigen Graphen G^{target} einer darunterliegenden Ebene und die Abbildung von Kanten der Graphen $G_{PS} \cup G_{MT}$ auf Knoten und Kanten von G^{target}.*

4.2.4. Strategien zur Parallelisierung von SystemC

Als erste Voraussetzung für die Abbildung eines SystemC Modells auf einen Multi- oder Manycoreprozessor müssen Kernelkomponenten bereitgestellt werden, die SystemC Modellpartitionen ausführen können. Zur Umsetzung dieser Kernelkomponenten gibt es eine Reihe von Möglichkeiten. In Abschnitt 3.1.2.4 wurde bereits der Begriff der Kernelpartitionierung eingeführt und existierende Ansätze in *asymmetrisch* und *symmetrisch* klassifiziert. Im nun folgenden Abschnitt soll dieser und weitere Freiheitsgrade sowie Auswirkungen von Entwurfsentscheidungen auch im Hinblick auf die Erfüllung der in Abschnitt 4.1 genannten allgemeinen Anforderungen genauer betrachtet werden.

4.2.4.1. Datenabhängigkeiten bei sequentiellem Scheduling

Der sequentielle SystemC Scheduler implementiert ein dynamisches Scheduling [248]: Welche SystemC Prozesse wie oft und in welcher Reihenfolge ausgeführt werden, steht vor der Laufzeit noch nicht (vollständig) fest. Zu einem bestimmten Simulationszeitpunkt sind nur ein Bruchteil aller zukünftigen Ereignisse bekannt. Diese Dynamik äußert sich in der Existenz zweier Schleifen in der Zustandsmaschine aus Abb. 2.12, der Deltacycle-Schleife und der Timedcycle-Schleife[4].

Die Phasen, die während des sequentiellen Schedulings durchlaufen werden, sind atomar (vgl. Abschnitt 2.3.2.2): Um die kausalen Abhängigkeiten zwischen Ereignissen korrekt aufzulösen, beginnt eine neue Phase immer erst dann, wenn die vorausgehende Phase vollständig beendet ist. Der Grund dafür sind Abhängigkeiten zwischen Phasen auf den in Abschnitt 2.3.2.2 aufgeführten Mengen und Variablen, die aber erst zur Laufzeit bekannt sind. Abhängigkeiten in einem Softwareprogramm können im Allg. in Datenabhängigkeiten, Kontrollflussabhängigkeiten und Resourcenabhängigkeiten klassifiziert werden [41]. Erstere lassen sich wiederum in *Read-After-Write (RAW)*, *Write-After-Read (WAR)* und *Write-After-Write (WAW)* Abhängigkeiten einteilen. Werden solche Abhängigkei-

[4] Immediate Notifications werden in der Zustandsmaschine aus Abb. 2.12 nicht explizit modelliert.

ten nicht beachtet, so wird das Simulationsmodell u.U. nicht kausal (d.h. zeitlich) korrekt ausgeführt (vgl. Definition 2.1).

Um Abhängigkeiten der genannten Typen zu visualisieren, eignet sich die Darstellung anhand eines dynamischen Abhängigkeitsgraphen (engl. *Dynamic Dependency Graph (DDG)*) [34, 41]. Ein DDG ist ein partiell geordneter DAG, dessen Knoten die Ausführung von Instruktionen repräsentieren und dessen Kanten die Abhängigkeiten zwischen den Instruktionen modellieren, die während der dynamischen Ausführung eines Programms entstehen [41].

Aus einer grobgranularen Perspektive können die Kernelphasen des sequentiellen SystemC Schedulers, die innerhalb eines Simulationslaufs ausgeführt wurden, als komplexe Makroinstruktionen und damit Knoten eines DDG in Form einer gerichteten Kette interpretiert werden. Da jede Kernelphase zu jedem Simulationszeitpunkt $t = (\tau, \delta)$ maximal einmal ausgeführt wird, können ausgeführte Kernelphasen mit einem Zeitstempel indiziert werden. Abb. 4.4 illustriert ein Beispiel für einen DDG einer sequentiellen SystemC Ausführung, die vom Zeitpunkt $(0, 0)$ bis zum Zeitpunkt $(7, 2)$ reicht.

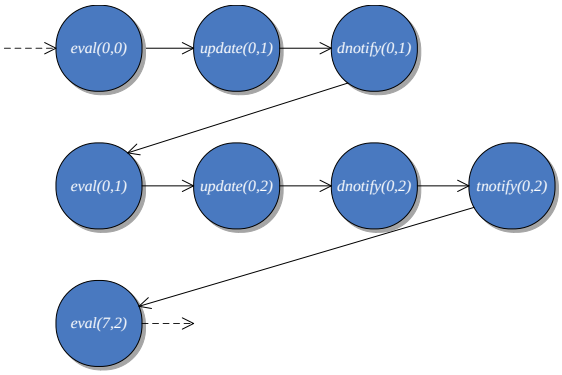


Abbildung 4.4.: DDG einer Ausführung des sequentiellen SystemC Schedulers

Damit eine Strategie zur Parallelisierung des sequentiellen SystemC Kernels gewährleisten kann, dass die Kausalitätsbeziehungen zwischen Ereignissen, wie sie durch sequentielle Ausführung eines Simulationsmodells definiert sind, auch bei paralleler Ausführung desselben Simulationsmodells erhalten bleiben, muss die Strategie im Allg. die Abhängigkeiten berücksichtigen, die durch den DDG der sequentiellen Ausführung spezifiziert sind.

Neben dem DDG aus Abb. 4.4 sind auch andere feingranularere Darstellungen der dynamisch auftretenden Abhängigkeiten möglich. Ein feingranularerer DDG kann z.B. nicht nur komplette Kernelphasen und deren Abhängigkeiten visualisieren, sondern u.U. auch Instruktionen und Abhängigkeiten, die innerhalb

der komplexen sequentiellen Kernelphasen aus Abb. 4.4 selbst existieren. Dies hat den Vorteil einer besseren Analysierbarkeit, erhöht aber die Komplexität.

Um die in einem DDG enthaltenene dynamische Information konservativ zu approximieren, eignen sich im Allg. statische Abhängigkeitsmodelle [96, 210, 141], die typischerweise im Rahmen einer Compileranalyse erstellt werden können. Ebenso wie die Komplexität eines DDG steigt in einer statischen Approximation die Komplexität mit der Verfeinerung der Granularität an.

4.2.4.2. Datenabhängigkeiten bei parallelem Scheduling

Ausgehend vom grobgranularen DDG der sequentiellen Ausführung aus Abb. 4.4 ist der naheliegendste Ansatz zur Parallelisierung, die Vervielfachung einzelner oder aller Knoten und damit, entsprechend der verfügbaren Ressourcen, die Erzeugung jeweils paralleler komplexer Instruktionen gleichen Typs. In einer parallelisierten Phase wird dann Rechenaufwand gleichzeitig bewältigt, der in der ursprünglichen Phase sequentiell abgearbeitet wurde. Die Vervielfachung von Knoten im DDG bzw. von Phasen setzt voraus, dass der Rechenaufwand einer Phase auf parallele Teilphasen gleichen Typs aufgeteilt werden kann. Im Allgemeinen entstehen dabei, orthogonal zu den in Abschnitt 4.2.4.1 erwähnten Typen von Datenabhängigkeiten, folgende neue Typen von Datenabhängigkeiten:

1. **Inter-Phasen-Abhängigkeiten (IRA)**: Abhängigkeiten zwischen aufeinanderfolgenden Teilphasen.
2. **Intra-Phasen-Abhängigkeiten (IAA)**: Abhängigkeiten zwischen parallelen Teilphasen des gleichen Typs.

IRA existierten als Sonderfall bereits bei sequentieller Ausführung. Im parallelen Fall lassen sie sich am einfachsten durch eine zeitliche Partitionierung bzw. die Einführung zusätzlicher Kontrollflussabhängigkeiten in Form globaler Synchronisation nach jeder parallelisierten Phase auflösen. Dieser Ansatz ist sehr konservativ und nur dann notwendig, wenn keine Kenntnisse über Datenpartitionierungen (vgl. Abschnitt 4.2.4.5) oder Eigenschaften des Simulationsmodells (vgl. Abschnitt 4.2.4.6) vorhanden sind. Solches Wissen könnte beispielsweise aus einer detaillierteren statischen Approximation des DDG aus Abb. 4.4 extrahiert werden. Als Beispiel für eine IAA denke man an das gleichzeitige Auslesen von lauffähigen SystemC Prozessen aus einer global verfügbaren Menge lauffähiger Prozesse R während einer parallelisierten Evaluation Phase. Im einfachsten Fall genügt für solche Zugriffe simples Locking, wobei dann die Ausführungsreihenfolge von SystemC Prozessen innerhalb eines Deltacycles evtl. nichtdeterministisch sein kann.

4.2.4.3. Determinismus

Sobald die Zugriffsreihenfolge auf Variablen nicht mehr kontrolliert wird und von zufälligen Verarbeitungs- oder Kommunikationslatenzen abhängt, so ist eine Parallelisierungsstrategie eine potentielle Quelle für nicht-deterministisches Verhalten. Im Fall von SystemC wird die Gefahr von nicht-deterministischem Verhalten dadurch verstärkt, dass die Modellierung auf höheren Abstraktionsebenen oft auf der direkten Kommunikation von SystemC Prozessen über gemeinsam genutzte Variablen basiert (vgl. Abschnitt 2.3.2.3). Dadurch steigt die Wahrscheinlichkeit für sog. nicht-deterministische Anomalien (vgl. [234, 233]. Falls der Kernel keine entsprechende Unterstützung bietet, liegt es in der Verantwortung des Anwenders, explizit Gegenmaßnahmen zu ergreifen.

Da in dieser Arbeit die Untersuchung von Methoden zur parallelen Ausführung zyklenakkurater und -approximativer Modelle auf Manycore Architekturen und weniger eine holistische Betrachtung einer deterministischen Ausführung aller Typen von Modellen im Fokus steht, wird eine Betrachtung von Determinismus nur insoweit mit einbezogen, wie dies für die in dieser Arbeit verwendeten Typen von Modellen und Modellierungstechniken notwendig ist.

4.2.4.4. Kernelpartitionierung

Durch Gruppierung von vervielfältigten Teilphasen unterschiedlichen Typs werden Kernelkomponenten K erzeugt. Dieser Vorgang wird im Folgenden auch als Kernelpartitionierung bezeichnet. Angenommen, nur die Evaluation Phase $eval()$ ist parallelisiert. Eine Möglichkeit zur Gruppierung ist dann die *asymmetrische* Aufteilung des SystemC Kernels in unterschiedliche Typen von Kernelkomponenten, z.B. eine Master Komponente k^m und mehrere Worker Komponenten $K^w = \bigcup_{1 \leq i \leq N} k_i^w$, wobei $K = k^m \cup K^w$. Eine mögliche Phasenaufteilung wäre beispielsweise folgende:

$$P(k^m) = \{init(), update(), dnotify(), tnotify()\} \tag{4.1}$$

$$\forall i : P(k_i^w) = \{eval_i()\} \tag{4.2}$$

Eine asymmetrische Partitionierung hat den Nachteil, dass der Master mit zunehmender Modellgröße und Anzahl an Workern zu einem Flaschenhals werden kann: Zum einen steigt der Berechnungsaufwand innerhalb der sequentiellen Phasen des Masters mit der Modellgröße. Zum anderen steigt bei wachsender Anzahl an Workern der Kommunikationsaufwand im Master. Dies kann durch eine Parallelisierung aller Phasen und *symmetrische* Partitionierung ent-

schärft werden: Sei $K^s = \bigcup_{1 \leq i \leq N} k_i^s$ die Menge aller Kernelkomponenten und N deren Anzahl, so gilt:

$$\forall i : P(k_i^s) = \{init_i(), eval_i(), update_i(), dnotify_i(), tnotify_i()\} \tag{4.3}$$

Variablen und Datenstrukturen des sequentiellen Kernels werden zu Kernelkonnektoren, sofern Kernelkomponenten sie gemeinsam nutzen. Dazu gehören U, R, N^δ, N^τ oder Datenstrukturen innerhalb von Instanzen des Modells. Im Fall eines Signals sind dies z.B. v^{cur}, v^{next} oder das in einem Signal enthaltene Ereignisobjekt ω.

4.2.4.5. Datenpartitionierung

Im Unterschied zur *zeitlichen Partitionierung* von Datenzugriffen durch Synchronisation entsteht durch Datenpartitionierung eine *räumliche Partitionierung* der Daten und Datenzugriffe. Datenpartitionierung ist ein weiterer Schritt, um Datenabhängigkeiten über den in Abschnitt 4.2.4.2 beschriebenen Ansatz hinaus zu reduzieren. Der in Abschnitt 4.2.4.2 beschriebene IAA Konflikt, der beim gleichzeitigen Zugriff auf die Menge der lauffähigen Prozesse R durch mehrere parallele Teilphasen entsteht, kann z.B. durch Partitionierung von R und Beschränkung des Zugriffs von Teilphasen auf jeweils eine Partition von R gelöst werden.

Als Folge einer Datenpartitionierung können Variablen und Datenstrukturen als *intern* oder *extern* deklariert werden. Auf interne Variablen hat nur eine einzige Kernelkomponente Zugriff, auf externe Variablen mehrere. Externe Variablen können wiederum in *lokal extern* und *global extern* eingeteilt werden. Auf lokal externe Variablen hat nur eine Teilmenge aller Kernelkomponenten Zugriff, global externe Variablen sind hingegen für alle Kernelkomponenten erreichbar.

Die global externe Verfügbarmachung von Variablen erhöht einerseits die Flexibilität enorm, andererseits aber auch den Aufwand für die korrekte Behandlung von Datenabhängigkeiten. Eine gezielte Einschränkung der Verfügbarkeit führt zu geringerem Aufwand in der Behandlung von Datenabhängigkeiten aber weniger Flexibilität. Welche Variablen wie deklariert werden können oder müssen, ist letztendlich von der Ausführungsplattform abhängig (siehe auch Abschnitt 2.4).

4.2.4.6. Weiterführende PDES Optimierungsstrategien

Konservative PDES Verfahren (siehe Abschnitt 2.2.3.4) zeichnen sich insbesondere dadurch aus, dass sie Methoden zur Reduktion von kausalen Abhängigkeiten

zwischen Ereignissen und folglich zur Steigerung der Parallelität bereitstellen, die über eine reine Datenpartitionierung hinausgehen. Diese Verfahren basieren insbesondere auf der effizienten Ausnutzung von modellspezifischer Information zur Reduktion des Synchronisationsaufwands [113]. Im Vergleich zum Ansatz aus Abschnitt 4.2.4.2 ermöglicht die Ausnutzung modellspezifischer Informationen, Datenabhängigkeiten zwischen aufeinanderfolgenden parallelisierten Kernelphasen eines grobgranularen DDG genauer zu identifizieren und die globale Synchronität durch die feingranularere Betrachtungsweise zu durchbrechen. Dabei können bestimmte Strategien der Datenpartitionierung von Vorteil sein. Eine Optimierung im Sinne der PDES lässt sich im Allg. in drei Schritte einteilen:

1. **Extraktion modellspezifischer Information**: Dies geschieht entweder manuell oder automatisiert. Die manuelle Extraktion hat den Vorteil, dass durch Anwenderwissen auch spezielle modellspezifische Datenabhängigkeiten und komplexere Zusammenhänge berücksichtigt werden können, kann allerdings zeitaufwändig sein. Eine automatisierte Extraktion ist effizienter, funktioniert aber nur so gut, wie es das verwendete Werkzeug zulässt. Können komplexe Datenabhängigkeiten nicht erkannt werden, so können kausale Fehler nur durch geringere Parallelität vermieden werden.
2. **Vorhersage von Garantien für kausale Unabhängigkeit**: Auf Basis der extrahierten Informationen werden Garantien für existierende kausale Unabhängigkeiten berechnet. Diese Berechnung kann entweder statisch vor der Laufzeit oder dynamisch zur Laufzeit erfolgen. Eine statische Berechnung erzeugt keinen zusätzlichen Laufzeitoverhead, kann aber evtl. zu konservative Vorhersagen zur Folge haben. Eine dynamische Berechnung erzeugt zusätzlichen Laufzeitoverhead, kann aber aktuelle Laufzeitinformationen in die Berechnung mit einfließen lassen, was möglicherweise zu weniger konservativen Vorhersagen führt.
3. **Adaption von Modell und parallelem Simulator**: Wünschenswert ist es, dass das Modell und der paralleler Simulator bis zu einem gewissen Grad anpassbar bzw. konfigurierbar sind und die Vorhersagen aus Schritt 2) gezielt zur Reduktion des Kommunikationsaufwands genutzt werden können. Eine geeignete Abbildung des Simulationsmodells kann z.B. zu einer besseren Lastverteilung, besserer Datenlokalität und besseren Garantien für kausale Unabhängigkeit führen. Eine solche Adaption kann ebenfalls manuell oder automatisiert erfolgen, wobei in komplexen Fällen eine automatisierte Adaption grundsätzlich vorzuziehen ist.

4.2.5. Überblick über implementierte Komponenten

Abb. 4.5 gibt einen Überblick über Verfahren, die auf den einzelnen Ebenen des Referenzmodells entwickelt wurden. Die Verfahren wurden als Erweiterung des

freien SystemC Kernels [1] in Form einer Klassenbibliothek umgesetzt. Von dieser Bibliothek wurden schrittweise neue Versionen entwickelt, welche die aufgeführten Verfahren (teilweise gleichzeitig) unterstützen.

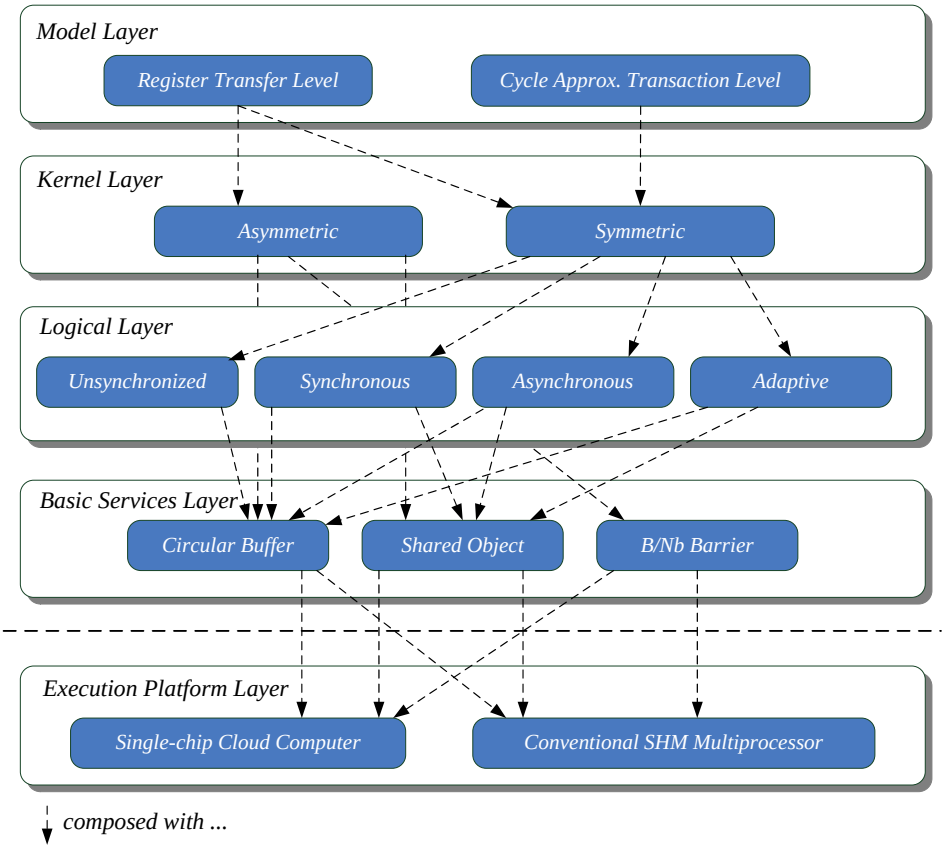


Abbildung 4.5.: Implementierte Konzepte

Da der Fokus auf der parallelen Simulation zyklenakkurater und zyklenapproximativer Modelle liegt, wird auf Modellebene generell mindestens das RTL Subset von SystemC [244] unterstützt. In Abschnitt 4.6 wird darüber hinaus eine TL Modellierungsmethode vorgestellt, die es erlaubt, die Genauigkeit von Modellen statisch oder dynamisch zu skalieren.

Der sequentielle SystemC Kernel wurde in zwei verschiedenen Varianten partitioniert. Dem Verfahren aus Abschnitt 4.3 liegt eine asymmetrische Kernelpartitionierung zugrunde. Die Verfahren aus den Abschnitten 4.4, 4.5 und 4.6 basieren auf einer symmetrischen Partitionierung.

Auf der logischen Ebene wurden vier verschiedene Synchronisationsverfahren implementiert, asynchron (vgl. Abschnitt 4.4), adaptiv (vgl. Abschnitt 4.5) synchron und unsynchronisiert (Abschnitt 4.6). Die synchrone Variante auf der logischen Ebene wurde zum Verfahren aus Abschnitt 4.5 als Vergleichsfall implementiert (vgl. Abschnitt 4.5.7) und entspricht einer durch statische Partitionierung optimierten Version des asymmetrischen Verfahrens aus Abschnitt 4.3.

Der Simulator kann mit Hilfe zweier verschiedener Backends auf Basisdienstebene sowohl auf den SCC als auch auf gewöhnliche cachekohärenten SHM Multiprozessoren abgebildet werden. Das SCC Backend ist eine objektorientierte Erweiterung der SCC-spezifischen RCCE Bibliothek [198, 15]. Die Erweiterungen beinhalten in der Hauptsache neue Schnittstellen, z.B. zur einfacheren Instanziierung gemeinsam genutzter Datenstrukturen in verschiedenen Speicherbereichen (Shared Objects) sowie zur Implementierung zentral koordinierter blockierender oder nicht-blockierender Barrieren (B/Nb Barrier). In beiden Fällen wurden bereits existierende Funktionen der RCCE API wiederverwendet. Des Weiteren wurde als neues Verfahren ein statischer/dynamischer Ringpuffer (Circular Buffer) zur asynchronen einseitigen On-Chip Kommunikation entwickelt (vgl. Anhang A). Das SHM Backend ist eine 1:1 Übertragung des SCC Backends durch Austausch von RCCE spezifischen Funktionen und Nutzung von herkömmlichem SHM zur Implementierung der genannten Datenstrukturen.

4.3. Asymmetrische synchrone Strategie

Im folgenden Abschnitt soll die Eignung einer asymmetrischen Kernelpartitionierung in Kombination mit einem vollständig synchronen Synchronisationsverfahren für den SCC untersucht werden. Ähnliche Verfahren haben sich auf cachekohärenten SHM Multiprozessoren als nützlich erwiesen: Die geringe Kommunikationslatenz auf diesen Architekturen erlaubt es, den hohen Synchronisationsaufwand vollständig synchroner Verfahren in Kauf zu nehmen [232, 105]. Durch die diesen Verfahren typischerweise zugrundeliegende asymmetrische Master/Worker Architektur ist auf einfache Weise eine dynamische Lastverteilung möglich. Weiterführende Optimierungen im Sinne einer Kausalitätsanalyse werden bei vollständig synchronen Verfahren nicht vorausgesetzt [44].

Im Folgenden werden zugrundeliegende Konzepte und Entwurfsentscheidungen erläutert. Anschließend werden Aspekte der Implementierung und Optimierungspotentiale beschrieben. Die Leistungsfähigkeit des Ansatzes wird anhand mehrerer Fallstudien untersucht und bewertet. Die Ergebnisse dieses Abschnitts wurden im Rahmen einer vom Autor betreuten Bachelorarbeit [Red11] erarbeitet und sind somit in Zusammenarbeit entstanden. Sie sind in [RRS+12] publiziert.

4.3.1. Anforderungen und Konzept

Ausgehend von bekannten Arbeiten zu vollständig synchroner Parallelisierung [232, 105], lagen dem entwickelten Konzept folgende spezielle Anforderungen zugrunde:

I) **Modellierungsmethode**: Es soll zumindest das RTL Subset von SystemC [244] unterstützt werden. Das Modell soll dabei prinzipiell als ein Prozess-Signal Graph G_{PS} beschreibbar sein (vgl. Definition 4.2).

II) **Abbildung des Simulationsmodells**: Es soll eine einfache dynamische Abbildung des Simulationsmodells auf die Ausführungsplattform möglich sein.

III) **Datenpartitionierung**: Auf eine Datenpartitionierung in interne und externe Daten soll zugunsten der einfachen dynamischen Lastverteilung und einer generellen global externen Deklaration verzichtet werden.

IV) **Kernelpartitionierung**: Der Kernel soll asymmetrisch partitioniert werden. Dabei sollen ausschließlich die $eval()$ und die $update()$ Phase parallelisiert sein. Kernelkomponenten sind in einen Master k^m und n Worker $k_1^w ... k_n^w$ unterteilt. Dabei gilt:

$$P(k^m) = \{init(), dnotify(), tnotify()\} \tag{4.4}$$

$$\forall i > 0 : P(k_i^w) = \{eval_i(), update_i()\} \tag{4.5}$$

V) **Synchronisation**: Die Kernelkomponenten sollen synchron in der Zeit voranschreiten.

Das resultierende Konzept ist in Abb. 4.6 illustriert. Da in Anforderung I nur das RTL Subset vorausgesetzt wird, hat die Ausführungsreihenfolge von SystemC Prozessen während eines Deltacycles keinen Einfluss auf das Ergebnis der Simulation[5]. Zur Sicherstellung der Kausalität müssen deswegen nur IRA berücksichtigt werden. Bei gleichzeitigem Zugriff auf ein und dieselbe Datenstruktur innerhalb einer Phase (IAA) genügt es, wenn wechselseitiger Ausschluss garantiert ist. Wegen der Anforderungen II und III benötigen alle Kernelkomponenten Zugriff auf einen global zugänglichen Zustandspeicher. Der Verzicht auf eine statische Partitionierung von Daten reduziert die dynamische Abbildung des Simulationsmodells auf die dynamische Abbildung von SystemC Prozessen. Wegen Anforderung V erfolgt nach jeder Kernelphase eine globale Synchronisation. Entstehende globale Wartezustände ermöglichen eine zentral gesteuerte dyna-

[5] Da SystemC Prozesse durch das E/U Paradigma grundsätzlich um mindestens einen Deltacycle voneinander entkoppelt sind, genügt die Herstellung der durch die Deltacycles definierten **partiellen Ordnung** von Prozessaktivierungen.

mische Verteilung von Prozessen durch einen zentralen Master (Anforderung IV).

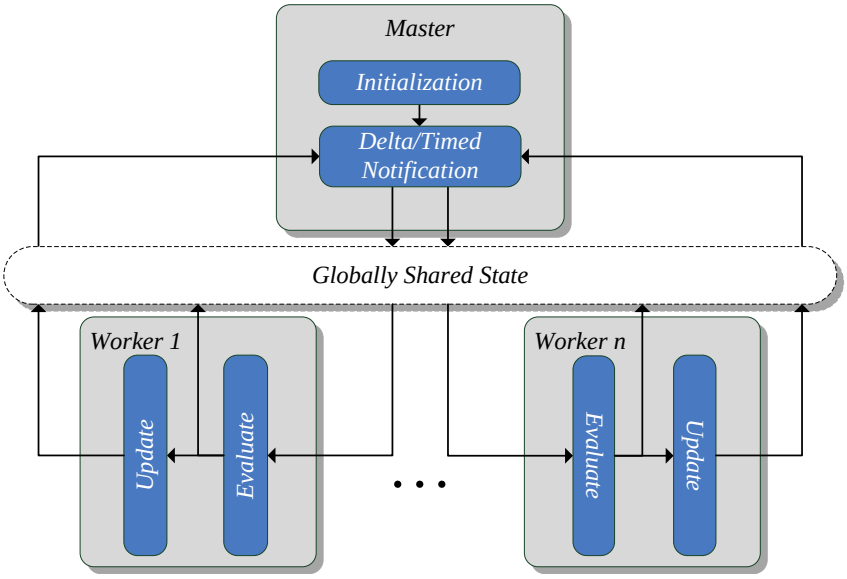


Abbildung 4.6.: Architekturkonzept der asymmetrischen synchronen Strategie

4.3.2. Datenpartitionierung

Als Basis für die Umsetzung einer jeden Kernelkomponente dient jeweils ein separater sequentieller SystemC Kernel. Damit existieren für U, N^{τ} und N^{δ}, R, τ und δ (vgl. Abschnitt 2.3.2.2) bereits in jeder Kernelkomponente interne Duplikate. Im Folgenden werden Varianten zur Umsetzung einer Datenpartitionierung anhand der genannten Variablen und Mengen diskutiert.

- **Aktualisierungsanfragen**: Diese werden von Workern k_i^w in den $eval_i()$ Phasen generiert und in den $update_i()$ Phasen verarbeitet. Wenn ein Worker k_i^w immer für die Durchführung genau der Updates verantwortlich ist, die er selbst generiert hat, können die n internen Datenstrukturen $U_1 ... U_n$ in den Workern komplett unabhängig voneinander verwendet werden. Teure gleichzeitige Zugriffe auf eine globale Datenstruktur U in der $eval_i()$ oder $update_i()$ Phase werden dadurch vermieden.
- **Lauffähige Prozesse**: Diese werden vom Master in $dnotify()$ bzw. $tnotify()$ erzeugt und von den Workern in den $eval_i()$ Phasen verarbeitet. Selektiert

jeder Worker k_i^w die von ihm auszuführenden Prozesse in der $eval_i()$ Phase selbstständig aus einem globalen R, verteilt sich die Rechenkomplexität automatisch gleichmäßig (sog. Work-Stealing [58]). Dies hat allerdings den Nachteil, dass zu viele gleichzeitige Zugriffe aufgrund des Koordinationsaufwands evtl. zu einem Flaschenhals werden können. Durch Verteilung der Einträge auf n Nachrichtenwarteschlangen $R_1^{mq}...R_n^{mq}$ wird daher die Verfügbarkeit insgesamt auf lokal extern eingeschränkt und so der Flaschenhals verringert: Der Master ist so für den Lastausgleich und die Zuteilung von lauffähigen SystemC Prozessen auf die $R_1^{mq}...R_n^{mq}$ zu den Workern verantwortlich (sog. Work-Sharing [97]).

- **Timed-/Delta-Notifications**: Diese werden von Workern in den $eval_i()$ und $update_i()$ Phasen generiert und vom Master in der $dnotify()$ bzw. $tnotify()$ Phase verarbeitet. Datenstrukuren zur Speicherung können als gemeinsam genutzte Strukturen N^τ und N^δ realisiert werden, mit dem bereits erwähnten Risiko, dass viele Workerzugriffe einen Flaschenhals verursachen. Zudem benötigen die Worker keinen Zugriff auf in der Zukunft liegende Notifications in N^τ. Als Alternative werden die Notifications daher wieder über n lokal externe Nachrichtenwarteschlangen $N_1^{mq}...N_n^{mq}$ von den k_i^w an den Master übermittelt.
- **Simulationszeit τ und Deltacycle δ**: τ muss extern verfügbar sein. Da nur der Master in der $tnotify()$ Phase auf τ schreiben kann, wird τ global extern deklariert. Die δ Variable wird vollständig als interne Kopie in jeder Kernelkomponente vorgehalten und separat inkrementiert.
- **Übrige Datenstrukturen des Modells**: Diese beinhalten z.B. Ereignisobjekte, Channel-, Modul- und Prozessinstanzen, etc. Aufgrund der Anforderungen II und III werden zunächst nur Datenstrukturen als intern klassifiziert, die während der Evaluation, Update und Notification Phasen nicht veränderlich sind. Die restlichen Variablen sind initial global extern verfügbar.

4.3.3. Globale Barriersynchronisation

Eine einfache Möglichkeit zur Realisierung globaler Synchronisation ist ein generischer Ansatz basierend auf globalen Barrieren. Ein solcher ist unabhängig von einer bestimmten Topologie des Simulationsmodells bzw. dessen Abbildung auf die Kernelkomponenten. Durch globale Barrieren wird die Simulation regelmäßig in einen global definierten Zustand überführt. Entstehende globale Wartezustände können zur zentral gesteuerten Lastumverteilung genutzt werden. Folgende Barrieren sind notwendig:

- **Evaluation nach Update**: Stellt das E/U Paradigma sicher: Worker dürfen Primitive Channels erst aktualisieren, wenn sichergestellt ist, dass sich kein Worker mehr in der Evaluation Phase befindet.
- **Update nach Notification**: Stellt sicher, dass mit Ende der Update Phase alle Notifications von den Workern an den Master übermittelt wurden.
- **Notification nach Evaluation**: Mit dieser Barrier initiiert der Master die Evaluation Phase und teilt den Workern die neue Simulationszeit mit.

4.3.4. Integration von Kommunikation und Synchronisation

Für die Prüfung von binären Entscheidungen und des Terminierungszustandes werden neben den im vorigen Abschnitt erwähnten neuen Datenstrukturen in jeder Kernelkomponente binäre Variablen *pass* und *term* eingeführt. Zum reinen Datenaustausch sind im Master k^m folgende Aktionen implementiert:

- *readNotifications()*: Lese vorhandene Notifications aus den N_i^{mq} aus und füge sie entsprechend ihrem Typ in N_0^τ oder N_0^δ ein. Prüfe gleichzeitig, ob alle SystemC Prozesse evaluiert sind. Wenn ja, dann gib eine 1 zurück, wenn nein, dann eine 0 [6].
- *distWork()*: Falls $|R_0| \neq 0$, dann verteile die lauffähigen Prozesse auf die Worker. Gib eine 1 zurück, sobald $|R_0| = 0$, ansonsten eine 0.
- *terminate()*: Sende ein Terminierungssignal an die Worker, welches das Ende der Simulation anzeigt.

In einem Worker k_i^w werden zum Datenaustausch folgende zusätzliche Aktionen benötigt:

- *readRunnables()*: Kopiere alle in R_i^{mq} vorhandenen Handles von lauffähigen Prozessen nach R_i.
- *nb/b_sendNotifications()*: Versuche, alle aktuell vorhandenen Notifications über N_i^{mq} an den Master zu versenden. Falls nicht ausreichend Speicher vorhanden ist, gib die Kontrolle an die Zustandsmaschine zurück (*nb*) oder warte, bis ausreichend Speicher vorhanden ist (*b*).
- *b_send_notify()*: Sende alle aktuell vorhandenen Notifications über N_i^{mq} an den Master. Blockiere solange, bis alle Notifications verschickt sind.
- *check_terminate()*: Prüfe, ob das Terminierungssignal gesetzt ist, wenn ja, dann gib eine 1 zurück, wenn nicht, dann eine 0.

[6] Damit ein Worker dem Master die erfolgreiche Evaluation von SystemC Prozesses mitteilen kann, existieren zusätzlich sog. *Confirmation Notifications*, welche ebenfalls über die N_i^{mq} verschickt werden.

Zur Barriersynchronisation existieren sowohl im Master k^m als auch in einem Worker k_i^w folgende Aktionen:

- *b_barrier()*: Initialisiere eine Barriere. Im Fall eines Workers, trete der Barriere bei und warte bis der Master sie freigibt. Im Fall des Masters, warte bis alle Worker der Barriere beigetreten sind und gib die Barriere dann frei.
- *nb_checkIn()*: Initialisiere eine Barriere. Im Fall eines Workers, trete der Barriere zusätzlich direkt bei. Verlasse die Aktion anschließend (Master und Worker).
- *barrier_passed()*: Im Fall eines Workers, warte bis der Master die Barriere freigibt. Im Fall des Masters, warte bis alle Worker der Barriere beigetreten sind und gib die Barriere dann frei.

Die im Master und den Workern implementierten endlichen Zustandsautomaten (engl. *Finite State Machines (FSM)*) sind in den Abb. 4.7 und 4.8 dargestellt.

4.3.5. Abbildung auf die Speicherarchitektur des SCC

Da die Mechanismen zur Kommunikation und Synchronisation nicht von der Topologie abhängig sind, kann auf eine logische Ebene verzichtet werden. Master und Worker werden direkt mit RCCE *Units of Execution (UEs)* [15] als Betriebssystemprozesse implementiert. Die Synchronisationsaktionen werden unmittelbar mit Hilfe blockierender und nicht-blockierender Barrierprimitive (*B/Nb Barrier*, vgl. Abschnitt 4.2.5) der Basisdienstebene umgesetzt. Vorhandene Datenstrukturen werden mit Primitiven der Basisdienstebene auf unterschiedliche Weise in die Speicherbereiche des SCC abgebildet. Abb. 4.9 illustriert die initiale Verteilung.

Im privaten Speicher jedes beteiligten SCC Kerns befindet sich ein vollständiges Duplikat des Programmcodes. Auch von statischen Variablen wird in jeder Kernelkomponente eine Kopie im privaten Speicher angelegt. Zu den statischen Variablen werden auch solche gezählt, die sich nur einmal in der Initialization Phase ändern. Deren Wert wird u.U. während der Initialisierung einmal aus dem SHM aktualisiert. Die in Abschnitt 4.3.2 erwähnten Nachrichtenwarteschlangen werden unter Verwendung von statischen Ringpuffern (Circular Buffer Primitive, vgl. Abschnitt 4.2.5 und Anhang A) auf den MPB abgebildet. Die Variable τ wird als Shared Object im MPB abgelegt. Dynamisch veränderliche Variablen des Modells werden zunächst vollständig als Shared Objects in den externen SHM ausgelagert.

Da sich der Cached Mode des SCC wegen der Notwendigkeit vieler teurer L2 Cache Flushes (vgl. Abschnitt 2.4.3.2) im Verlauf der Implementierung als sehr langsam herausgestellt hat (siehe hierzu auch [223] oder [249]), bleibt die Ca-

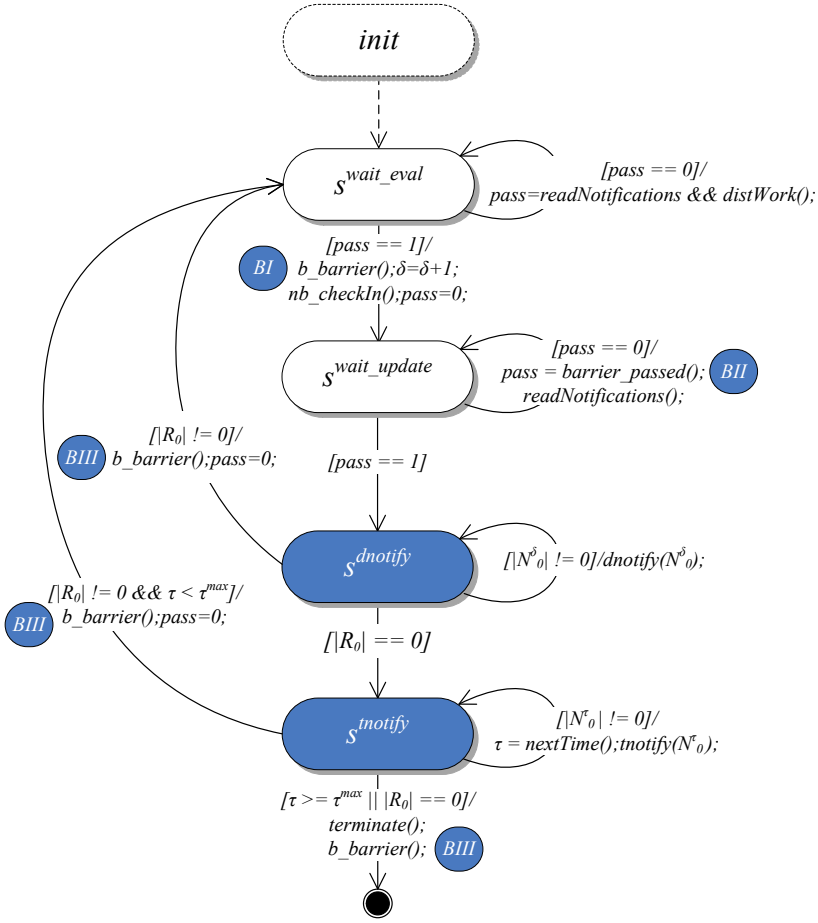


Abbildung 4.7.: Zustandsmaschine im Master k^m

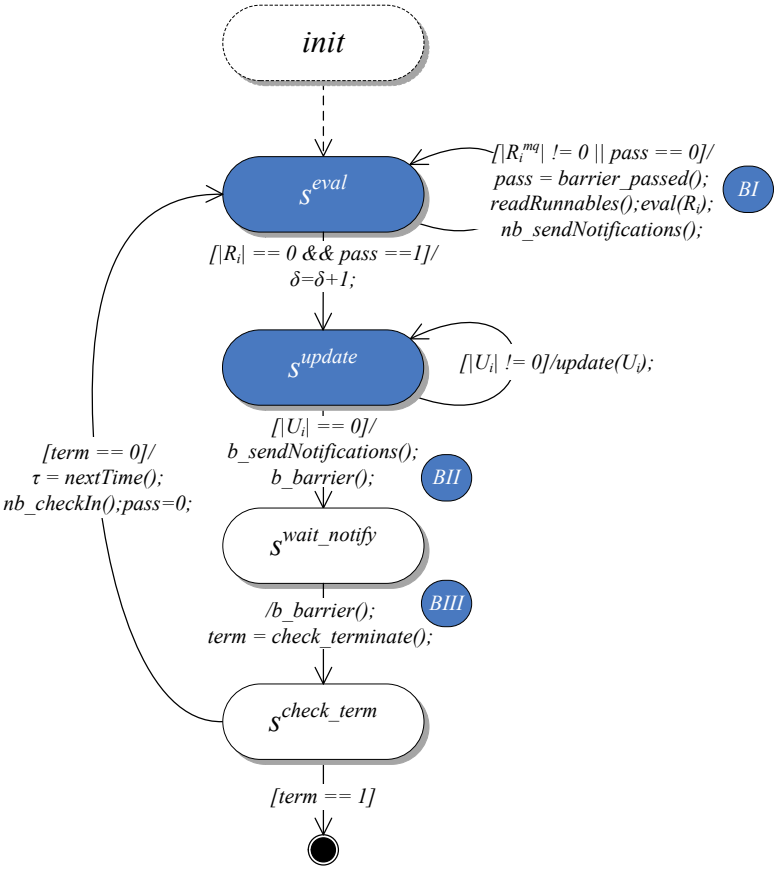


Abbildung 4.8.: Zustandsmaschine in einem Worker k_i^w

cheunterstützung für Zugriffe auf den externen SHM vollständig deaktiviert (UCM).

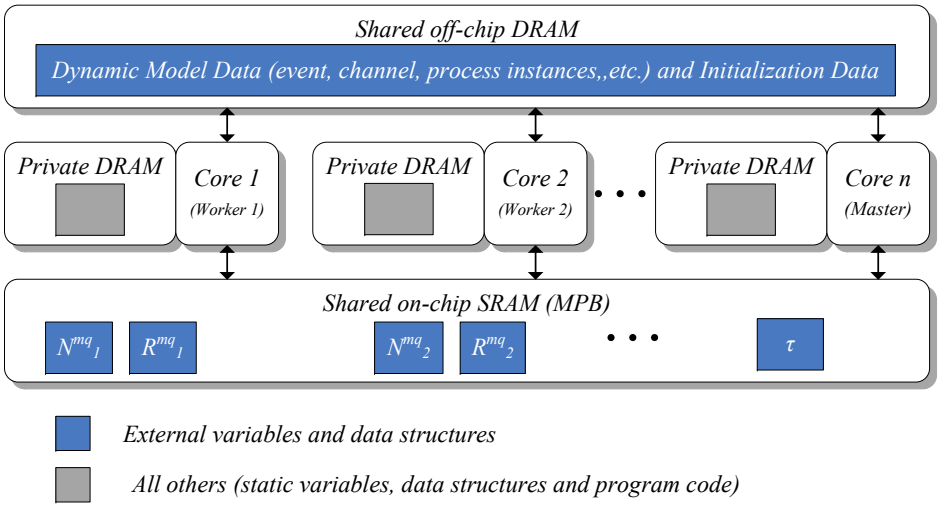


Abbildung 4.9.: Speichernutzung bei asymmetrischer Kernelpartitionierung

4.3.6. Weiterführende Strategien

4.3.6.1. Verbesserter dynamischer Lastausgleich

Eine der Hauptaufgaben des Masters ist die gleichmäßige Verteilung der zu evaluierenden Prozesse auf die R_i der Worker k_i^w (Work Sharing). In der Zustandsmaschine aus Abb. 4.7 geschieht dies innerhalb der Aktion *distWork()*. In der Grundversion legt der Master mit jeder neuen Evaluation Phase zunächst eine initiale Verteilung fest.

Da verschiedene SystemC Prozesse unterschiedliche Last erzeugen, kann es vorkommen, dass ein Worker während der Evaluation Phase leerläuft und andere Worker noch beschäftigt sind. Der dynamische Lastausgleich kann deswegen dahingehend verbessert werden, dass der Master auch während der Evaluation Phase die Last dynamisch umverteilt. Um einen Leerlauf nach Möglichkeit zu vermeiden, beobachtet der Master in der aktuellen Implementierung die R_i^{mq} Warteschlangen und verteilt noch zu bearbeitende Prozesse in den Warteschlangen bei Bedarf um.

Ein zusätzlich zum Work Sharing überlagertes Work Stealing wird dann angewendet, wenn eine der Warteschlangen tatsächlich leerläuft, ohne dass der Mas-

ter schnell genug aktiv wurde, um diesen Fall zu vermeiden. Der entsprechende Worker versucht dann selbstständig, Prozesse aus den Warteschlangen der anderen Worker zu „stehlen“.

4.3.6.2. Statische Abbildung von *SC_THREAD* Prozessen

Da jeder SystemC Prozess auf einem beliebigen Worker ausgeführt werden kann, werden die Stacks von *SC_THREAD* Prozessen in der Grundversion im externen SHM abgelegt. Durch die Größe der Stacks von 64 KB kann die auf dem SCC fehlende hardwareseitige Cachekohärenz die Ausführungsperformanz negativ beeinflussen. Um die Anzahl der Zugriffe auf den als UCM konfigurierten externen SHM zu reduzieren, kann der Master deswegen so eingestellt werden, dass er lauffähige *SC_THREAD* Prozesse immer dem gleichen Worker zuordnet und auf dynamische Verteilung von *SC_THREAD* Prozessen verzichtet. Damit kann der Stack eines Prozesses im privaten Speicherbereich des entsprechenden Workers abgelegt werden.

Existieren nur *SC_METHOD* Prozesse, was in RTL u.ä. Modellen typisch ist, wird die dynamische Lastverteilung anhand von Work Stealing und Work Sharing durch die statische Abbildung von *SC_THREAD* Prozessen nicht beeinflusst. Auch bei Existenz von *SC_METHOD* und *SC_THREAD* Prozessen wird eine vollständige Aushebelung der dynamischen Lastverteilung dadurch vermieden, dass die aktuelle Implementierung immer zuerst die *SC_THREAD* Prozesse und erst dann die *SC_METHOD* Prozesse in die R_i^{mq} eingefügt. Dies lässt sich wie folgt erklären: Da das Work Sharing Verfahren immer auf den zuletzt in die Warteschlangen eingefügten Prozessen arbeitet, sind diese aller Wahscheinlichkeit nach vom Typ *SC_METHOD*. Da das Work Stealing Verfahren erst dann aktiv wird, wenn bereits eine Warteschlange leergelaufen ist, sind die statisch abgebildeten *SC_THREAD* Prozesse mit hoher Wahrscheinlichkeit bereits verarbeitet.

4.3.6.3. Lokale Zustandspufferung

Treten innerhalb eines Deltacycles auf einem Kern mehrere Schreib- oder Lesezugriffe auf eine bestimmte v^{next} oder v^{cur} Variable eines Primitive Channels auf, so kann dies mehrere teure aber unnötige SHM Zugriffe zur Folge haben. Die Ursache ist, dass der dynamische Zustand des Modells in der Grundversion der Parallelisierungsstrategie, insbesondere die v^{next} und v^{cur} Variablen, vollständig im externen SHM abgelegt werden (vgl. Abschnitt 4.3.5).

Durch Ausnutzung des E/U Paradigmas (vgl. Abschnitt 2.3.2.2) ist es möglich, die L1 und L2 Hardware Caches der SCC Kerne zu verwenden und den beschriebenen Sachverhalt zu entschärfen. Dazu werden zusätzlich zu den stati-

schen Modelldaten (vgl. Abschnitt 4.3.5) lokale Kopien $v^{cur,cpy}$ und $v^{next,cpy}$ aller v^{cur} und v^{next} Variablen im privaten Speicherbereich eines jeden SCC Kerns abgelegt. Diese Kopien erlauben während eines Deltacycles eine weitgehend rein lokale Ausführung von SystemC Prozessen. Die Kohärenz zwischen den Kopien in den Caches der SCC Kerne wird softwareseitig hergestellt. Zu diesem Zweck müssen die *read()*, *write()* und *update()* Methoden der SystemC Channels entsprechend adaptiert werden. Die Funktionsweise ist wie folgt (vgl. Abbildung 4.10):

Bei einem Schreibzugriff durch Aufruf von *write()* wird der neue Wert eines Channels zunächst in der lokalen $v^{next,cpy}$ Kopie abgelegt. Erst in der Updatephase wird durch den Aufruf von *update()* sowohl die lokale Kopie $v^{cur,cpy}$ als auch die v^{cur} Variable im SHM mit dem Wert der $v^{next,cpy}$ Variablen aktualisiert. Bei einem Lesezugriff durch Aufruf von *read()* wird der Wert der $v^{cur,cpy}$ Variablen immer nur beim ersten Auslesen innerhalb eines Deltacycles mit dem Wert des zugehörigen v^{cur} aus dem externen SHM aktualisiert[7].

4.3.6.4. Statische Abbildung von beliebigen SystemC Prozessen

Typischerweise haben nur wenige im Modell vorhandene SystemC Prozesse direkten Zugriff auf ein und dieselbe Menge von Zustandsvariablen. Werden alle Prozesse, die potentiell auf eine Zustandsvariable zugreifen, auf den gleichen SCC Kern statisch abgebildet, so kann diese Zustandsvariable in dessen privaten Speicher abgebildet werden[8]. Dadurch wird der Kommunikationsaufwand noch weiter verringert. Dabei sollte jedoch beachtet werden, dass durch die statische Abbildung beliebiger Prozesstypen der dynamische Lastausgleich weiter eingeschränkt wird.

Die Umsetzung basiert auf einer Anwenderschnittstelle, mit deren Hilfe Prozessgruppen manuell definiert werden können. Die in einer Prozessgruppe befindlichen Prozesse werden dann immer auf ein und demselben SCC Kern evaluiert. Da eine externe Vorhaltung von Zustandsvariablen, auf die nur die Prozesse einer Gruppe zugreifen, nicht mehr notwendig ist, können diese dauerhaft als interne Variablen in den privaten Speicher des ausführenden Workers abgebildet werden. Zur Klassifikation von Channels in intern oder extern existiert deswegen ebenfalls eine Anwenderschnittstelle.

[7] Die aktuelle Implementierung setzt diesen Mechanismus ausschließlich für Signale vom Typ *sc_signal* um. Bei anderen Primitive Channels, welche z.B. mehrere Writer unterstützen, kann eine verteilte Koordination der Aktualisierung z.B. dadurch verhindert werden, dass alle Prozesse, die gleichzeitig schreibenden Zugriff auf ein und dasselbe Signal haben, immer auf den gleichen Worker abgebildet werden.

[8] Prozesse, die potentiell auf einen Primitive Channel bzw. dessen Zustandsvariablen zugreifen, können dadurch identifiziert werden, dass in ihrem Rumpf ein *read()* oder *write()* Aufruf auf dem betreffenden Channel erfolgt.

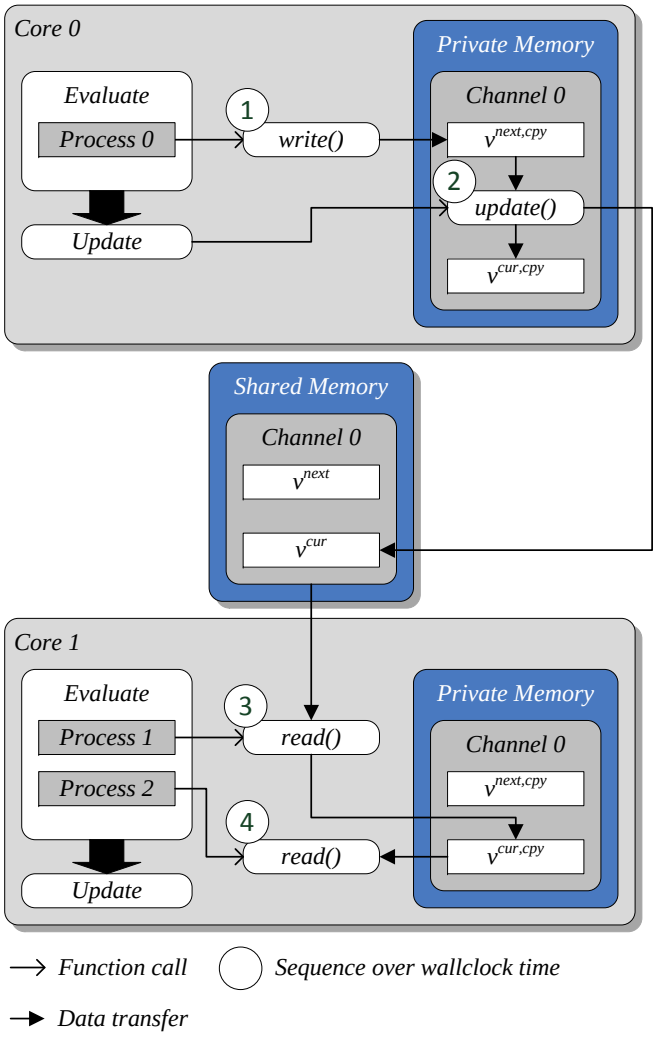


Abbildung 4.10.: Pufferung beim Evaluate/Update Paradigma (Quelle: [Red11])

Für die Gruppierung und Klassifikation von Prozessen und Channels wurde bewusst eine für den Anwender sichtbare Schnittstelle gewählt. Als Erweiterung könnte entsprechender Quellcode bei Bedarf auch automatisch auf Basis einer Analyse des Prozess-Signal Graphen G_{PS} erzeugt werden. Eine dafür geeignete Werkzeugkette wurde in Kombination mit einer anderen Parallelisierungsstrategie entwickelt (vgl. Abschnitt 4.5.6). An dieser Stelle wird allerdings auf eine Automatisierung verzichtet.

4.3.6.5. Optimierung von *sc_clock* Channels

Taktsignale können in SystemC anhand eines spezialisierten *sc_signal* Channels namens *sc_clock* modelliert werden. Ein *sc_clock* Channel besitzt die Besonderheit, dass der Signalwert zu äquidistanten Zeitpunkten automatisch zwischen 0 und 1 alterniert wird. Der Umschaltvorgang wird durch einen *sc_clock*-internen Prozess gesteuert. Dieser generiert regelmäßig eine Timed Notification, auf die ist er selbst sensitiv ist. Wird er durch eine Timed Notification getriggert, legt er den zukünftigen Signalwert fest, ruft dabei *request_update()* auf und generiert eine Delta Notification. Durch den Aufruf von *request_update()* wird der Signalwert letztlich aktualisiert. Erst durch die Delta Notification wird einem empfangenden Prozess die Existenz der Taktflanke signalisiert und dieser mit einem Deltacycle Verzögerung evaluiert.

Die beschriebene Implementierung hat den Nebeneffekt, dass in den Timedcycles, die durch den *sc_clock* Channel erzeugt werden, u.U. ausschließlich der *sc_clock*-interne Prozess selbst ausgeführt wird. Ein solcher Timedcycle ist daher schlecht bis gar nicht parallelisierbar.

Eine Optimierung besteht deswegen darin, die Taktflanke bereits im Timedcycle per Timed Notification zu signalisieren und den *sc_clock*-internen Prozess gemeinsam mit allen anderen auf den *sc_clock* Channel sensitiven Prozessen zu evaluieren. Der damit erzielbare Performanzgewinn ist umso größer, je geringer die Anzahl an Deltacycles pro Timedcycle ist.

Durch die beschriebene Optimierung ist nicht mehr sichergestellt, dass der im *sc_clock* Channel intern vorhandene Prozess vor den anderen sensitiven Prozessen evaluiert wird. Um dennoch auszuschließen, dass die auf den Channel sensitiven Prozesse einen veralteten Wert des Taktsignals lesen, wird dieser beim Aufruf von *read()* nicht mehr aus der entsprechenden Zustandsvariablen gelesen, sondern direkt aus der aktuellen Simulationszeit abgeleitet.

4.3.7. Bewertung

In den folgenden Abschnitten wird die Leistungsfähigkeit des beschriebenen Parallelisierungsansatzes untersucht und bewertet. Das Augenmerk liegt dabei auf Ausführungsperformanz und Skalierbarkeit auf dem SCC.

Da sich die lokale Pufferung von Zustandsdaten sowie die statische Abbildung von *SC_THREAD* Prozessen als essentiell für einen Performanzgewinn herausgestellt haben, werden sie als fester Bestandteil der Implementierung betrachtet. Die restlichen in Abschnitt 4.3.6 aufgeführten Strategien werden auf ihren zusätzlichen Nutzen hin anhand unterschiedlicher Modelle untersucht.

Für die experimentelle Bewertung wurden zwei verschiedene SystemC Modelle eingesetzt, I) eine synthetische Ringpipeline sowie II) ein detailliertes RTL Modell des HeMPS [75] (siehe Abschnitt 4.2.3.1). Die Rechenkerne des SCC wurden bei 533 MHz, das Mesh bei 800 MHz und der DDR3 Speicher ebenfalls bei 800 MHz betrieben.

4.3.7.1. Synthetisches Szenario

Das Ziel der Simulation mit dem synthetischen Simulationsmodell war es, ein erstes Gefühl für existierende Performanzgrenzen des asymmetrischen synchronen Simulationsansatzes sowie der parallelen Simulation auf dem SCC überhaupt zu bekommen. Die Struktur des synthetischen Pipelinemodells ist in Abb. 4.11 dargestellt.

Das Modell besteht aus zwei Modulen *Component_1* und *Component_2*, die durch eine konfigurierbare Anzahl *m* an Signalen vom Typ *sc_uint<8>* verbunden sind. Jedes dieser beiden Module umfasst wiederum *n* Submodule, die als Pipelinestufen *stage_1 ... stage_n* bezeichnet sind. Jede Stufe besteht aus einem SystemC Prozess vom Typ *SC_METHOD* oder *SC_THREAD*. Das Modell ist vollständig synchron. In jedem Deltacycle führt jede Stufe eine konfigurierbare Anzahl *c* an Fließkommaoperationen aus, liest die Werte der Eingangssignale, inkrementiert diese um eins und schreibt sie in die Ausgangssignale. Aufgrund der vielen existierenden Freiheitsgrade wurde die Untersuchung anhand des synthetischen Modells auf den einfachen Fall eines Masters und zweier Worker beschränkt.

Konstante Anzahl an SystemC Prozessen

Zunächst wurde die Beschleunigung in Abhängigkeit der Modellparameter *m* und *c*, der Prozesstypen *SC_METHOD* und *SC_THREAD* sowie der Kernelparameter *DYNAMIC*, *FIX* und *MANUAL*, welche aktivierten/deaktivierten dynamischen Lastausgleich sowie manuelle Gruppierung bezeichnen, gemessen. Die Abb. 4.12 und 4.13 illustrieren die gemessene Beschleunigung durch Par-

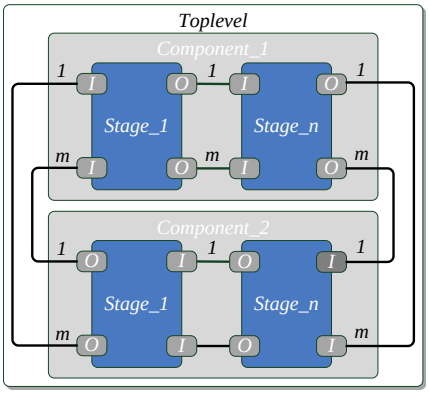


Abbildung 4.11.: Struktur des synthetischen Pipelinemodells

allelisierung im Vergleich zu sequentieller Ausführung auf einem SCC Kern in Abhängigkeit der erwähnten Parameter.

Sowohl für $m = 1$ als auch für $m = 100$ ist ein deutlicher Anstieg der Beschleunigung mit steigender Berechnungskomplexität c in den SystemC Prozessen zu verzeichnen. Für kleine Werte von c ist die parallele Ausführung allerdings signifikant langsamer als die serielle. Die Berechnungskomplexität während der Evaluation Phase ist in diesen Fällen offensichtlich zu klein, um den Kommunikationsaufwand auszugleichen.

Der Einfluss der anderen Parameter ist stark von der Anzahl der Signale m zwischen den Pipelinestufen abhängig. Im Allgemeinen skaliert die Implementierung im Falle einer großen Anzahl an Signalen ($m = 100$) besser. Man würde vermuten, dass sich eine große Anzahl an Signalen wegen häufiger SHM Zugriffe eher negativ auf die Performanz auswirkt. Offensichtlich überwiegen aber sowohl bei statischer als auch auch bei dynamischer Abbildung zusätzlich entstehende parallelisierbare Anteile, die dann in der parallelen Update Phase genutzt werden können.

Deaktivierter dynamischer Lastausgleich (*FIX*) und manuelle Gruppierung (*MANUAL*) führen zu einer zusätzlichen Verbesserung der Beschleunigungswerte. Dies wird speziell bei $m = 100$ deutlich (siehe Abb. 4.13). Die Verschiebung von Teilen des Modellzustands in den privaten Speicherbereich und die Reduktion von SHM Zugriffen erweist sich damit als vorteilhaft. Umgekehrt wird deutlich, dass der dynamische Lastausgleich nicht zu einer besseren Performanz beitragen kann.

Bei Verwendung von *SC_THREAD* Prozessen anstelle von *SC_METHOD* Prozessen liegt die gemessene Beschleunigung generell um einige Prozentpunkte

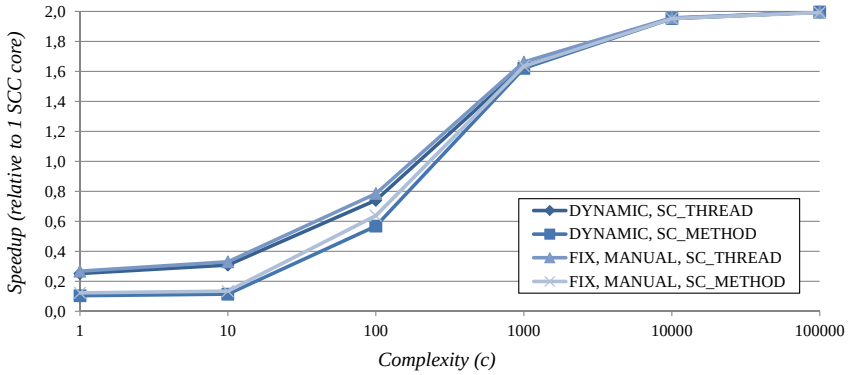


Abbildung 4.12.: Beschleunigung der synthetischen Ringpipeline mit $m = 1$ und $n = 5$

höher, da der Wechsel zum Stack einer *SC_THREAD* Co-Routine mehr Rechenaufwand generiert als der simple Aufruf einer *SC_METHOD* Callback-Funktion. Dieser zusätzliche Rechenaufwand gehört zu den parallelisierbaren Anteilen eines Modells, da er auf den Workern erzeugt wird. Er wirkt sich vor allem für kleine Werte von c aus. Für große Werte von c überwiegt hingegen der Berechnungsaufwand innerhalb der Prozesse, weswegen die Differenz in der Beschleunigung zwischen der *SC_THREAD* und der *SC_METHOD* Variante verschwindet.

Variation der Anzahl an SystemC Prozessen

Schließlich wurde der Einfluss untersucht, den die Anzahl von SystemC Prozessen n auf die Ausführungsperformanz hat. Als feste Modellparameter wurde $m = 10$ und $c = 10$ gesetzt und *SC_METHOD* Prozesse ausgewählt. Dynamischer Lastausgleich wurde deaktiviert, und die Verteilung von Prozessen wurde durch manuelle Gruppierung optimiert. Die Messergebnisse sind in Abb. 4.14 dargestellt.

Mit steigender Prozessanzahl steigt auch der Anteil des parallelisierbaren Rechenaufwands in der Evaluation Phase. Bei der gegebenen Konfiguration ist eine relativ große Anzahl an Prozessen notwendig, um eine Beschleunigung > 0 zu erzielen.

Der Abfall der Beschleunigung für $n > 100$ ist auf den mit wachsendem n steigenden Verwaltungsaufwand im Master und den steigenden Kommunikationsaufwand mit dem Master zurückzuführen: Bei kleinen Werten für c müssen sehr viele Prozesse in kurzen Zeitabständen vom Master geschedult werden. Auch bei statischer Lastverteilung ist der Master für das Scheduling auf die Worker verantwortlich. Die Zuteilung führt zu vielen Zugriffen auf die durch Master

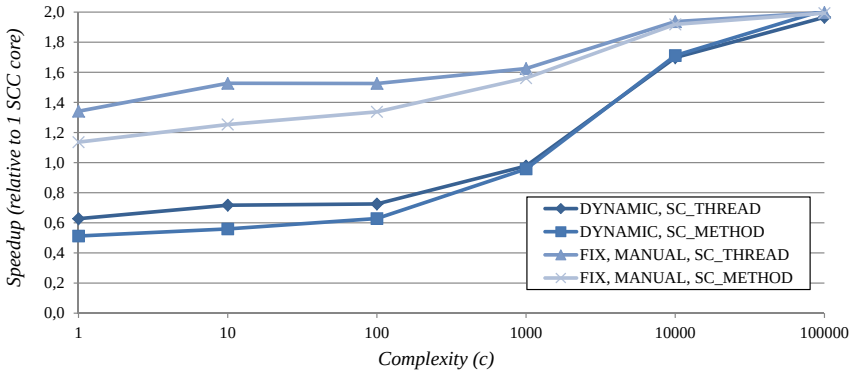


Abbildung 4.13.: Beschleunigung der synthetischen Ringpipeline mit $m = 100$ und $n = 5$

und Worker gemeinsam genutzten Prozess-Warteschlangen. Da jeder Prozess in jedem Deltacycle ausgeführt wird und neue Notifications erzeugt, müssen diese Notifications umgekehrt von den Workern an den Master zur Verarbeitung übermittelt werden.

Die abfallende Beschleunigung für $n > 100$ hängt nicht mit der steigenden Anzahl an Signalen zusammen, die ebenfalls durch die wachsende Anzahl an Pipelinestufen entstehen. Die zusätzlichen Signale werden, aufgrund der manuellen Gruppierung, vollständig im privaten Speicher abgelegt. Sie verursachen deswegen ausschließlich parallelisierbare Anteile. Die Anzahl an Signalen im externen SHM bleibt hingegen konstant.

4.3.7.2. Reales Szenario

Im zweiten Experiment wurde die Ausführungsperformanz anhand einer RTL Beschreibung des HeMPS evaluiert. In den durchgeführten Testläufen wurde auf den Plasmakernen eine Dummy-Applikation ausgeführt: Jeder Plasmakern führt wiederholt eine Anzahl an Ganzzahladditionen in einem Task aus. Anschließend wird eine Nachricht über das simulierte NoC an einen benachbarten Plasmakern verschickt.

Um eine Beschleunigung zu erzielen, mussten für die Messungen sämtliche Optimierungen angewendet werden, die in den vorangegangenen Abschnitten beschrieben wurden. Die Prozesse der einzelnen Plasma-Prozessoren wurden jeweils zu einer Gruppe zusammengefasst. Das Hermes NoC mit den Routern wurde in zwei Prozessgruppen zusammengefasst. Die die Gruppen verbindenden Signale wurden in den externen SHM abgebildet. Die Beschleunigungswer-

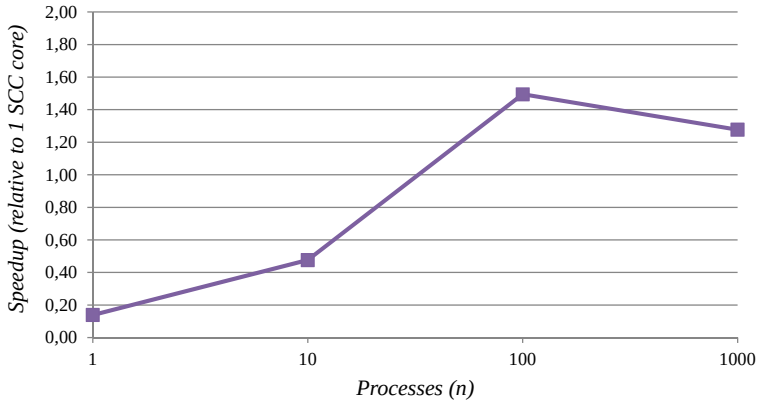


Abbildung 4.14.: Beschleunigung der synthetischen Ringpipeline in Abhängigkeit der Anzahl an SystemC Prozessen

te, die bei der Ausführung eines 2x2 und eines 4x4 HeMPS Modells auf bis zu 16 Workern gemessen wurden, sind in Abb. 4.15 zu sehen.

Beim 2x2 Modell ist durch Parallelisierung keine Beschleunigung > 1 möglich. Beim 4x4 Modell wird mit sechs Workern eine maximale Beschleunigung von 27% erreicht. Die Skalierbarkeit des Simulators auf eine größere Anzahl an SCC Kernen ist deutlich limitiert: Die Beschleunigung beider Modelle stagniert zwischen sechs und acht Workern und nimmt danach ab.

Die schlechte Skalierbarkeit kann zum einen auf den Schedulingaufwand im Master, den Kommunikationsaufwand mit dem Master und den verhältnismäßig geringen Berechnungsaufwand in jedem Worker zurückgeführt werden. Bei stärkerer Parallelisierung kommt der Master der schnellen Parallelverarbeitung in den Workern nicht mehr hinterher. Zum anderen wird mit steigender Anzahl an Workern die globale Barriersynchronisation immer kostspieliger. Schließlich ist die Beschleunigung durch die unterschiedlich großen Prozessgruppen und die damit einhergehende ungleichmäßige statische Lastverteilung limitiert.

4.3.7.3. Diskussion

Die beschriebene Strategie wurde insbesondere unter der Prämisse implementiert, eine einfache dynamische Lastverteilung von SystemC Prozessen auf Kernelkomponenten zu ermöglichen. Dies hat sich auf cachekohärenten SHM Multiprozessoren in Verbindung mit einem barrierbasierten Master/Worker Schema als nützlich erwiesen. Daher wurde in dieser Arbeit ein zu bekannten Ansätzen [232, 105] ähnliches Programmiermodell umgesetzt, das stark auf die Nutzung

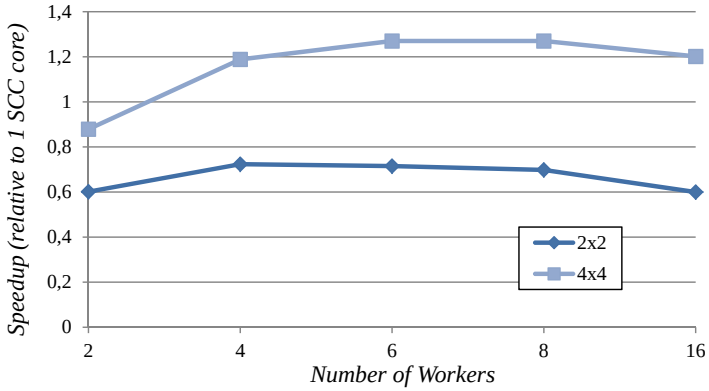


Abbildung 4.15.: Beschleunigung der HeMPS Modells in Abhängigkeit von Modellgröße und Anzahl an Workern

von Shared Memory ausgelegt ist. Dieses hat sich auf dem SCC allerdings aus folgenden Gründen als ungünstig herausgestellt:

1. Mangels hardwareseitiger Cachekohärenz existieren hohe Zugriffslatenzen auf den externen SHM. Daher ist es schwer möglich, den Kommunikationsaufwand durch den Berechnungsaufwand auszugleichen. Die dynamische Lastverteilung ist dadurch nahezu wirkungslos.
2. Die zentralisierte Softwarearchitektur hat einen nicht zu vernachlässigenden Kommunikations- und Berechnungsaufwand im Master zur Folge, der mit steigender Kernanzahl sehr schnell zum Flaschenhals wird.
3. Die globale Barriersynchronisation in Verbindung mit ungleicher statischer Lastverteilung limitiert die Parallelität.

Wie sich bei den experimentellen Untersuchungen herausgestellt hat, sind eine Vielzahl an Optimierungen notwendig, um mit dem beschriebenen Ansatz eine messbare Beschleunigung der parallelen Simulation gegenüber der sequentiellen Simulation zu erzielen. Aus Sicht des Modells hat vor allem die Anzahl an SystemC Prozessen, deren Rechenkomplexität und die damit eng verbundene Zugriffsfrequenz auf Daten, die sich in gemeinsam genutzten Speicherbereichen befinden, großen Einfluss auf die Effizienz.

Durch geeignete modellspezifische Optimierungen wie Gruppierung von SystemC Prozessen, Einschränkung der Verfügbarkeit und Verschiebung von Daten in den privaten Speicher konnte der Umfang an gemeinsam genutzten Daten und die Zugriffsfrequenz auf diese Daten reduziert werden. Im folgenden Abschnitt soll daher untersucht werden, inwieweit ein vollständiger Verzicht auf

globale Verfügbarkeit in Verbindung mit einer dezentralen Softwarearchitektur zu besserer Effizienz beitragen kann.

4.4. Symmetrische asynchrone Strategie

In der Untersuchung im vorigen Abschnitt hat sich herausgestellt, dass sich die asymmetrische Master/Worker Architektur in Verbindung mit der vollständig synchronen Ausführung auf dem SCC schnell zu einem Flaschenhals entwickeln kann. Die Anwendung dynamischer Lastverteilung konnte nicht signifikant zu einem Ausgleich des Kommunikations- und Synchronisationsoverheads beitragen. Vielmehr hat sich die Einschränkung der Datenverfügbarkeit in Verbindung mit der statischen Abbildung des Simulationsmodells als eine Maßnahme mit Potential zur Steigerung der Effizienz erwiesen.

Der in diesem Abschnitt beschriebene Ansatz zielt daher in erster Linie auf die Erhöhung der Datenlokalität und die Vermeidung zentralisierter Kommunikations- und Synchronisationsstrukturen ab. Dazu wird ein auf dem asynchronen Null Message Algorithmus (vgl. Abschnitt 2.2.3.4) basierendes Verfahren entwickelt. Es werden Bedingungen hergeleitet, die zur Ausführung von RTL ähnlichen SystemC Modellen mit Hilfe des NMA erfüllt sein müssen. Anschließend wird anhand einer prototypischen Implementierung die Leistungsfähigkeit bewertet. Der Ansatz wird durch eine teilautomatisierte Werkzeugkette ergänzt. Teile dieses Abschnitts sind bereits in [RRE+12] publiziert. Die Werkzeugkette wurde im Rahmen einer vom Autor betreuten Studienarbeit [Erd12] umgesetzt und ist in Zusammenarbeit entstanden.

4.4.1. Anforderungen und Konzept

Folgende Anforderungen liegen der beschriebenen Strategie zugrunde:

I) **Modellierungsmethode**: Es soll zumindest das RTL Subset von SystemC [244] unterstützt werden. Das Modell soll prinzipiell als ein Prozess-Signal Graph G_{PS} beschreibbar sein (vgl. Definition 4.2).

II) **Abbildung des Simulationsmodells**: Eine statische Abbildung des Simulationsmodells auf die Ausführungsplattform ist ausreichend.

III) **Datenpartitionierung**: Gemeinsam genutzte Datenstrukturen sollen so weit wie möglich in intern und lokal extern partitioniert werden.

IV) **Kernelpartitionierung**: Es soll eine symmetrische Kernelpartitionierung in n identische Kernelkomponenten k^s erfolgen, die jeweils alle Kernelphasen implementieren:

$$\forall i \in 1...n : P(k_i^s) = \{init_i(), eval_i(), update_i(), \\ dnotify_i(), tnotify_i()\} \tag{4.6}$$

V) **Synchronisation**: Globale Synchronisation soll strikt vermieden werden.

Abb. 4.16 illustriert das resultierende Konzept. Wegen Anforderung I genügt es, IRA korrekt aufzulösen. Wegen der Anforderungen II und III können bereits in der Konzeptionsphase des parallelen Simulators weitergehende Optimierungen (vgl. Abschnitt 4.2.4.6) zur Erhöhung der Datenlokalität eingeplant werden. Beispielsweise kann statisch bestimmt werden, welche Variablen welche Kernelkomponenten miteinander „verbinden“. Dies ermöglicht unmittelbar die Ableitung der Kommunikationstopologie zwischen den Kernelkomponenten und die Identifikation interner und lokal externer Variablen vor der Laufzeit (siehe Abb. 4.16). Wegen Anforderung IV implementiert jede Kernelkomponente alle Kernelphasen. Bis auf Anforderung V existieren keine Einschränkungen für die Wahl des Synchronisationsverfahrens zwischen den Kernelkomponenten.

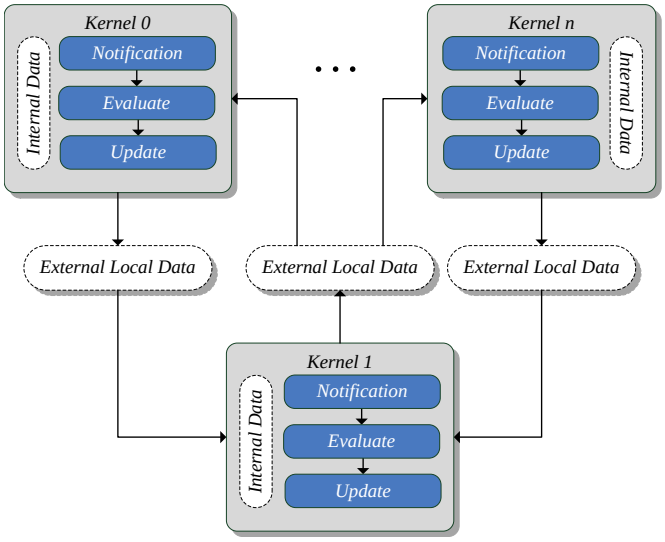


Abbildung 4.16.: Architekturkonzept der symmetrischen synchronen Strategie

4.4.2. Datenpartitionierung auf Kernelebene

Wie beim asymmetrischen Verfahren wird für die Umsetzung der Kernelkomponenten jeweils ein vollständiger sequentieller Kernel verwendet, so dass auch hier bereits interne Duplikate von U, N^{τ}, N^{δ}, R, τ und δ existieren. Da Kernelkomponenten generell nur mit Nachbarn kommunizieren, genügt für die betroffenen Variablen lokal externe Verfügbarkeit. Für die Realisierung von Kommunikation und Synchronisation eignet sich ein nachrichtenbasierter Mechanismus. Zur Umsetzung des nachrichtenbasierten Mechanismus wird im Gegensatz zur asymmetrischen Strategie eine separate logische Ebene verwendet, mit der topologieabhängige Funktionalität verwaltet werden kann. Im Fall von τ wird als Alternative direkt gemeinsam genutzter Speicher verwendet (siehe Abschnitt 4.4.3.2).

Im Modell können bis auf eine Teilmenge der Signale, welche lokal extern verfügbar sein müssen (vgl. Definition 4.4), alle Datenstrukturen als intern deklariert werden. Um möglichst viel Spielraum für ein bestimmtes Synchronisationsverfahren zu lassen, kann für ein lokal externes Signal ein vollständiges internes Duplikat in jeder Kernelkomponente vorgehalten werden, welche auf das Signal Zugriff hat.

4.4.3. Logische Ebene

Um globale Synchronisation zu vermeiden wird ein asynchrones PDES Verfahrens ähnlich dem NMA verwendet. Bei asynchroner PDES wird die Gesamtsimulation nicht gezielt in einen global gültigen Zustand überführt (vgl. Abschnitt 2.2.3.4). Es gilt vielmehr, globale Wartezustände soweit als möglich zu vermeiden. Typischerweise ist eine asynchrone PDES von der Topologie eines Modells abhängig. Daher ist es sinnvoll, eine zusätzliche logische Ebene einzuführen, die die topologieabhängige Funktionalität verwaltet.

Die logische Ebene besteht aus logischen Prozessen und logischen Links. Ein logischer Prozess lp_i kapselt genau eine Kernelkomponente k_i^s (vgl. Abschnitt 4.2.3.2). Die SystemC Prozesse und internen Signale, die auf k_i^s bzw. lp_i abgebildet werden, sind daher identisch. Ein logischer Link l_{ij} von lp_i nach lp_j existiert genau dann, wenn ein auf lp_i abgebildeter SystemC Prozess auf ein externes Signal schreibt, das von einem auf lp_j abgebildeten SystemC Prozess gelesen wird. Das Netzwerk auf logischer Ebene kann insgesamt als ein **Logischer-Prozess Graph** $G_{LP}(LP, L)$ modelliert werden:

Definition 4.5 *(**Logischer-Prozess Graph**): Ein **Logischer-Prozess Graph** $G_{LP}(LP, L)$ ist ein gerichteter Graph mit Knoten $lp \in LP$ und Kanten $l \in L$. Jeder Knoten repräsentiert genau einen **logischen Prozess** und jede Kante einen gerichteten*

logischen Link. *Zwei Knoten lp_i und lp_j sind durch eine Kante l_{ij} verbunden, wenn ein Nachrichtenaustausch von lp_i in Richtung lp_j möglich ist. lp_i wird dann als* ***adjazent zu*** *lp_j bezeichnet.*

4.4.3.1. Nachrichtenbasierte Kommunikation über logische Links

Da Prozesse in RTL Modellen nur über Signale kommunizieren, lassen sich alle extern ausgelösten Änderungen am internen Zustand einer Kernelkomponente auf Änderungen am Zustand externer Signale zurückführen. In einer parallelen SystemC RTL Simulation dient ein logischer Link daher ausschließlich zur Übertragung von **Signalnachrichten** für alle auf ihn abgebildeten externen Signale:

Definition 4.6 (***Signalnachricht***): *Eine Signalnachricht, die von lp_i an lp_j über l_{ij} übertragen wird, enthält folgende Informationen:*

- *Eine id des zugehörigen externen SystemC Signals s,*
- *eine Variable v^{msg}, welche den Wert der v^{next} Variablen des Signals übermittelt,*
- *einen Zeitstempel τ_i^{msg} des Sendezeitpunkts in lp_i.*

Mit der *id* können Nachrichten einem bestimmten externen Signal zugeordnet werden. Mit der v^{msg} Variable wird die neue Zustandsinformation eines Signals in Form des nächsten gültigen Signalwerts v^{next} übermittelt. Der Zeitstempel τ_i^{msg} dient zur Ableitung des Zeitpunktes der Aktualisierung des sichtbaren Signalzustandes (v^{cur}) im Empfänger.

Für eine asynchrone PDES auf Basis von Signalnachrichten muss ein logischer Link das FIFO Prinzip auf der gesamten Kommunikationsstrecke zwischen zwei logischen Prozessen implementieren (vgl. Abschnitt 2.2.3.4). Vorausgesetzt, die Basisdienstebene garantiert bereits eine FIFO-basierte Übertragung zwischen Prozessorkernen, so sind die Hauptaufgaben der logischen Ebene bzgl. Kommunikation:

1. Zuordnung von ausgehenden Signalnachrichten zu logischen Links,
2. Zuordnung von auf logischen Links eingehenden Signalnachrichten zu Signalen und deren Zwischenpufferung entsprechend dem umgesetzten Synchronisationsverfahren.

4.4.3.2. Basisverfahren zur Synchronisation

Das Basisverfahren zur Synchronisation auf logischer Ebene ist eine abgewandelte Form des NMA. Es basiert (im Unterschied zum SystemC Kernel) auf ei-

nem einfachen skalaren Zeitmodell. Logische Prozesse besitzen lokale Zeiten τ_i. Jedem logischen Link l_{ij} wird eine interne Variable genannt **Linkzeit** τ_{ij}^{link} und ein **Lookahead** Δl_{ij}^{τ} zugeordnet:

Definition 4.7 *(**Linkzeit**): Die **Linkzeit** τ_{ij}^{link} eines logischen Links l_{ij} von lp_i nach lp_j entspricht der letzten in lp_j bekannten lokalen Zeit τ_i von lp_i.*

Zur Ableitung der Linkzeit kann der Zeitstempel τ^{msg} von Signalnachrichten verwendet. Unter der Voraussetzung, dass Nachrichten in der Reihenfolge ihrer Zeitstempel übertragen werden, gibt der Zeitstempel den frühesten Zeitpunkt an, an dem in Zukunft eine Nachricht von lp_i verschickt werden kann. Beim Empfang in lp_j entspricht der Zeitstempel τ_i^{msg} dem letzten bekannten Wert der Linkzeit τ_{ij}^{link}.

Der Lookahead eines logischen Links l_{ij} spezifiziert dessen zeitliche Verzögerung und ist eine Konstante. Entsprechend dem NMA [78] darf ein logischer Prozess immer dann in der Zeit voranschreiten, wenn seine lokale Zeit kleiner ist als das Minimum aus der Summe von Linkzeiten und Lookaheads $\Delta\tau$ aller eingehenden logischen Links. Diese Bedingung für den Zeitfortschritt lässt sich im Kontext von SystemC anhand einer spezialisierten Fassung der lokalen Kausalitätsbedingung aus Definition 2.2 wie folgt schreiben:

Definition 4.8 *(**LOCC: Local Causality Condition**): Ein logischer Prozess lp_j darf die nächste Notification bei τ_j^{next} verarbeiten, wenn*

$$\forall lp_i \in LP_j^{adj} : \tau_j^{next} \leq \tau_i^{link} + \Delta l_{ij}^{\tau}. \tag{4.7}$$

Für den maximal möglichen Zeitfortschritt gilt:

$$\tau_j^{max} = \min_{\forall i}(\tau_i^{link} + \Delta l_{ij}^{\tau}) \tag{4.8}$$

Damit die Linkzeit τ_{ij}^{link} auch ohne Nachrichten von lp_i abgeleitet werden kann und Deadlocks vermieden werden, muss lp_i regelmäßig seine lokale Zeit τ_i verfügbar machen. Im NMA werden dazu Null Messages verwendet. In dieser Arbeit wird ein alternativer Ansatz gewählt: Es wird eine lokal externe Variable τ_i^{link} deklariert, die für alle adjazenten logischen Prozesse zugänglich ist. lp_i muss den Wert von τ_i dann regelmäßig in τ_i^{link} schreiben. τ_i^{link} muss von lp_i immer dann auf τ_i aktualisiert werden, wenn sicher ist, dass (trotz evtl. interner Nachrichtenpufferung) tatsächlich keine Nachrichten mit einem kleineren Zeitstempel als τ_i verschickt werden können.

Voraussetzung für die beschriebene Methode ist, dass die Zielplattform die Reihenfolge von Speicherzugriffen eines einzelnen Prozessors auf beliebige gemeinsam genutzte Bereiche erhält. Dies ist beim SCC und gewöhnlichen cachekohärenten Architekturen gegeben.

4.4.3.3. Deadlocks durch Deltacycles

Durch die Anforderung, dass Kommunikation nur mit benachbarten Prozessen stattfinden sein soll, entsteht die Einschränkung, dass keine Zyklen aus logischen Links mit einem Zero Lookahead (vgl. Abschnitt 2.2.3.4) von $\Delta l^{\tau} = 0$ existieren dürfen, da diese zu Deadlocks führen [78]. Dies sei an folgendem Beispiel verdeutlicht:

Eine parallele Simulation bestehe aus drei logischen Prozessen lp_0 bis lp_2, die entsprechend Abb. 4.17 über logische Links miteinander in einem Zyklus verbunden sind. Jeder logische Link habe einen Lookahead von $\Delta l^{\tau} = 0$. Ein Zeitfortschritt $\Delta\tau > 0$ kann nur durchgeführt werden, wenn alle eingezeichneten Ungleichungen gleichzeitig erfüllt sind, was niemals möglich ist.

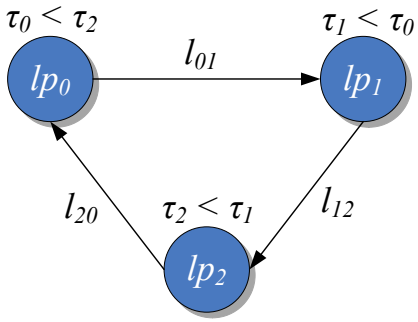


Abbildung 4.17.: Entstehung von Deadlocks beim Null Message Algorithmus

Das gleiche Problem tritt auch auf, wenn man den NMA auf eine parallele SystemC Simulation mit vektoriellem Zeitmodell (vgl. Abschnitt 2.3.2.2) anwenden möchte. Durch die Verzögerung eines Signals von nur einem Deltacycle existiert bzgl. τ generell ein Lookahead von $\Delta l^{\tau} = 0$. Zirkuläre Abhängigkeiten im Netzwerk logischer Prozesse führen dann dazu, dass die Simulation nicht in Timedcycles voranschreiten kann. Darüber hinaus geht die partielle Ordnung von Notifications während eines Timedcycles verloren. Ohne geeignete Maßnahmen ist der NMA daher nicht auf SystemC RTL Modelle u.ä. anwendbar.

4.4.3.4. Elimination zirkulärer Abhängigkeiten

Der gewählte Lösungsansatz basiert auf der kontrollierten Relaxation der Synchronität zwischen adjazenten logischen Prozessen von der Ebene der Deltacycles auf die Ebene der Timedcycles. Im Folgenden wird gezeigt, dass durch Elimination von zirkulären Abhängigkeiten mit $\Delta l^{\tau} = 0$ auf Basis gezielter Latenzannotationen dennoch Zyklengenauigkeit einer NMA basierten parallelen RTL Simulation erreicht werden kann. Das Verfahren setzt sich aus zwei Schritten zusammen:

1. Klassifikation von logischen Links hinsichtlich einer sog. Kritikalitätseigenschaft.
2. Gezielte Annotation zeitlicher Verzögerungen und Einschränkung der Abbildung des Simulationsmodells.

Klassifikation logischer Links

Im Folgenden wird angenommen, dass logische Links in *deadlock-kritisch* und *deadlock-unkritisch* klassifizierbar sind und nur *kritische Zyklen* bestehend aus deadlock-kritischen logischen Links zum Deadlock führen. In diesem Fall können kritische Zyklen durch geeignete Abbildung eines Simulationsmodells ausgeschlossen werden. Abb. 4.18 illustriert dazu ein Beispiel.

Die linke Seite von Abb. 4.18 zeigt einen logischen Prozessgraphen $G_{LP}(LP, L)$. O.B.d.A. sind logische Prozesse $lp \in LP$ über logische Links $l \in L$ in einer Mesh-Topologie verbunden. Wenn keine Klassifikation logischer Links vorgenommen wird, existieren zirkuläre Abhängigkeiten mit einem Lookahead von $\Delta l^{\tau} = 0$ zwischen jedem Paar logischer Prozesse. Genauer gesagt: Der gesamte Graph G_{LP} ist stark zusammenhängend [152]. Die logischen Prozesse können folglich niemals in Timedcycles voranschreiten.

Ist es hingegen möglich, logische links in kritisch (c) und nicht-kritisch (nc) zu klassifizieren, so können zirkuläre Abhängigkeiten u.U. eliminiert werden, da diese nur noch für kritische Links berücksichtigt werden müssen. Dies wird auf der rechten Seite von Abb. 4.18 deutlich, in der nur noch der Graph $G_{LP}^{crit}(LP, L^c)$, ein Teilgraph von G_{LP} mit $L^c \subseteq L$, dargestellt ist.

Im Fall von SystemC RTL bündelt ein logischer Link ausschließlich Signale. Daher muss die Kritikalitätseigenschaft eines logischen Links aus den Eigenschaften der Signale abgeleitet werden, die auf ihn abgebildet sind. Beispielsweise gilt:

Definition 4.9 *(**LDP: Link Delay Property**): Der Lookahead δl_{ij}^{τ} eines logischen Links l_{ij} entspricht der minimalen Verzögerung Δs_{ij}^{min} aller SystemC Signale $s \in S_{ij}$, die l_{ij}*

bündelt. Der logische Link l_{ij} wird als ***delta-verzögert*** *bezeichnet, wenn $\Delta l_{ij}^{\tau} = 0$. Er wird als* ***zeitverzögert*** *bezeichnet, wenn $\Delta l_{ij}^{\tau} > 0$.*

Mit Definition 4.9 folgt für die Kritikalität eines logischen Links:

Definition 4.10 *(**LCC1: First Link Criticality Condition**): Ein logischer Link ist deadlock-kritisch, wenn er delta-verzögert ist. Ansonsten ist er deadlock-unkritisch.*

Um kritische Zyklen zu vermeiden, muss ein Modell derart auf logische Prozesse abgebildet werden, dass keine delta-verzögerten und damit kritischen logischen Links entstehen, die Zyklen in G_{LP}^{crit} erzeugen. Da ein SystemC Signal normalerweise aber immer delta-verzögert ist, existieren nach Definition 4.10 weiterhin nur kritische logische Links. Um dieses Dilemma zu lösen, werden Signale in einem zweiten Schritt gezielt mit zeitlichen Verzögerungen $\Delta s^{\tau} > 0$ annotiert.

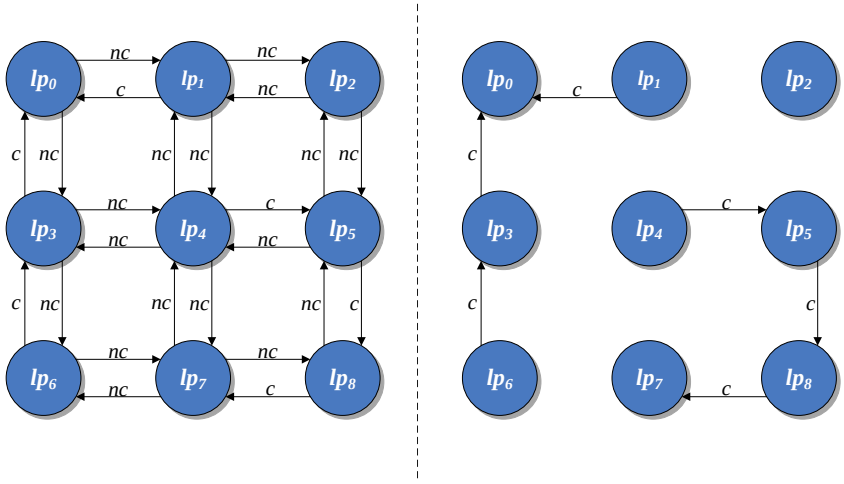


Abbildung 4.18.: G_{LP} (links) und G_{LP}^{crit} (rechts) ohne zirkuläre Abhängigkeiten zwischen deadlock-kritischen logischen Links

Annotation von Verzögerungen und Modellabbildung

Grundsätzlich erfolgt eine Annotation von Verzögerungen nur an externe Signale. Externe Signale sind daran zu erkennen, dass sie im Verlauf der Partitionierung (siehe Definition 4.4) auf logische Links abgebildet werden. Zyklengenaue Simulation einer Partitionierung ist möglich, wenn nach der Annotation folgende Bedingungen erfüllt sind:

- **Bedingung I**: Wenn ein externes Signal Teil einer Menge von externen Signalen ist, die sich im gleichen kombinatorischen Pfad befinden, so muss die Summe der annotierten Verzögerungen $\sum \Delta s_i^\tau$ aller externen Signale innerhalb dieses Pfades kleiner als die minimale Aktivierungsperiode $\Delta\tau^{period}$ des Modells[9] sein.
- **Bedingung II**: Kombinatorische SystemC Prozesse[10] müssen immer eine vollständige Sensitivitätsliste besitzen.
- **Bedingung III**: Zwischen logischen Prozessen dürfen keine kritischen Zyklen existieren.

Bedingung I ist für den Erhalt der Zyklengenauigkeit notwendig. Sie garantiert, dass kombinatorische Prozesse in jedem Fall rechtzeitig vor der nächsten Taktflanke einen stabilen Endzustand erreichen. Die Erfüllung von Bedingung I schließt somit aus, dass Annotationen nicht zu einer Verzögerung bis nach der nächsten Taktflanke führen. Die partielle Ordnung, die durch die Deltacycles definiert ist, bleibt dabei nur insoweit erhalten, wie es für zyklengenaue Simulation notwendig ist.

Bedingung II garantiert, dass neue Signalwerte nicht gespeichert werden, bevor deren Änderung zu einem späteren Zeitpunkt von einem lesenden Prozess registriert wird. Durch unvollständige Sensitivitätslisten gespeicherte Werte sind im Allg. von der durch die Deltacycles definierten partiellen Ordnung von Prozessaktivierungen abhängig. Ändert sich diese, so ändern sich die gespeicherten Werte und damit u.U. der Endzustand der Signale, der im nächsten Taktzyklus gelesen wird.

Generell würden bei einer Logiksynthese von Modellen mit unvollständiger Sensitivitätsliste sog. Latches entstehen. Latches werden nur in wenigen Spezialfällen benötigt und in synchronen Schaltwerken normalerweise gemieden, da sie in der späteren Implementierung meist zu Glitches und Timingfehlern führen. Prozesse mit unvollständiger Sensitivitätsliste können daher ausgeschlossen werden.

Bedingung III ist schließlich notwendig, um Deadlocks zu verhindern. Kritische Zyklen sind in jedem Fall ausgeschlossen, wenn alle vorhandenen externen Signale mit Werten $\Delta s^\tau > 0$ annotiert werden können.

Sind alle Bedingungen erfüllt, kann das partitionierte und annotierte RTL Modell unmittelbar mit dem NMA basierten Verfahren aus Abschnitt 4.4.3.2 parallel und zyklengenau ausgeführt werden. Dabei ist zu beachten, dass Taktsignale grundsätzlich nicht verteilt werden können. Eine Propagation des Taktsignals durch

[9] Die minimale Aktivierungsperiode $\Delta\tau^{period}$ entspricht üblicherweise einem oder einem halben Taktzyklus.

[10] Kombinatorische Prozesse sind SystemC Prozesse, die (auch) asynchron durch beliebige Signale und nicht nur durch das Taktsignal aktiviert werden können.

mehrere logische Prozesse würde dazu führen, dass jeder vom Taktsignal durchlaufene logische Prozess dieses um den Betrag des Lookahead verzögert. Dieses Problem kann dadurch umgangen werden, dass jede Modellpartition mit einem separaten SystemC Prozess für die Taktgenerierung ausgestattet wird.

4.4.4. Integration nachrichtenbasierter Kommunikation

Zur Integration der Abschnitt 4.4.3.1 beschriebenen nachrichtenbasierten Kommunikation mit dem SystemC Kernel wird ein neuer Channeltyp namens *PDESSignal* eingeführt. Eingehende und ausgehende logische Links werden innerhalb der logischen Prozesse mit Hilfe sog. Sockets repräsentiert. Aktualisierungen ausgehender externer Signale des Typs *PDESSignal* werden in Signalnachrichten verpackt und mit Aufruf der *write()* Methode direkt an das zugehörige Ausgangssocket übermittelt. Eingehende Signalnachrichten werden von einem Eingangssocket an einen *InputAdaptor* weitergeleitet. Für jedes externe Eingangssignal existiert ein separater Adapter im empfangenden logischen Prozess. Dessen Aufgabe ist nicht nur die reine Zwischenpufferung. Vielmehr dient er zur kausal korrekten Integration eingehender Kommunikation in den SystemC Kernel.

In Abb. 4.19 ist die Implementierung eines Adapters für einen logischen Prozess lp_i dargestellt. Nachrichten werden durch Aufruf von *insert_message()* vom Eingangssocket gelesen. Die Behandlung der Signalnachricht hängt dann vom Zeitstempel der Nachricht und der lokalen Zeit des empfangenden logischen Prozesses lp_i ab:

1. $\tau^{msg} + \Delta s^{\tau} = \tau_i$: In diesem Fall erfolgt die Aktualisierung des zugehörigen Signals durch Aufruf von *imm()* und Erzeugung einer Immediate Notification direkt.
2. $\tau^{msg} + \Delta s^{\tau} > \tau_i$: Dieser Fall erfordert eine Verzögerung der Aktualisierung bzgl. τ. Dies erfolgt mit Hilfe der Methode *timed()* und einem *Update FIFO*. Falls der FIFO bei Aufruf von *timed()* leer ist, so wird zum Zeitstempel τ^{msg} einer Nachricht die Verzögerung Δs^{τ} des zugehörigen Signals addiert und auf ω eine Timed Notification registriert. Anschließend wird die Nachricht in den *Update FIFO* geschrieben. Zusätzlich zu den normalen SystemC Prozessen im Modell existiert innerhalb des Adapters ein Prozess namens *update_process()*, welcher auf Notifications von ω sensitiv ist. Sobald $\tau_i = \tau^{msg} + \Delta s^{\tau}$ erreicht ist, existieren zwei Möglichkeiten:
   a) Ein lesender Prozess im Modell wird bei τ_i vor *update_process()* ausgeführt: In diesem Fall wird die Aktualisierung durch Aufruf von *update_process()* aus der *read()* Methode heraus und Einlesen des Wertes der obersten Signalnachricht im FIFO durchgeführt.

b) *update_process()* wird bei τ_i vor allen lesenden Prozessen im Modell ausgeführt: In diesem Fall erfolgt die Aktualisierung durch den Aufruf von *update_process()* vom SystemC Kernel.

Nach der Aktualisierung wird innerhalb von *update_process()* die oberste Signalnachricht im *Update FIFO* gelöscht. Falls der FIFO nicht leer ist, wird der Zeitstempel der nächsten Signalnachricht zur Registrierung einer neuen Timed Notification auf ω verwendet.

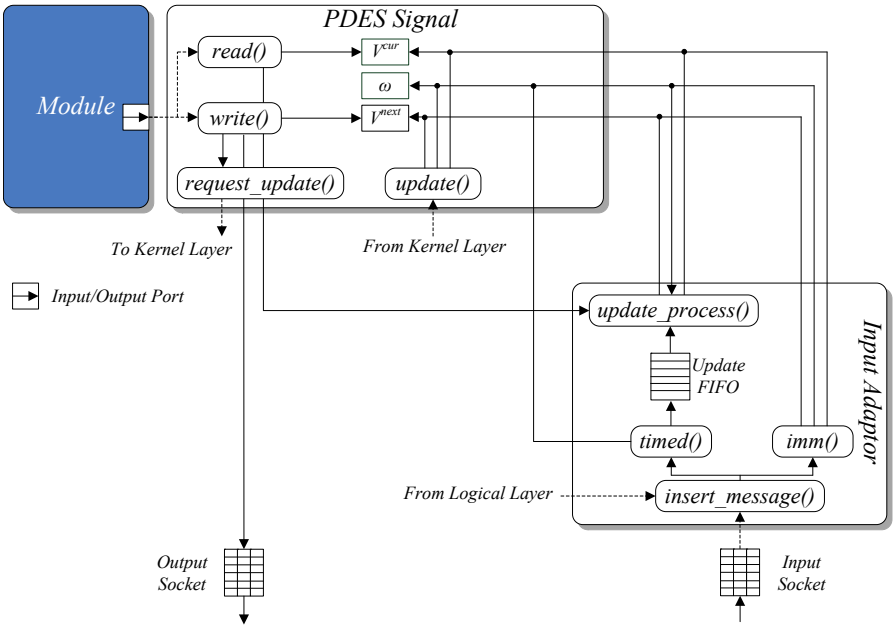


Abbildung 4.19.: Integration nachrichtenbasierter Kommunikation

4.4.5. Integration des Synchronisationsverfahrens

Zur detaillierten Beschreibung des Verfahrens werden zunächst folgende zusätzliche Variablen eingeführt:

- τ^{next}: Speichert den (skalaren) Zeitstempel der nächsten lokal vorhandenen Timed Notification / des nächsten lokal vorhandenen Timeouts.
- τ^{bound}: Speichert den den Zeitstempel, der entsprechend Ausdruck 4.8 den nächsten maximal möglichen Zeitfortschritt spezifiziert.

Die Integration der Synchronisationsverfahrens mit einer Kernelkomponente k_i^s in einen logischen Prozess lp_i kann anhand folgender Aktionen beschrieben werden:

- *checkLOCC*(τ^{next}): Prüfe, ob für lp_i ein Zeitfortschritt entsprechend der LOC Bedingung aus Definition 4.8 möglich ist. Gib anschließend die maximale Zeitgrenze τ^{bound} entsprechend Ausdruck 4.8 zurück, bis zu der ohne Verletzung kausaler Abhängigkeiten ein Zeitfortschritt durchgeführt werden kann.
- *updateT()*: Aktualisiere die τ_i^{link} Variable. Prüfe dabei, ob sich noch Nachrichten in lokalen Ausgangspuffern der Sockets befinden (falls beim letzten Aufruf von *dispatch()* nicht alle Nachrichten verschickt werden konnten). Wenn nein, setze τ_i^{link} auf τ_i. Wenn ja, setze τ_i^{link} auf den Zeitstempel der zuletzt versendeten Nachricht.

Folgende Aktion modelliert den Zugriff auf logische Links:

- *dispatch()*: Leite von der lokalen Simulation generierte und evtl. noch im lokalen Senderpuffer (vgl. Anhang A.2) zwischengespeicherte Nachrichten an die logischen Prozesse weiter, für die die Nachrichten bestimmt sind. Lese dann alle aktuell in den Eingangssockets verfügbaren Nachrichten aus. Identifiziere das zu einer Signalnachricht gehörige Signal und füge den empfangenen neuen Signalwert v^{msg} durch Aufruf der *insert_message()* Methode auf dem zugehörigen Input Adaptor in die Simulation ein. Setze die Linkzeit der durch die Eingangssockets repräsentierten eingehenden logischen Links auf den Zeitstempel der Nachricht, die zuletzt von einem zu ihm gehörigen Eingangssocket gelesen wurde. Falls auf einem Link keine Nachricht empfangen wurde, so verwende die lokal externe τ^{link} Variable des zugehörigen benachbarten logischen Prozesses zur Ableitung der Linkzeit.

Abb. 4.20 illustriert den Zustandautomaten. Die zeitliche Synchronisation mit anderen logischen Prozessen in $s^{checkLOCC}$ kann wegen der Annotation von Verzögerungen bzgl. τ an Signale mit dem Empfang von Aktualisierungen verbunden sein kann (Rückwärtspfad von $s^{checkLOCC}$ nach $s^{dispatch}$). Wenn diese unmittelbar lauffähige Prozesse erzeugen ($|R| \neq 0$ im Rückwärtspfad von $s^{dispatch}$ nach s^{eval}) wird sofort ein weiterer Deltacycle durchgeführt. Dadurch ist vollständiger Empfang aller Nachrichten eines Deltacycles nicht garantiert und es wird eine nicht-deterministische Anzahl an Deltacycles per Timedcycle generiert. Da das RTL Subset vorausgesetzt wird und unvollständige Sensitivitätslisten ausgeschlossen sind, bleibt die Zyklengenauigkeit dennoch erhalten.

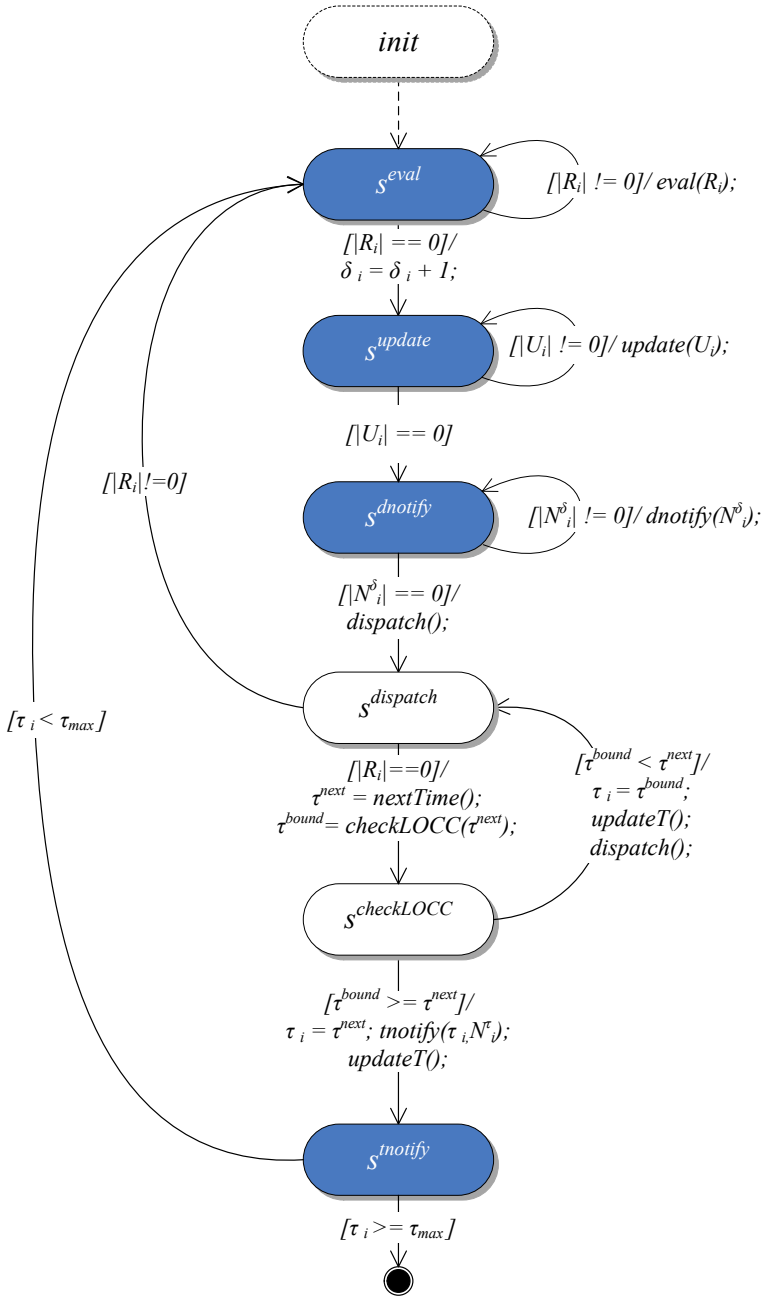


Abbildung 4.20.: Asynchrone Zustandsmaschine in einer Kernelkomponente k_i^s

4.4.6. Manuelle Partitionierung des Simulationsmodells

Die Partitionierung des Simulationsmodells muss manuell spezifiziert werden. Dies beinhaltet die I) Zuweisung von SystemC Prozessen zu Partitionen und II) die Deklaration von Signalen als interne und externe Signale.

Zur Umsetzung von Punkt I) wird eine neue Wrapper-Basisklasse zur Verfügung gestellt. Diese erlaubt eine Partitionierung des Modells auf dem Toplevel mit einer Granularität von Modulen. Für jeden Modultyp auf dem Toplevel muss durch Ableitung von der neuen Basisklasse ein Wrapper entwickelt werden. Instanzen der Module müssen dann durch Instanzen der spezialisierten Wrapper ausgetauscht werden. Ein Wrapper erlaubt die bedingte Instanziierung des gekapselten Moduls anhand eines Flags. Auf diese Art und Weise kann ein Modell grundsätzlich mit Modulgranularität verteilt werden. Durch Nummerierung werden Wrapper-Instanzen Modellpartitionen zugeordnet. Zur Umsetzung von Punkt II) müssen alle Signale, die Wrapper-Instanzen verbinden, durch den neuen *PDESSignal* Channeltyp ausgetauscht werden. Anhand der Schnittstelle des *PDESSignal* Channeltyps kann der Lookahead spezifiziert werden.

4.4.7. Abbildung auf die Speicherarchitektur des SCC

Zur Abbildung auf die SCC Speicherarchitektur werden logische Prozesse wie in Abschnitt 4.3.5 mit RCCE UEs [15] als Betriebssystemprozesse implementiert. Im privaten Speicher jedes beteiligten SCC Kerns befindet sich ein vollständiges Duplikat des Programmcodes. Im Unterschied zum asymmetrischen Ansatz wird für die Speicherung der Daten kein externer SHM genutzt. Diese sind fast vollständig im privaten Speicher der SCC Kerne hinterlegt. Nur im MPB Bereich eines jeden logischen Prozesses lp_i existiert jeweils eine Instanz eines Shared Object und eines Circular Buffer Primitivs. Ersteres dient zur Speicherung der Linkzeit in Form der τ_i^{link} Variablen. Letzteres dient zur Realisierung der abstrakten FIFOs der logischen Links. Abb. 4.21 illustriert die resultierende Verteilung auf die Speicherarchitektur des SCC.

4.4.8. Teilautomatisierte Werkzeugunterstützung

Zur Unterstützung der beschriebenen symmetrischen Parallelisierungsstrategie ist eine teilautomatisierte Werkzeugkette entstanden, die beide Schritte der in Abschnitt 4.2.3 beschriebenen Methodik zur Simulationssynthese implementiert. Für das in dieser Arbeit verwendete HeMPS MPSoC [75] existierte bereits eine Generator (HeMPS Editor) mit einer grafischen Benutzeroberfläche zur manu-

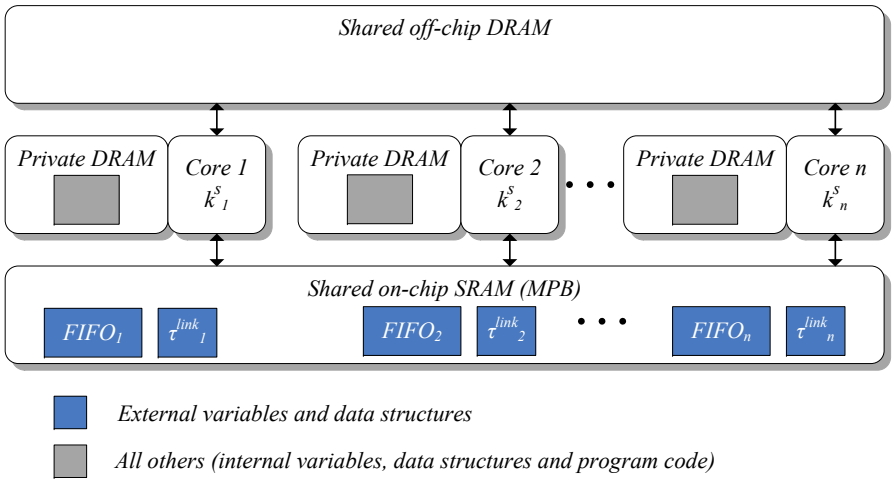


Abbildung 4.21.: Speichernutzung bei symmetrischer Kernelpartitionierung

ellen Abbildung der Applikation auf das MPSoC Modell. Dieser wurde um die notwendigen Funktionen erweitert.

4.4.8.1. Originaler HeMPS Editor

Mit der originalen Version des Generatorwerkzeugs kann sowohl eine HeMPS Hardwareplattform in Form von Anzahl der PEs, Topologie, Speichergröße, etc. als auch die Abbildung von Tasks einer oder mehrerer Applikationen auf die HeMPS Plattform spezifiziert werden. Als Ergebnis erhält man zunächst eine Konfigurationsdatei (*SimulationModelInfoFile.hmp* Datei), in der die Plattformspezifikation und die Abbildungsinformation in einem proprietären Format hinterlegt sind.

Mit Hilfe der *SimulationModelInfo.hmp* Datei und der HeMPS Komponentenbibliothek (*HeMPS Library*) wird das Modell der Hardwareplattform generiert. Mit der Abbildungsinformation in der *SimulationModelInfo.hmp* Datei, dem Applikationscode und den Templatedateien für das Plasma RTOS (*Tasks and RTOS*) wird die Softwareplattform erzeugt. Die Software wird anschließend kompiliert und das Task Repository des HeMPS Masters mit den resultierenden Binärdaten initialisiert. Das Ergebnis ist ein vollständiges Simulationsmodell, das Hardware und Software beinhaltet.

4.4.8.2. Erweiterter HeMPS Editor

Der beschriebene Prozess der Abbildung der Applikation auf das Simulationsmodell wurde direkt mit dem Prozess der Abbildung des Simulationsmodells auf die Ausführungsplattform kombiniert und in einen *Extended HeMPS Generator* integriert. Damit liegt die Durchführung beider Schritte prinzipiell im Aufgabenbereich des Modellerstellers. Der gesamte Ablauf ist in Abb. 4.22 dargestellt.

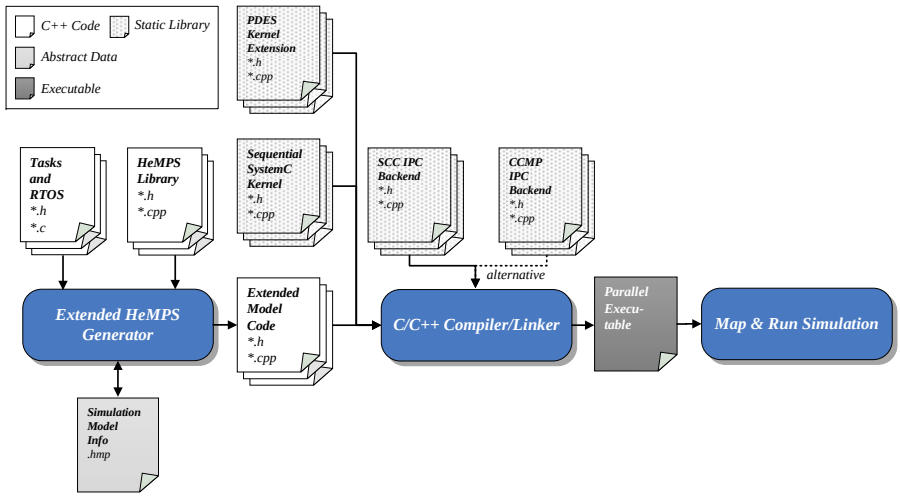


Abbildung 4.22.: Teilautomatisierte Werkzeugkette

Die entwickelte Lösung nutzt die in Abschnitt 4.4.6 beschriebene API zur manuellen Modellpartitionierung. Für jedes Toplevel Modul des HeMPS Modells (PEs, NIs, ROs) wurde ein Wrapper entwickelt. Sowohl die grafische Benutzeroberfläche des originalen HeMPS Editors als auch das proprietäre Format der Konfigurationsdatei wurden so erweitert, dass Partitionen aus Toplevelmodulen anhand von IDs spezifiziert werden können. Mit Hilfe von Platzhaltern, die manuell in ein Template der Topleveldatei des SystemC Modells eingefügt wurden, kann das Generatorwerkzeug den Code zur Instanziierung automatisch modifizieren und damit die Modellpartitionierung ändern. Das Ergebnis des Generierungsvorgangs ist der sog. Extended Model Code.

In Kombination mit den Quellcodedateien des parallelen SystemC Kernels (Bibliotheken der Kernelebene, der logischen Ebene und der Basisdienstebene) kann anschließend durch Kompilation und Verlinkung das *ParallelExecutable* erzeugt werden. In Abhängigkeit des gewählten Backends zur *Inter-Process Communication (IPC)* auf Basisdienstebene ist dieses auf dem SCC oder einem cachekohärenten SHM Multiprozessor ausführbar. Mit Hilfe eines simplen Skripts wird das

Parallel Executable auf der Zielplattform gestartet. Abb. 4.23 zeigt beispielhaft die erweiterte grafische Benutzeroberfläche des erweiterten HeMPS Generators. Im Beispiel existieren drei Partitionen. Module, die ein und derselben Partition angehören, sind farblich identisch unterlegt.

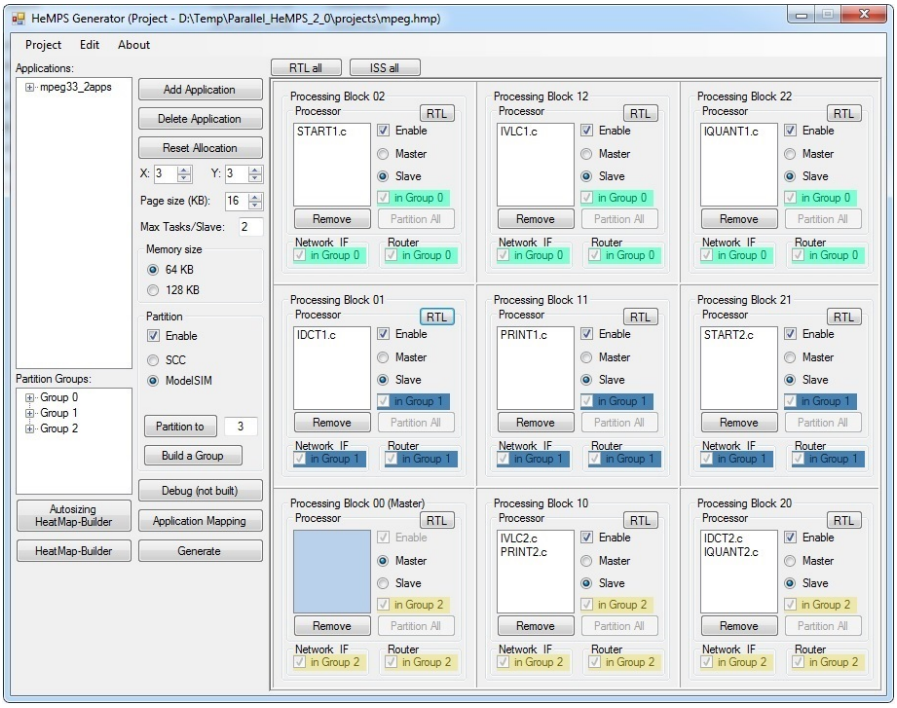


Abbildung 4.23.: Erweiterter HeMPS Editor

4.4.8.3. Randbedingungen zur Verteilung von HeMPS

Damit HeMPS korrekt ausgeführt wird, muss die Verteilung von Toplevel Modulen auf logische Prozesse und die Annotation von externen Signale so erfolgen, dass die Bedingungen aus Abschnitt 4.4.3.4 erfüllt sind. Eine Analyse des Quellcodes des Simulationsmodells lässt folgende Schlüsse zu:

- **Bedingung I**: Um das HeMPS Modell zwischen einem Network Interface und dem zugehörigen Router zu partitionieren, muss Δs^{τ} einen beliebigen Wert aus folgendem Intervall besitzen: $0 < \Delta s^{\tau} < \frac{\Delta \tau^{cycle}}{2}$. Dies ist möglich, da sich keines der Signale zwischen einem Network Interface und dem zugehörigen Router auf ein und demselben kombinatorischen Pfad befindet.

Die Schranke von $\frac{\Delta\tau^{cycle}}{2}$ ist durch das Kommunikationsprotokoll zwischen den NoC Routern gegeben, welches die positive und die negative Taktflanke nutzt. Um das sog. Time Creep Problem [113] zu vermeiden und einen maximalen Lookahead zu erhalten, sollte Δs^{τ} auf den maximal möglichen Wert gesetzt werden.

Zwischen Processing Element und Network Interface ist Bedingung I mit der Einschränkung $0 < \Delta s^{\tau} < \frac{\Delta\tau^{cycle}}{2}$ noch nicht erfüllt. Um den Lookahead nicht weiter reduzieren zu müssen und den Aufwand für die manuelle Partitionierung zu reduzieren, wurde auf eine Partitionierung zwischen Processing Elements und zugehörigen Network Interfaces verzichtet.

- **Bedingung II**: Da das HeMPS Modell keine unvollständigen Sensitivitätslisten besitzt, ist diese Bedingung immer erfüllt.
- **Bedingung III**: Da grundsätzlich alle externen Signale mit $0 < \Delta s^{\tau} < \frac{\Delta\tau^{cycle}}{2}$ annotiert sind und auf Annotationen mit $\Delta s^{\tau} = 0$ verzichtet wird, existieren generell keine kritische Zyklen zwischen logischen Prozessen.

4.4.9. Bewertung

4.4.9.1. Evaluation der Skalierbarkeit

Zur Bewertung der Skalierbarkeit wurde die Dauer der parallelen Ausführung unterschiedlich großer HeMPS Modelle auf einer unterschiedlichen Anzahl an SCC Kernen gemessen. Mit Hilfe der sequentiellen Ausführungsdauer auf einem SCC Kern wurde jeweils die erzielte Beschleunigung berechnet. Für die PEs kamen reine RTL Modelle und zyklenapproximative Simulatoren, sog. *Cycle-Approximate Simulators (CAS)*, zum Einsatz. Letztere können exakter auch als *Pin-Accurate Cycle-Approximate Level (PA-CAL)* Modelle bezeichnet werden. Sie besitzen eine signalbasierte Schnittstelle, modellieren Pipelinekonflikte intern aber nicht zeitlich akkurat. Dies resultiert in einem optimistischen Modell mit einem geringeren Anzahl an *Cycles per Instruction (CPI)* [RAS+11].

Auf den PEs von HeMPS wurde eine unterschiedliche Anzahl an fünfstufigen MPEG Decoder Pipelines ausgeführt. *TASK 1* einer Pipeline erzeugt 8x8 MPEG Blöcke. Diese werden von einem Variable Length Decoder in *TASK 2*, einer Inverse Quantization Stufe in TASK 3, einer Inverse Discrete Cosine Transformation Stufe in *TASK 4* und einem Ausgabetask (*TASK 5*) verarbeitet.

Die Tasks der Pipelines wurden zeilenweise auf die PEs abgebildet. Anschließend wurde das Modell mit einer Granularität von Tiles (1 Tile = 1 PE + 1 NI + 1 RO) partitioniert und möglichst gleichmäßig auf logische Prozesse verteilt. Dabei wurde darauf geachtet, dass sich möglichst immer benachbarte Tiles in einer Mo-

dellpartition befinden. Auf jedem SCC Kern wurde dann ein logischer Prozess ausgeführt. Falls die Anzahl an logischen Prozessen kleiner war als die Anzahl an SCC Kernen, so wurden die übrigen SCC Kerne nicht verwendet. Die Simulation wurde bei einer simulierten Taktfrequenz von 100 MHz ausgeführt. Dies resultierte generell in einer annotierten Verzögerung von $\frac{t_c}{2} = 5ns$ und damit einem generellen Lookahead von $\Delta l^{\tau} = 4.9ns$. Die gemessenen Charakteristika der Beschleunigung sind in den Abb. 4.24 und 4.25 dargestellt.

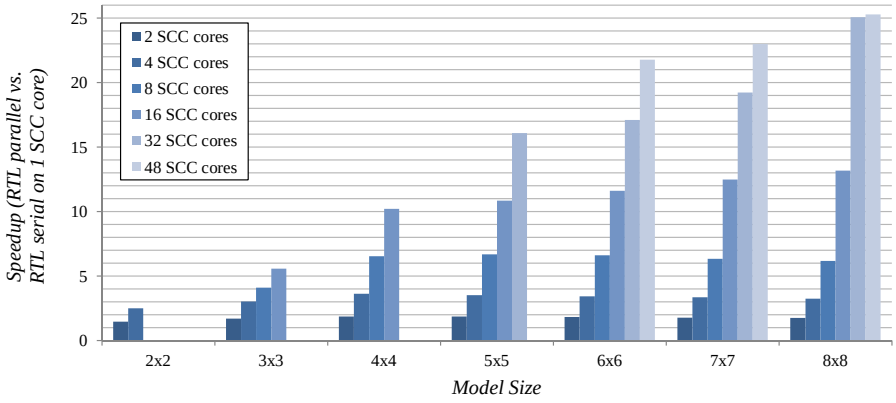


Abbildung 4.24.: Beschleunigung RTL parallel vs. RTL seriell

Abb. 4.24 zeigt die Beschleunigung reiner RTL Modelle mit Dimensionen von 2x2 bis 8x8. Im Unterschied zum asymmetrischen Verfahren war die Beschleunigung in allen gemessenen Fällen größer als 1.0x. Die Beschleunigung kleinerer Modelle ist aus zwei Gründen limitiert: Zum einen beschränkt die grobgranulare Partitionierung die maximale Anzahl möglicher Partitionen (im Fall eines 2x2 Modells ist diese Limitiert auf vier). Nimmt man eine konstante Anzahl an Partitionen an, so bieten große Modelle ein besseres Verhältnis zwischen parallelisierbaren und nicht parallelisierbaren Anteilen bzw. von Berechnungsanteilen zu Kommunikationsanteilen. Beispielsweise steigt die Beschleunigung von 2.5x auf 3.6x, wenn man anstelle eines 2x2 Modells ein 4x4 Modell auf vier Partitionen verteilt simuliert.

Die größte Beschleunigung wurde im Fall eines 8x8 Modells gemessen, das auf 48 SCC Kernen ausgeführt wurde. Sie beträgt 25.3x. Dieser Wert ist nahezu identisch zu dem Wert, der für das gleiche Modell auf nur 32 SCC Kernen gemessen wurde. Ein Grund dafür ist, dass mit 48 SCC Kernen das Limit der verfügbaren Kerne erreicht ist. Die Anzahl der in beiden Fällen genutzten SCC Kerne unterscheidet sich nur noch um den Faktor 1.5 und nicht um den Faktor 2.0, wie in allen anderen Fällen. Außerdem resultiert die Nutzung von 32 SCC Kernen in Kombination mit einem 8x8 Modell in einer idealen Partitionierung, bei der

jeder SCC Kern zwei HeMPS Tiles simuliert. Im Gegensatz dazu ist die Partitionierung im Fall von 48 SCC Kernen unausgewogen: 32 SCC Kerne simulieren nur ein Tile während die übrigen 16 Kerne zwei Tiles ausführen müssen.

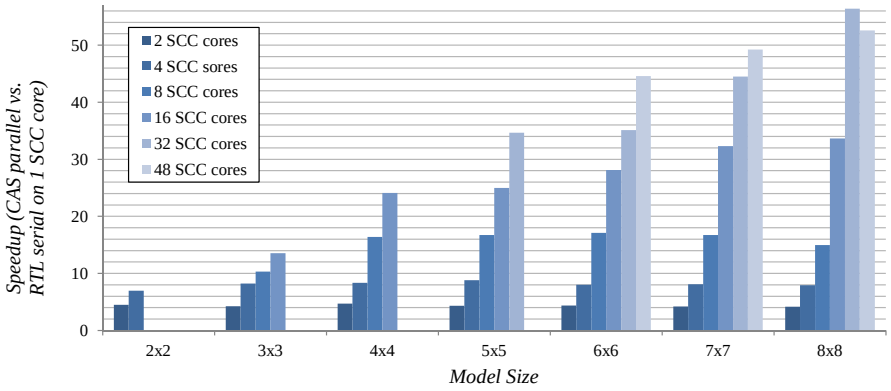


Abbildung 4.25.: Beschleunigung CAS parallel vs. RTL seriell

Tauscht man die auf RT Level modellierten Plasmakerne gegen die CAS aus, so kann die Beschleunigung im Vergleich zur reinen RTL Simulation weiter gesteigert werden. Die Messergebnisse sind in Abb. 4.25 dargestellt. Im Vergleich zu sequentieller RTL Simulation wurde bei CAS-basierter Simulation für ein 8x8 Modell auf 32 SCC Kernen eine Beschleunigung von 56.3x gemessen. Die zusätzliche Beschleunigung ist auf die höhere Abstraktion der CAS zurückzuführen. Eine interessante Beobachtung ist die Tatsache, dass die Beschleunigung auf 32 SCC Kernen größer war als auf 48 Kernen. Dies kann auf die gleiche Ursache zurückgeführt werden wie die limitierte Beschleunigung des reinen RTL Modells auf 48 SCC Kernen. Im Fall der CAS tritt der Effekt wegen der geringeren parallelisierbaren Anteile im Modell stärker in Erscheinung.

4.4.9.2. Evaluation des Einflusses der Modellabbildung

In einer zweiten Untersuchung wurde der Einfluss betrachtet, den die Art und Weise der Abbildung I) von Tasks der simulierten Anwendung auf das Simulationsmodell II) des Simulationsmodells auf den SCC hat. Dafür wurde zunächst eine fünf Tasks umfassende Dummy Applikation geschrieben. Ähnlich wie bei der MPEG Applikation aus dem vorigen Abschnitt, bilden die Tasks eine Pipeline, bei der jeder Task nur mit seinem Vorgänger und seinem Nachfolger kommuniziert. Die Tasks führen eine Endlosschleife aus, in der so viele Nachrichten wie möglich über das NoC an den jeweiligen Nachfolgetask geschickt werden.

Fünf Instanzen dieser Applikation wurden horizontal und zeilenweise auf ein 5x5 HeMPS Modell abgebildet. Dies ist in Abb. 4.26 dargestellt.

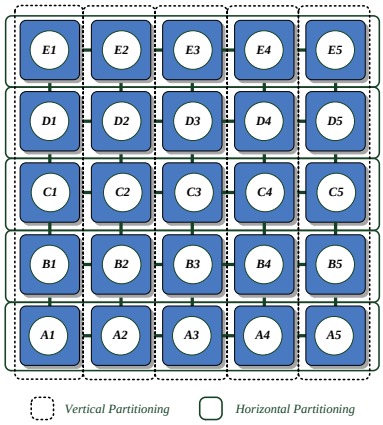


Abbildung 4.26.: Abbildung von Tasks und Modellpartitionen

Anschließend wurde das Modell zunächst ebenfalls horizontal (identisch zur Orientierung der Pipelines) und dann vertikal (orthogonal zur Orientierung der Pipelines) partitioniert. Wegen des XY Routings des Hermes NoCs war im ersten Fall die Kommunikation zwischen Tasks ausschließlich auf Modellpartitionen beschränkt und damit rein intern. Im zweiten Fall kommunizierten hingegen nur Tasks miteinander, die in unterschiedlichen Modellpartitionen ausgeführt wurden. Die fünf logischen Prozesse wurden schließlich in einer Reihe nebeneinander auf fünf SCC Kerne abgebildet. Die Simulation wurde für 10^7 simulierte Zyklen unter Verwendung von RTL und CAS-basierten Plasmakernen ausgeführt. Tabelle 4.1 zeigt die gemessenen Laufzeiten der beschriebenen vier Fälle (Fälle A-D) relativ zu CAS-basierter Simulation mit horizontaler Partitionierung (Fall A).

Durch den Wechsel von den CAS zur RTL Beschreibung wird die Laufzeit erwartungsgemäß mehr als verdoppelt. Vergleicht man für beide Fälle die Partitionierung, so resultiert eine vertikale im Vergleich zur einer horizontalen Partitionierung in einer Erhöhung der Laufzeit um nur 1% für die CAS basierte Modell und um 3,7% für das RTL Modell. In beiden Fällen, CAS und RTL überwiegt auch für vertikale Partitionierung offensichtlich weiterhin der Aufwand für die Berechnung des Simulationsmodells gegenüber der Kommunikation zwischen logischen Prozessen.

4.4.9.3. Diskussion

Die Messergebnisse zeigen, dass der Übergang vom asymmetrischen zum symmetrischen Verfahren in Kombination mit statischer Partitionierung zu einer signifikanten Steigerung der Effizienz zyklenakkurater MPSoC Simulationen beitragen kann. Dies ist zum einen auf die Reduktion des Synchronisationsaufwandes zurückzuführen, die mit der Relaxation auf die Ebene von Timed Notifications einhergeht. Zum anderen wird die Skalierbarkeit durch den Übergang von einem zentralen zu einem dezentralen Verfahren verbessert, welches ausschließlich nachrichtenbasierte On-Chip Kommunikation nutzt.

Darüber hinaus konnte gezeigt werden, dass bei zyklenakkuraten signalbasierten Modellen eine leichte Abhängigkeit der Ausführungsperformanz von der auf dem MPSoC Modell ausgeführten Software existiert. In diesem Zusammenhang soll später untersucht werden, inwieweit dies auch für abstraktere MPSoC Modelle gilt.

Ein Nachteil der verwendeten Annotationsmethode ist die Tatsache, dass kritische Zyklen zwischen logischen Prozessen unbedingt vermieden werden müssen. Dadurch ist die Anzahl möglicher Partitionierungen beschränkt. Außerdem geht durch die Annotationsmethode Genauigkeit verloren: Beispielsweise kann in Abhängigkeit von Partitionierung und Annotationen eine unterschiedliche Anzahl an Deltacycles und Timedcycles auftreten. Auch die parallele Simulation eines RTL Modells mit unvollständigen Sensitivitätslisten führt u.U. zu anderen Ergebnissen als eine sequentielle Simulation des Modells.

Die beschriebene Kernelimplementierung verursacht zusätzlich eine nicht-deterministische Anzahl an Deltacycles pro Timedcycle. Deltaereignisse werden über mehrere Timed- und Deltacycles *verschmiert*. Dies ist evtl. nicht für alle denkbaren Verifikationsszenarien wünschenswert. Die Kernel API zur Partitionierung eines Simulationsmodells limitiert die Granularität auf Toplevel Module. In Verbindung mit der teilautomatisierten der Werkzeugkette ist die Flexibilität beschränkt, da der Generator im Falle eines anderen Modells adaptiert oder komplett neu entwickelt werden muss.

Tabelle 4.1.: Einfluss von Task- und Modellabbildung

Fall	Partitionierung	Plasma Modell	Relative Laufzeit
A	horizontal	CAS	1
B	vertikal	CAS	1.01
C	horizontal	RTL	2.42
D	vertikal	RTL	2.51

4.5. Adaptive symmetrische Strategie

Im folgenden Abschnitt wird ein adaptiver Ansatz vorgestellt, der mit dem Ziel entwickelt wurde, die Nachteile bzgl. Annotation, Partitionierung und Determinismus der symmetrischen asynchronen Strategie aus dem vorigen Abschnitt auszugleichen. Globale Synchronisation wird so weit wie möglich vermieden. Dazu wird eine Kombination aus rein lokaler und kollektiver Synchronisation in abgeschlossenen Bereichen der logischen Prozesstopologie angewendet. Dies erfordert eine umfassende modell- und plattformadaptive Konfiguration der Simulationsumgebung. Als Voraussetzung dafür ist eine tiefgreifende Modellanalyse erforderlich. Zur Unterstützung von Konfiguration und Modellanalyse wird der Ansatz daher durch eine vollständig automatisierte Werkzeugkette ergänzt. Im folgenden Abschnitt werden zunächst Konzepte und Implementierungsaspekte beleuchtet. Anschließend wird der Ansatz in einer umfangreichen Fallstudie bewertet. Ein Großteil der Ergebnisse dieses Abschnitts wurde im Rahmen einer vom Autor betreuten Masterarbeit [Red14] erarbeitet und ist in Zusammenarbeit entstanden. Teile sind in [RRB+14] publiziert.

4.5.1. Anforderungen

Für die adaptive symmetrische Strategie wurde Anforderung V aus Abschnitt 4.4.1 in folgender Form modifiziert:

V) **Synchronisation**: Eine globale Synchronisation soll *soweit wie möglich* vermieden werden.

Daneben wird eine zusätzliche Anforderung gestellt:

VI) **Delta Notifications**: Die durch die Deltacycles definierte partielle Ordnung soll erhalten bleiben.

4.5.2. Adaptive logische Ebene

Bei der symmetrischen asynchronen Strategie aus Abschnitt 4.4 wird auf die deterministische Synchronisation des Deltacyclezählers δ zugunsten eines skalaren Zeitmodells verzichtet. Um die durch die Deltacycles definierte partielle Ordnung zu erhalten, ist eine Synchronisation des Deltacyclezählers δ allerdings unverzichtbar. Um dies auch ohne globale Synchronisation zu ermöglichen, wird zunächst ein angepasstes Zeitmodell eingeführt. Darauf basierend werden anschließend weitere geeignete Modelleigenschaften formuliert, die eine genauere Klassifikation von logischen Links als in Abschnitt 4.4.3.4 erlauben. Dies ist die Voraussetzung für die Ableitung unterschiedlicher Kausalitätsbedingungen

innerhalb logischer Prozesse und die (weitgehende) Vermeidung globaler Synchronisation bei gleichzeitig deltazyklengenauer paralleler Ausführung.

4.5.2.1. Angepasstes SystemC Zeitmodell

Beim Zeitmodell des SystemC Standards [27] zählt δ die innerhalb eines gesamten Simulationslaufes ausgeführten Deltacycles und ist unabhängig von τ (vgl. Abschnitt 2.3.2.2). Zur vollständigen Vermeidung globaler Synchronisation ist es vorteilhaft, von diesem Modell auf eine Variante überzugehen, bei der nicht mehr die Deltacycles einer gesamten Simulationsausführung, sondern mit einer Variablen θ jeweils die Deltacycles innerhalb einzelner Zeitschritte bzgl. τ gezählt werden. θ wird dann zu Beginn eines jeden Zeitschritts auf null gesetzt. In dieser Weise ist beispielsweise auch das Zeitmodell der DE Domäne von Ptolemy II [217] implementiert.

Durch das Zurücksetzen von θ mit jedem Zeitfortschritt bzgl. τ wird eine Beziehung zwischen τ und θ hergestellt, anhand derer man auftretende Notifications in einer parallelen Simulation bestehend aus mehreren logischen Prozessen bis auf die Ebene von Deltacycles leichter partiell ordnen kann. Insbesondere ermöglicht es diese Art der Zeitmodellierung den logischen Prozessen, zu einem Simulationszeitpunkt τ eine unterschiedliche Anzahl an Deltacycles ausführen, ohne miteinander explizit synchronisieren zu müssen. Man denke hier an zwei logische Prozesse lp_1 und lp_2, die ausschließlich über einen logischen Link l_{12} von lp_1 nach lp_2 miteinander verbunden sind. In diesem Fall kann lp_2 u.U. eine größere Anzahl an Deltacycles bei einem bestimmten τ ausführen als lp_1, ohne die Notwendigkeit, dies lp_1 mitteilen zu müssen.

4.5.2.2. Deterministisches Basisverfahren zur Synchronisation

Mit dem vektoriellen Zeitmodell müssen Zeitstempel (z.B. von Signalnachrichten) sowie der Lookahead um θ ergänzt werden. Die Linkzeit aus Definition 4.7 und die LOC Bedingung aus Definition 4.8 werden dann durch eine erweiterte Linkzeit und eine erweiterte LOC Bedingung ersetzt:

Definition 4.11 *(**Erweiterte Linkzeit**): Die **erweiterte Linkzeit** $t_{ij}^{link} = (\tau_{ij}, \theta_{ij})$ eines logischen Links l_{ij} von lp_i nach lp_j entspricht der letzten in lp_j bekannten lokalen Zeit $t_i = (\tau_i, \theta_i)$ von lp_i.*

Definition 4.12 *(**ELOCC: Extended Local Causality Condition**): Ein logischer Prozess lp_j darf das nächste Ereignis bei $t_j^{next} = (\tau_j^{next}, \theta_j^{next})$ verarbeiten, wenn*

$$\forall lp_i \in LP_j^{adj} : t_j^{next} \leq t_{ij}^{link} + \Delta l_{ij}^{\tau,\theta}. \tag{4.9}$$

Für den maximal möglichen Zeitfortschritt gilt:

$$t_j^{max} = \min_{\forall i}(t_{ij}^{link} + \Delta l_{ij}^{\tau,\theta}) \tag{4.10}$$

Um das Evaluate/Update Paradigma korrekt und deterministisch auszuführen, muss garantiert sein, dass alle Signalnachrichten für einen Deltacycle empfangen wurden, bevor dieser ausgeführt wird. Angenommen, der Lookahead eines logischen Links ist generell durch $\Delta l^{\tau,\theta} = (0,1)$ gegeben. Neben einem kausal korrekten Zeitfortschritt garantiert die Einhaltung der erweiterten lokalen Kausalitätsbedingung genau dann zusätzlich den vollständigen Empfang aller Signalnachrichten für einen Deltacycle in lp_j, bevor dieser ausgeführt wird, wenn

1. lp_j die Linkzeit im Unterschied zu Abschnitt 4.4.3.2 nur noch über eine lokal externe Variable t_i^{link} ableitet und nicht mehr zusätzlich den Zeitstempel von Signalnachrichten nutzt.
2. lp_i die Variable t_i^{link} immer nur dann auf den Wert des gerade ausgeführten Deltacycles inkrementiert, wenn bereits alle Signalnachrichten, die im gerade ausgeführten Deltacycle generiert wurden, verschickt sind.

Basierend auf dem angepassten Zeitmodell und dem beschriebenen Basisverfahren werden in den weiteren Abschnitten spezielle Eigenschaften von RTL u.ä. signalbasierten Modellen identifiziert, die für eine genauere Klassifikation von logischen Links in deadlock-kritisch und -unkritisch genutzt werden können. Die neue Klassifikation ist die Voraussetzung für die Vermeidung globaler Synchronisation trotz eines generellen Lookaheads von $\Delta l^{\tau,\theta} = (0,1)$. Bei den folgenden Definitionen wird daher einer genereller Lookahead von $\Delta l^{\tau,\theta} = (0,1)$ vorausgesetzt.

4.5.2.3. Kanalaktivität

Eine geeignete Modelleigenschaft, die als Grundlage zur Klassifikation logischer Links im Hinblick auf den Erhalt der durch die Deltacycles definierten partiellen Ordnung dienen kann, ist die *Kanalaktivität*:

Definition 4.13 *(**CAP: Channel Activity Property**): Ein SystemC Primitive Channel ch wie z.B. ein Signal ist **aktiv** (bzgl. des SystemC Prozesses p), wenn p sensitiv auf ch ist. Ansonsten ist ch **passiv** (bzgl. p).*

In einem RTL Modell ist ein Signal s aktiv bzgl. Prozess p, wenn das Signal Teil der statischen Sensitivitätsliste von p ist. Passive Signale treten typischerweise in synchronen Prozessen auf, die nur auf ein Taktsignal sensitiv sind. In diesem Fall sind alle Signale außer dem Taktsignal passiv bzgl. des synchronen Prozesses, da deren Wertänderung nur infolge einer bzgl. τ verzögerten Taktflanke ausgelesen werden kann.

Mit der CAP kann aus dem Prozess-Signal Graphen $G_{PS}(P, S)$ aus Definition 4.2 der sog. *Sensitivitätsgraph* $G_S(P, E)$ abgeleitet werden. Dieser ist später für die Identifikation weiterer Modelleigenschaften notwendig. Im Gegensatz zu G_{PS} enthält G_S nur dann eine gerichtete Kante e_{ab} von Knoten p_a nach p_b, wenn p_b auf mindestens eines der Signale, die p_a und p_b verbinden, zusätzlich sensitiv ist. Dabei sind Taktsignale nicht Teil von G_S. Abb. 4.27 illustriert beispielhaft einen Sensitivitätsgraphen mit Prozessen p_1 bis p_7. In der Abbildung werden die Prozesse p_1 und p_2 durch ein zeitverzögerndes Taktsignal (zeitgesteuertes Ereignisobjekt ω^τ) aktiviert. Infolge der Aktivität von Signalen können u.U. alle anderen (sensitiven) Prozesse auch ausgeführt werden. Aus der Aktivität von Signalen kann folgende Definition für die Aktivität eines logischen Links abgeleitet werden:

Definition 4.14 *(**LAP: Link Activity Property**): Ein logischer Link l_{ij} heißt **aktiv** (bzgl. lp_j), wenn die Kernelkomponente in lp_j durch eine über l_{ij} übertragene Nachricht eine Aktivierung eines SystemC Prozesses auslösen kann, ansonsten heißt l_{ij} **passiv**. Bzgl. SystemC RTL heißt l_{ij} folglich aktiv, wenn l_{ij} mindestens ein Signal bündelt, das durch eine Kante $e \in G_S(P, E)$ repräsentiert wird. Ansonsten heißt l_{ij} passiv.*

Da passive logische Links SystemC Prozesse nicht aktivieren können, müssen sie im Empfänger bei der Prüfung der ELOCC im Kontext des Zeitfortschritts nicht berücksichtigt werden, sondern nur im Kontext einer deterministischen Nachrichtenübertragung. Ersteres ist möglich, da die Generierung zyklischer Abhängigkeiten auf der Ebene von Deltacycles über passive logische Links ausgeschlossen ist. Letzteres ist möglich, wenn die in Abschnitt 4.5.2.2 beschriebenen Änderungen am Basisverfahren verwendet werden. Dies führt zur modifizierten ersten Bedingung für Link Kritikalität (vgl. Definition 4.10):

Definition 4.15 *(**MLCC1: Modified First Link Criticality Condition**): Ein deltaverzögerter logischer Link ist deadlock-unkritisch, wenn er passiv ist.*

In der Implementierung kann die unterschiedliche Art der Nutzung der ELOCC durch zwei aufeinanderfolgende Prüfungen der ELOCC realisiert werden, wobei passive und damit unkritische logische Links zwar bei der Prüfung der vollständigen Übermittlung von Nachrichten für einen Deltacycle vor der Ausführung dieses Deltacycles berücksichtigt werden müssen, nicht aber bei der Prüfung eines möglichen Zeitfortschritts bzgl. τ.

4.5.2.4. Delta-Beschränktheit

Um die Chance für deadlock-unkritische logische Links weiter zu vergrößern und damit die Anzahl valider Partitionierungen des Modells zu erhöhen, wird eine weitere Modelleigenschaft in Betracht gezogen. Diese wird als *Delta-Beschränktheit* oder schlicht *Beschränktheit* von SystemC Prozessen oder SystemC Signalen bezeichnet:

Definition 4.16 *(**DBP: Delta-Boundedness Property**): Ein Prozess p heißt **delta-beschränkt** bzgl. eines zeitgesteuerten Ereignisobjekts ω^τ, wenn es möglich ist, ausgehend von einer Timed Notification n_ω^τ von ω^τ eine maximale Anzahl an Deltacycles $\Delta_p^{bound}(\omega^\tau) < \infty$ zu bestimmen, ab der Aktivierungen von p bis zur nächsten Timed Notification ausgeschlossen werden können. $\Delta_p^{bound}(\omega^\tau)$ wird als **Delta-Schranke** von Prozess p bzgl. ω^τ bezeichnet.*

*Ein SystemC Signal s heißt **delta-beschränkt** bzgl. eines zeitgesteuerten Ereignisobjekts ω^τ, wenn es möglich ist, ausgehend von einer Timed Notification n_ω^τ von ω^τ eine maximale Anzahl an Deltacycles $\Delta_s^{bound}(\omega^\tau) < \infty$ zu bestimmen, ab der schreibende Zugriffe auf s bis zur nächsten Timed Notification ausgeschlossen werden können. $\Delta_s^{bound}(\omega^\tau)$ wird als **Delta-Schranke** von Signal s bzgl. ω^τ bezeichnet.*

Da wegen MLCC1 bereits ausgeschlossen ist, dass passive logische Links zu einem Deadlock führen können, ist die Eigenschaft der Delta-Beschränktheit letztlich nur für aktive Signale relevant. Daher genügt es, diese Eigenschaft anhand des im vorigen Abschnitt eingeführten Sensitivitätsgraphen G_S herzuleiten.

Bestimmung der Delta-Beschränktheit von Prozessen

Man betrachte erneut den Sensitivitätsgraphen G_S aus Abb. 4.27. Die Prozesse p_1 und p_2 werden aufgrund eines zeitverzögernden Taktsignals *clk* aktiviert. Die Zeitverzögerung wird signalintern mit Hilfe des zeitgesteuerten Ereignisobjekts ω^τ erreicht. Prozesse, die unmittelbar auf zeitgesteuerte Ereignisobjekte sensitiv sind, werden immer im ersten Deltacycle ($\theta = 0$) nach dem Auftreten der zugehörigen Timed Notification ausgeführt. Folglich ist für jedes zeitgesteuerte

Ereignisobjekt $\omega^\tau \in \Omega$ eine Menge $R(\omega^\tau, \theta = 0) \subseteq R$ von lauffähigen Prozessen definiert, die nach der Timed Notification von ω^τ in Deltacycle $\theta = 0$ garantiert evaluiert werden.

Bei bekanntem Ereignisobjekt ω^τ kann man die Beschaffenheit von $R(\omega^\tau, \theta = 0)$ durch statische Analyse vollständig herleiten. Für $\theta > 0$ lässt sich $R(\omega^\tau, \theta)$ allerdings vor der Laufzeit nicht mehr exakt vorherbestimmen. Eine genaue Bestimmung von $R(\omega^\tau, \theta)$ ist nur noch durch eine dynamische Analyse möglich, wie sie typischerweise zur Generierung eines DDG verwendet wird (vgl. Abschnitt 4.2.4.1). Ein DDG ist allerdings deswegen nicht für die Herleitung von Delta-Schranken geeignet, da er nur einen einzigen, meist zeitlich beschränkten Ausführungspfad und nicht alle möglichen Ausführungspfade vollständig beschreibt.

Alternativ kann man im Rahmen einer statischen Abhängigkeitsanalyse jedoch die Mengen der Prozesse $\mathcal{R}(\omega^\tau, \theta)$ bestimmen, die im Zyklus θ *potentiell* evaluiert werden. Beginnend bei der bereits bekannten Menge $\mathcal{R}(\omega^\tau, 0)$ können die Mengen für $\theta > 0$ somit konservativ geschätzt werden. Für den Deltacycle θ gilt dann im Allgemeinen $R(\omega^\tau, \theta) \subseteq \mathcal{R}(\omega^\tau, \theta)$.

Aufgrund der Sensitivität müssen Prozesse in $\mathcal{R}(\omega^\tau, \theta + 1)$ von einer Delta Notification aktiviert worden sein, die einer der Prozesse in $\mathcal{R}(\omega^\tau, \theta)$ generiert hat. Die Evaluation eines Prozesses, der durch einen Knoten p_j in G_S repräsentiert wird, kann nur dann per Delta Notification von einem Prozess ausgelöst worden sein, der durch einen Knoten p_i in G_S repräsentiert wird, wenn von p_i zu p_j eine gerichtete Kante existiert. Dieser Zusammenhang kann allgemein durch den folgenden Ausdruck beschrieben werden:

$$\mathcal{R}(\omega^\tau, \theta + 1) = \{p_j : \exists e_{ij} \in E \wedge p_i \in \mathcal{R}(\omega^\tau, \theta)\} \tag{4.11}$$

Damit Prozesse einer beliebigen Menge $\mathcal{R}(\omega^\tau, \theta + n)$ mit $n \in \mathbb{N}\backslash\{0\}$ von Prozessen der Menge $\mathcal{R}(\omega^\tau, \theta)$ direkt oder indirekt aktiviert werden können, müssen erstere über einen Pfad in G_S von letzteren aus erreichbar sein. Oder anders ausgedrückt: Zu einem beliebigen Prozess $p_i \in \mathcal{R}(\omega^\tau, \theta)$, der potentiell bei Deltacycle θ evaluiert wird, muss ein sog. *Aktivierungspfad* der Länge θ im Graphen G_S existieren, der bei einem durch ω^τ aktivierten Prozess beginnt. Die Mengen $\mathcal{R}(\omega^\tau, \theta)$ können mit Hilfe einer Tiefensuche [152] bestimmt werden, die für jeden Prozess $p_i \in \mathcal{R}(\omega^\tau, 0)$ mit p_i als Startknoten wiederholt wird.

Für einen bestimmten Prozess p_i kann u.U. ein unendlich langer Aktivierungspfad existieren, nämlich dann, wenn p_i Teil eines Zyklus in G_S ist oder von einem Zyklus aus erreichbar ist. p_i ist dann in unendlich vielen Mengen $\mathcal{R}(\omega^\tau, \theta)$ enthalten.

Sofern p_i nicht von einem Zyklus aus erreichbar ist, kann man eine obere Schranke $\Delta_{p_i}^{bound}(\omega^\tau)$ für den letztmöglichen Deltacycle angeben, in dem p_i evaluiert wird. Der Wert von $\Delta_{p_i}^{bound}(\omega^\tau)$ entspricht der Länge des längsten Aktivierungspfades, der in p_i endet. Falls $\Delta_{p_i}^{bound}(\omega^\tau) < \infty$, so gilt allgemein:

$$\theta > \Delta_{p_i}^{bound}(\omega^\tau) \Rightarrow p_i \notin \mathcal{R}(\omega^\tau, \theta) \Rightarrow p_i \notin R(\omega^\tau, \theta). \tag{4.12}$$

Die Auflistung aller Mengen $\mathcal{R}(\omega^\tau, \theta)$, in denen ein SystemC Prozess p_i enthalten ist, zusammen mit dem Wert der Delta-Schranke $\Delta_{p_i}^{bound}(\omega^\tau)$, sofern diese existiert, wird im Folgenden auch als das sog. *Aktivitätsmuster* von p_i bzgl. ω^τ bezeichnet.

Im Beispiel aus Abb. 4.27 sind die Prozesse p_4, p_5 und p_6 Teil eines unendlich langen Aktivierungspfades. Für den Prozess p_3 aus Abb. 4.27 gilt hingegen $\Delta_{p_3}^{bound}(\omega^\tau) = 1$.

Bestimmung der Delta-Beschränktheit von aktiven Signalen

Kommunikation über Signale besteht immer aus einem Schreib- und einem zugehörigen Lesevorgang. Die Kommunikation eines Prozesses p_i mit einem Prozess p_j über ein aktives Signal, welches durch eine Kante e_{ij} im Graphen G_S repräsentiert wird, ist erst dann vollständig abgeschlossen, wenn das Signal nach einem Schreibvorgang von p_i zum ersten Mal von p_j gelesen wird. Etwas formeller ausgedrückt, muss für eine Kommunikation bei ein und demselben τ ein Schreibvorgang bei einem $t_{e_{ij}}^{write} = (\tau, \theta_{e_{ij}}^{write})$ und ein Lesevorgang bei einem $t_{e_{ij}}^{read} = (\tau, \theta_{e_{ij}}^{read})$ erfolgen. Wegen der Delta-Verzögerung von Signalen muss $t_{e_{ij}}^{write} < t_{e_{ij}}^{read}$ gelten. Im Allgemeinen ist bei gleichem τ *potentiell* Kommunikation über ein aktives Signal möglich, das durch eine Kante $e_{ij} \in G_S(E, P)$ repräsentiert wird, wenn

$$p_i \in \mathcal{R}(\omega^\tau, \theta_{e_{ij}}^{write}) \wedge p_j \in \mathcal{R}(\omega^\tau, \theta_{e_{ij}}^{read}) \text{ mit } \theta_{e_{ij}}^{write} < \theta_{e_{ij}}^{read}. \tag{4.13}$$

Um für ein durch e_{ij} repräsentiertes aktives Signal eine Delta-Schranke bzgl. ω^τ herzuleiten, muss der letzte Deltacycle θ bestimmt werden, in dem Kommunikation über e_{ij} innerhalb eines durch ω^τ ausgelösten Timedcycles möglich ist. Dieser kann mit Hilfe der Delta-Schranke $\Delta_{p_i}^{bound}(\omega^\tau)$ des auf e_{ij} schreibenden Prozesses p_i berechnet werden. Spezifiziert man die Delta-Schranke von e_{ij} mit dem letztmöglichen Schreibzeitpunkt (der Lesezeitpunkt ergibt sich aufgrund

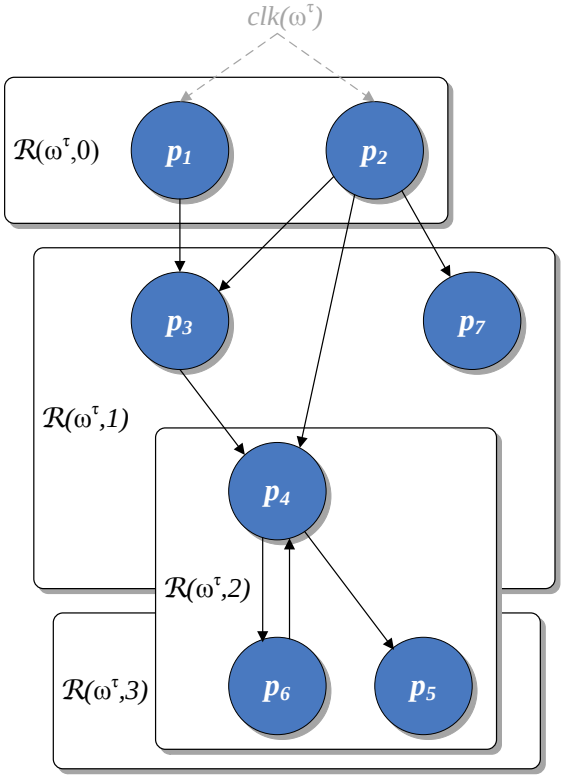


Abbildung 4.27.: Sensitivitätsgraph G_S mit den Mengen $\mathcal{R}(\omega^\tau, \theta)$ (Quelle: [Red14])

der Aktivitätseigenschaft durch Addition von eins), so kann $\Delta_{e_{ij}}^{bound}(\omega^\tau)$ wie folgt bestimmt werden:

$$\Delta_{e_{ij}}^{bound}(\omega^\tau) = \Delta_{p_i}^{bound}(\omega^\tau). \tag{4.14}$$

Basierend auf der Definition der Delta-Beschränktheit von einzelnen Signalen lässt sich die Eigenschaft der Delta-Beschränktheit für einen logischen Link wie folgt definieren:

Definition 4.17 *(**LBP: Link Boundedness Property**): Ein logischer Link l heißt **delta-beschränkt** bzgl. eines Ereignisobjekts ω^τ, wenn auf l delta-beschränkte aber keine unbeschränkten aktiven Signale abgebildet sind. $\Delta_l^{bound}(\omega^\tau)$, die Delta-Schranke von l bzgl. ω^τ, entspricht dem Maximum der Delta-Schranken aller auf l abgebildeten aktiven Signale bzw. der sie repräsentierenden Kanten aus dem Sensitivitätsgraphen G_S:*

$$\Delta_l^{bound}(\omega^\tau) = \max_{\forall e \in E(l)} (\Delta_e^{bound}(\omega^\tau)). \tag{4.15}$$

Dabei bezeichnet $E(l) \subseteq E$ hier die Teilmenge der Signalkanten aus G_S, die auf l abgebildet sind. Aus der Delta-Beschränktheit folgt für die Kritikalität aktiver logischer Links:

Definition 4.18 *(**LCC2: Second Link Criticality Condition**): Ein delta-verzögerter und aktiver logischer Link ist deadlock-unkritisch, wenn er delta-beschränkt ist. Ansonsten ist er deadlock-kritisch.*

Angenommen lp_i und lp_j sind durch einen delta-beschränkten logischen Link l_{ij} verbunden. Um kausale Korrektheit während der parallelen Simulation zu gewährleisten, muss innerhalb eines Timedcycles exakt nach dem Deltacycle, welcher durch die Delta-Schranke $\Delta_{l_{ij}}^{bound}$ spezifiziert ist, zum letzten Mal zwischen lp_i und lp_j synchronisiert werden, bevor der nächste Zeitfortschritt bzgl. τ durchgeführt werden kann. Anschließend existieren dann keine echten Datenabhängigkeiten mehr auf l_{ij} zum aktuellen Zeitpunkt τ. Im Allgemeinen gilt folgende zusätzliche Kausalitätsbedingung:

Definition 4.19 *(**LOBC: Local Bound Condition**): Ein logischer Prozess lp_j hat die Delta-Schranke überschritten, sobald folgende Bedingung für alle eingehenden aktiven, delta-verzögerten aber delta-beschränkten logischen Links $l_{ij} \in L_j^b$ erfüllt ist:*

$$\forall i, i \neq j : t_{l_{ij}}^{link} > (\tau_j, \Delta_{l_{ij}}^{bound}(\omega^\tau)). \tag{4.16}$$

Wird das SystemC Modell so auf logische Prozesse abgebildet, dass keine kritischen Zyklen existieren, dann ist die Erfüllung der LOBC ein weiteres notwendiges Kriterium für einen Zeitfortschritt bzgl. τ. Hinreichend ist nur die Erfüllung von beiden Kausalitätsbedingungen, der ELOCC und der LOBC.

4.5.2.5. Dynamische Auflösung zirkulärer Abhängigkeiten

Deadlock-unkritische logische Links sind fundamental für die Vermeidung von kritischen Zyklen im Graphen der logischen Prozesse G_{LP}. In manchen Fällen kann die Abbildung eines Modells auf logische Prozesse, ohne dabei kritische Zyklen zu generieren, u.U. aber zu einer unausgeglichenen Partitionierung führen, die die Performanz negativ beeinflusst. Aus diesem Grund ist es wünschenswert, Zyklen in manchen Regionen der Topologie von G_{LP} zu erlauben aber gleichzeitig Mechanismen bereitzustellen, mit deren Hilfe Deadlocks während der Laufzeit aufgelöst werden können.

Als ein Beispiel betrachte man die logischen Prozessgraphen G_{LP} and G_{LP}^{crit} in Abb. 4.28. Im Unterschied zu G_{LP} aus Abb. 4.18 sind die Links l_{01} and l_{74} nun deadlock-kritisch. Dies resultiert in der Existenz von zwei disjunkten stark zusammenhängenden Komponenten (engl. *Strongly Connected Components (SCC)*) in G_{LP}^{crit} mit einer Kardinalität größer oder gleich zwei, nämlich $\{lp_0, lp_1\}$ und $\{lp_4, lp_5, lp_7, lp_8\}$. Diese werden als *Domänen* (engl. *Domain*) bezeichnet:

Definition 4.20 *(**Domäne / Domänenprozess**): Gegeben sei ein logischer Prozessgraph G_{LP} und dessen Teilgraph G_{LP}^{crit}, der nur deadlock-kritische logische Links beinhaltet. Mit der Menge der stark zusammenhängenden Komponenten $SCC(G_{LP}^{crit})$ von G_{LP}^{crit} ist die Menge der disjunkten **Domänen** durch*

$$DOM = \{dom_0, ...dom_n\} = \{scc \in SCC(G_{LP}^{crit} : |scc| > 1)$$

gegeben. Ein logischer Prozess lp ist Teil einer Domäne dom_k, wenn eine Abbildung $domain(lp)$ existiert, sodass

$$domain(lp) = [dom_k \in DOM : lp \in dom_k]$$

gilt. lp wird dann als ***Domänenprozess*** *bezeichnet.*

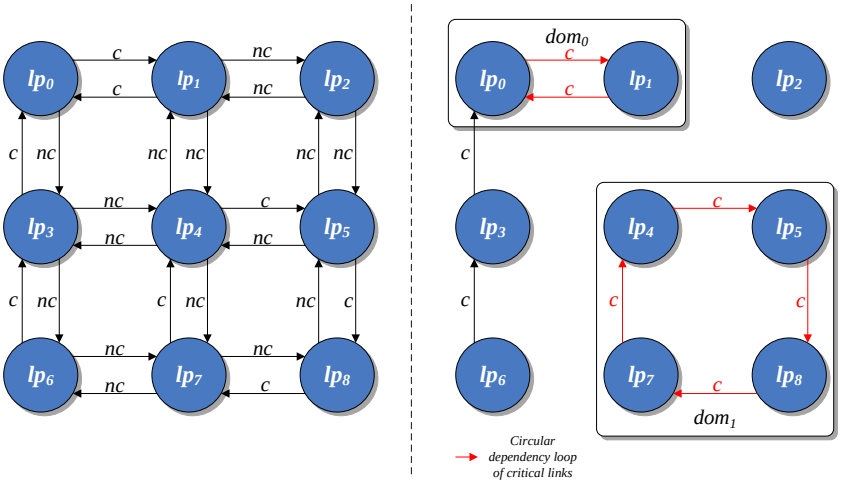


Abbildung 4.28.: G_{LP} (links) und G_{LP}^{crit} (rechts) mit zirkulären Abhängigkeiten zwischen deadlock-kritischen logischen Links

Bei Existenz von kritischen Zyklen kann ein Domänenprozess $lp_j \in dom_k$ keine Rückschlüsse mehr über die maximale Anzahl an Deltacycles ziehen, bis zu der er mit adjazenten logischen Prozessen auf der Ebene von Deltacycles synchronisieren muss, bevor er einen Zeitfortschritt bzgl. τ durchführen darf. Ausschließlich das Wissen über den lokalen Zustand eingehender logischer Links reicht in diesem Fall für die Erkennung eines validen Zeitfortschritts bzgl. τ nicht aus. Beispielsweise kann lp_4 in G_{LP}^{crit} aus Abb. 4.28 nicht wissen, ob lp_5 in einem bestimmten Deltacycle θ eine Nachricht an lp_7 verschickt hat, die lp_7 dazu veranlasst, in Deltacycle $\theta + 1$ an lp_4 eine weitere Nachricht zu senden.

Folglich ist es die Aufgabe eines Domänenprozesses, trotz kritischer Zyklen in G_{LP}^{crit} zu verhindern, dass innerhalb des Zeitschritts τ eine unendliche Anzahl an Deltacycles θ ausgeführt wird und somit ein Deadlock entsteht. Die gewählte Lösung basiert auf einer domäneninternen kollektiven Deadlockerkennung zur Laufzeit. Dazu wird ein einzelner Domänenprozess als Master auf logischer Ebene deklariert (z.B. lp_5 für dom_1 in Abb. 4.28), alle anderen als Slaves.

Sobald sich ein Slave im *lokalen Wartezustand* befindet, informiert er den Master seiner Domäne über einen aus seiner *lokalen Sicht* möglichen Zeitfortschritt bzgl.

τ. Mit Hilfe der lokalen Sichten der Slaves generiert der Master eine *domänenweite Sicht*. Mit dieser kann der Master prüfen, ob ein Zeitfortschritt bzgl. τ tatsächlich möglich ist. Die domänenweite Sicht kann z.B. anhand gemeinsam genutzter Variablen erzeugt werden. Aktuell existieren dazu für jeden Domänenprozess lp_i zwei lokal externe Datenfelder, $t_i^{sync} = (\tau_i^{sync}, \theta_i^{sync})$ und $t_i^{msg} = (\tau_i^{msg}, \theta_i^{msg})$. Im ersten Datenfeld legt ein Slave lp_i, der sich im lokalen Wartezustand befindet, seine aktuelle lokale Zeit t_i ab, in zweiten Datenfeld den Zeitstempel der zuletzt übertragenen Nachricht. Anhand dieser Information prüft der Master ob sog. *transiente Nachrichten* irgendwo in der Domäne existieren. Dies sind Nachrichten, die von einem Sender bereits verschickt worden sind, vom Empfänger allerdings nicht rechtzeitig registriert wurden, so dass dieser aufgrund des lokalen Wartezustands bereits eine Anfrage an den Master gestellt hat. Wenn keine transienten Nachrichten existieren, dann liegt der *Verklemmungszustand* vor:

Definition 4.21 ***(DSC: Domain Starvation Condition)**: Befinden sich alle logischen Prozesse einer Domäne $dom_k \in DOM$ im **lokalen Wartezustand**, so befindet sich die Domäne dom_k insgesamt im **Verklemmungszustand**, wenn folgende Bedingung erfüllt ist:*

$$\begin{aligned} &\left[\forall lp_i \in dom_k : R_i = \emptyset \wedge U_i = \emptyset \wedge N_i^{\delta} = \emptyset\right] \\ &\wedge \left[\min_{\forall lp_i \in dom_k} \{t_i^{sync}\} > \max_{\forall lp_i \in dom_k} \{t_i^{msg}\} + (0,1)\right] \end{aligned} \tag{4.17}$$

Der obere Teil von Ausdruck 4.17 definiert den lokalen Wartezustand, den jeder logische Prozess lp_i zum Zeitpunkt der Anfrage erreicht haben muss. Mit dem unteren Teil werden transiente Nachrichten ausgeschlossen. Nur wenn beide Teile von Ausdruck 4.17 erfüllt sind, teilt der Master dies den Slaves mit Hilfe eines Datenfeldes $t^{grant} = (\tau^{grant}, \delta^{grant})$ mit. In dieses schreibt er den Zeitstempel des nächsten Zeitschritts.

4.5.3. Deltazyklengenaue nachrichtenbasierte Kommunikation

Zur deltazyklengenauen Integration nachrichtenbasierter Kommunikation wird der in Abschnitt 4.4.4 beschriebene adapterbasierte Mechanismus entsprechend Abb. 4.29 erweitert. Die Behandlung einer Signalnachricht in *insert_message()* hängt wieder vom Zeitstempel der Nachricht und der lokalen Zeit des empfangenden logischen Prozesses lp_i ab:

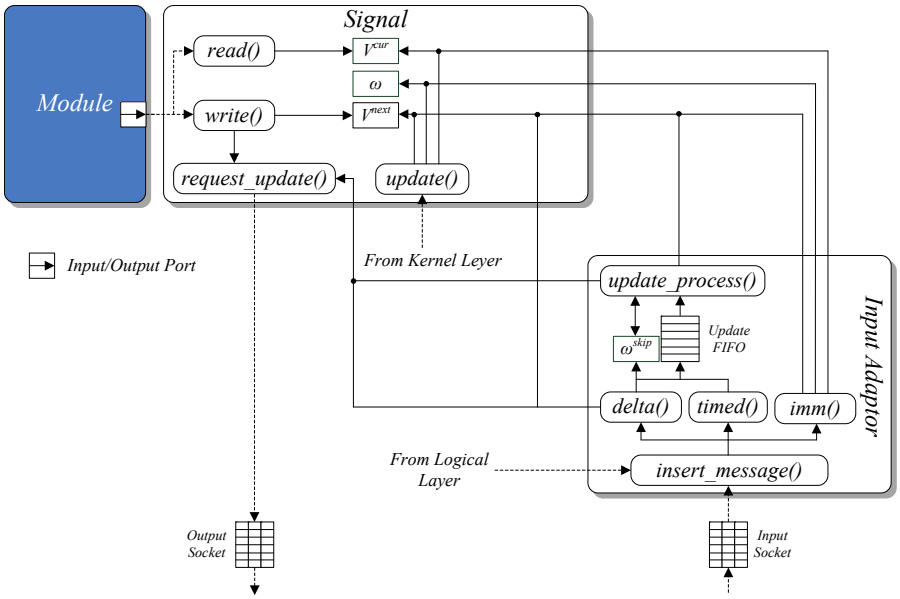


Abbildung 4.29.: Integration nachrichtenbasierter Kommunikation

1. $t^{msg} = t_i$: In diesem Fall erfolgt die Aktualisierung des zugehörigen Signals durch Aufruf von *imm()* und Erzeugung einer Immediate Notification direkt.
2. $\tau^{msg} = \tau_i \wedge \theta^{msg} > \theta_i$: In diesem Fall wird die Aktualisierung mit *delta()* um ein $\Delta\theta$ verzögert. Falls $\Delta\theta = 1$, so wird durch Aufruf von *request_update()* ein normaler E/U basierter Schreibvorgang durchgeführt. Falls $\Delta\theta > 1$, so wird die Nachricht zunächst an den *Update FIFO* weitergeleitet und dort zwischengepuffert. Die Aktualisierung wird dann mit Hilfe von *update_process()* gesteuert, welcher sensitiv auf ein Ereignisobjekt ω^{skip} ist. Mit Hilfe von ω^{skip} werden solange Überbrückungsereignisse durch Delta Notifications generiert, bis der Deltacycle θ^{top} der obersten Nachricht im *Update FIFO* erreicht ist. Dann erst wird die Aktualisierung von *update_process()* durch Aufruf von *request_update()* vorgenommen.
3. $\tau^{msg} > \tau_i$: Um die Aktualisierung (zusätzlich) bzgl. τ zu verzögern, erzeugt die Methode *timed()* in ähnlicher Weise wie im vorigen Fall ein Überbrückungsereignis in Form einer Timed Notification auf ω^{skip}.

4.5.4. Integration adaptiver Synchronisation

Zur Beschreibung der Integration des adaptiven Synchronisationsverfahrens mit einem Kernel k_i^s in einen logischen Prozess lp_i werden neben der binären Variable *pass* zunächst folgende zusätzliche Mengen definiert:

- $L^{in} = L^{dom} \cup L^{ext}$: Menge aller Eingangslinks eines logischen Prozesses.
- L^{dom}: Teilmenge aller Eingangslinks von adjazenten logischen Prozessen innerhalb der eigenen Domäne.
- $L^{ext} = L^{pa,ext} \cup L^{ub,ext} \cup L^{b,ext}$: Teilmenge der Eingangslinks von adjazenten logischen Prozessen außerhalb der eigenen Domäne.
- $L^{pa,ext}$: Teilmenge der passiven Eingangslinks von L^{ext}.
- $L^{ub,ext}$: Teilmenge der delta-unbeschränkten aktiven Eingangslinks von L^{ext}.
- $L^{b,ext}$: Teilmenge der delta-beschränkten aktiven Eingangslinks von L^{ext}.

Folgende zusätzliche Aktionen sind notwendig:

- *checkELOCC(L,*τ^{next}*,*θ^{next}*)*: Prüfe, ob in lp_i die ELOC Bedingung aus Definition 4.12 erfüllt ist. Beziehe dabei nur die durch die Menge L spezifizierten logischen Links in die Prüfung mit ein. Gib anschließend eine 1 zurück, falls die ELOC Bedingung erfüllt ist, ansonsten eine 0.
- *checkLOBC(L)*: Prüfe, ob in lp_i die LOB Bedingung aus Definition 4.16 erfüllt ist. Beziehe dabei nur die durch die Menge L spezifizierten logischen Links in die Prüfung mit ein. Gib anschließend eine 1 zurück, falls die LOB Bedingung erfüllt ist, ansonsten eine 0.
- *checkDSC()*: Prüfe, ob für alle logischen Prozesse, welche Teil der eigenen Domäne sind, die DSC aus Definition 4.21 erfüllt ist. Mit Erfüllung der DSC ist aus Sicht der logischen Links der Menge L^{dom} ein Zeitfortschritt bzgl. τ möglich. Gib eine 1 zurück, falls die DSC erfüllt ist, ansonsten eine 0.
- *dispatchOut()*: Leite alle von der lokalen Simulation generierten und evtl. im lokalen Senderpuffer (vgl. Anhang A.2) zwischengespeicherten Nachrichten an die logischen Prozesse weiter, für die die Nachrichten bestimmt sind. Falls eine Weiterleitung nicht möglich ist, rufe intern *dispatchIn()* auf und iteriere solange, bis alle Nachrichten weitergeleitet sind.
- *dispatchIn()*: Setze zunächst die Linkzeit jedes eingehenden logischen Links auf den Wert der lokal externen t^{link} Variablen des zugehörigen logischen Prozesses. Lese dann alle aktuell in den Eingangssockets der eingehenden logischen Links vorhandenen Nachrichten aus. Identifiziere das zu einer Signalnachricht gehörige Signal und füge den empfangenen neuen Signalwert v^{msg} durch Aufruf der *insert_message()* Methode auf dem zugehörigen Input Adaptor in die Simulation ein. Im Fall passiver Signale kann es vor-

kommen, dass $t^{msg} < t_i$. Dieser Fall wird genauso behandelt wie $t^{msg} = t_i$ in Abschnitt 4.5.3.

- *skip()*: Sei $t_i^{max} = \min_{\forall l_{ji} \in L_i^{in}}(t_{ji}^{link} + \Delta l_{ji}^{\tau,\theta})$. Falls $\tau_i^{max} > \tau_i$, dann setze $\theta_i = \infty$. Falls $t_i^{max} > t_i \wedge \tau_i^{max} \leq \tau_i$, dann setze $\theta_i = \theta_i^{max}$.
- *nextEdgeTime()*: Gib den Zeitpunkt τ^{edge} der nächsten Taktflanke zurück.
- *updateT()*: Aktualisiere die t_i^{link} Variable durch Kopieren des Wertes der lokalen Zeit t_i in t_i^{link}.

Abb. 4.30 illustriert den Zustandsautomaten des Verfahrens. Wegen des angepassten Zeitmodells wird der Zähler für den lokalen Deltacycle δ_i durch den synchronisierten Deltacyclezähler θ_i ergänzt. Im Vergleich zu den vorangegangenen Verfahren existiert eine weitaus komplexere Kausalitätsprüfung. Die entsprechenden Prüfaktionen $checkELOCC()$, $checkLOBC()$, $checkDSC()$ arbeiten auf den oben genannten Mengen.

Nur wenn im $s^{dnotify}$ Zustand keine lauffähigen Prozesse mehr vorhanden sind, ist u.U. ein Zeitfortschritt bzgl. τ möglich (lokaler Wartezustand). Dazu müssen ELOCC, LOBC und DSC für relevante logische Links erfüllt sein. Rein passive logische Links können bei der Prüfung auf einen Zeitfortschritt grundsätzlich vernachlässigt werden (vgl. Abschnitt 4.5.2.3).

Im s^{ELOCC_INFTY} Zustand wird *checkELOCC()* nur auf der Teilmenge $L^{ub,ext}$ ausgeführt. Diese Links sind zwar allesamt kritisch, da $L^{ub,ext} \cap L^{dom} = \emptyset$, sind sie allerdings nicht Teil eines Zyklus der eigenen Domäne. Damit ist garantiert, dass die Linkzeit dieser Links vom Sender nach einer endlichen Anzahl Deltacycles bzgl. τ inkrementiert wird. Daher kann die Prüfung von ELOCC in s^{LOCC_INFTY} mit $\theta = \infty$ erfolgen. Ist die ELOCC in s^{LOCC_INFTY} erfüllt, wird die LOBC in s^{LOBC} auf der Teilmenge $L^{b,ext}$ geprüft. Ist diese auch erfüllt, wird zuletzt in s^{DSC} die DSC auf der Teilmenge L^{dom} geprüft. Sind alle drei Bedingungen erfüllt, wird der Zeitfortschritt durchgeführt.

Bevor der nächste Timed- oder Deltacycle tatsächlich ausgeführt werden darf, muss in s^{ELOCC} noch einmal die ELOCC auf allen Eingangslinks L^{in} geprüft werden. Dies dient nicht mehr einem kausal korrekten Zeitfortschritt [11], sondern ausschließlich dem vollständigen und damit deterministischen Empfang aller Nachrichten für den nächsten Zyklus.

4.5.5. Abbildung auf die Speicherarchitektur SCC

Die Methode der Abbildung auf den SCC ist weitgehend identisch zur Methode, die bei der symmetrischen asynchronen Strategie aus Abschnitt 4.4 verwendet

[11] Der Zeitfortschritt ist mit dem Ausführen von *tnotify()* bereits abgeschlossen.

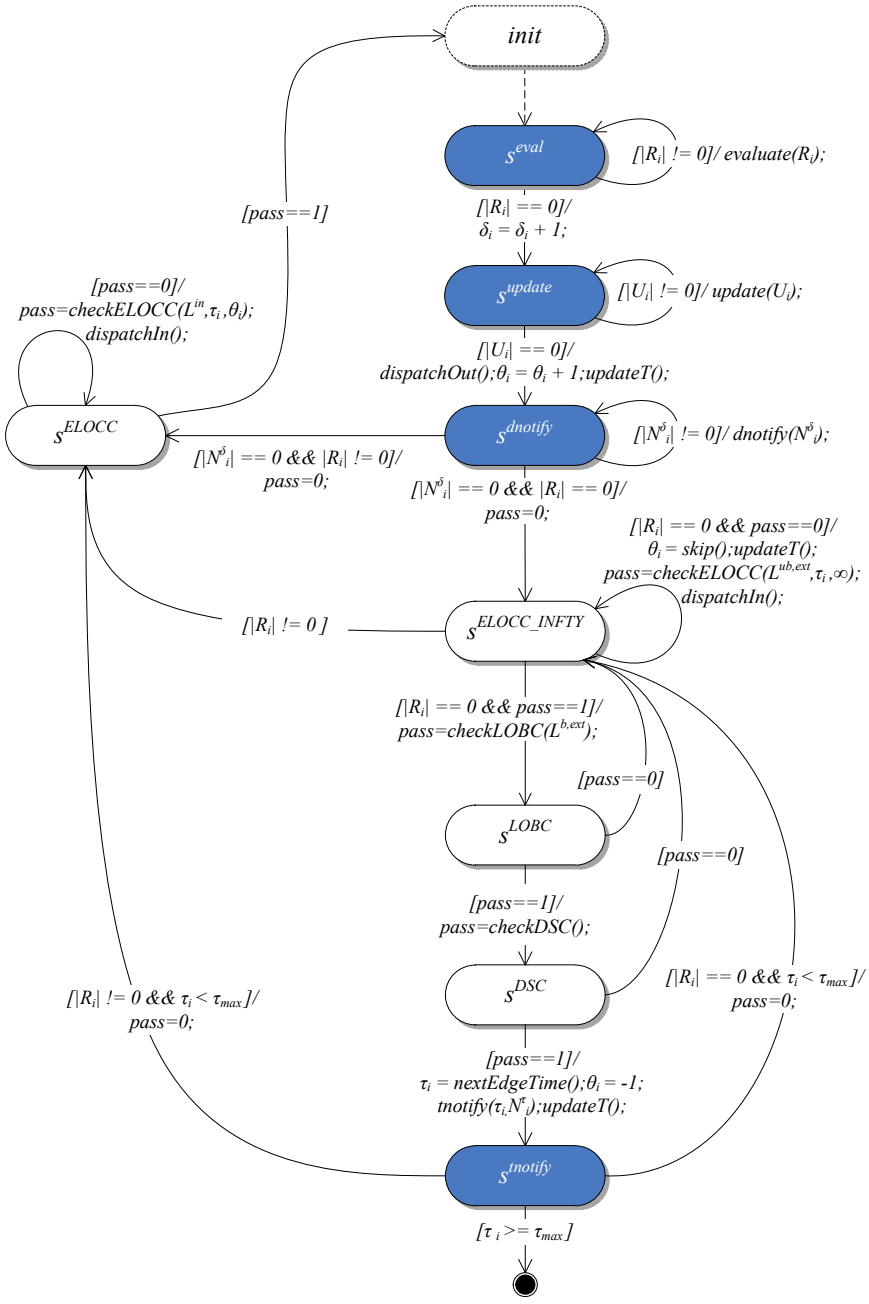


Abbildung 4.30.: Adaptive Zustandsmaschine in einer Kernelkomponente k_i^S

wurde. Anstelle der τ_i^{link} Variablen, die mit Hilfe von *Shared Object* Primitiven im MPB allokiert wurden, existieren nun im MPB *Shared Object* Instanzen für die t_i^{link}, t_i^{sync}, t_i^{msg} und t^{grant}. Abb. 4.31 illustriert die Verteilung auf die Speicherarchitektur des SCC. Kernel k_2^s fungiert im Beispiel als Master, die anderen Kernel als Slaves.

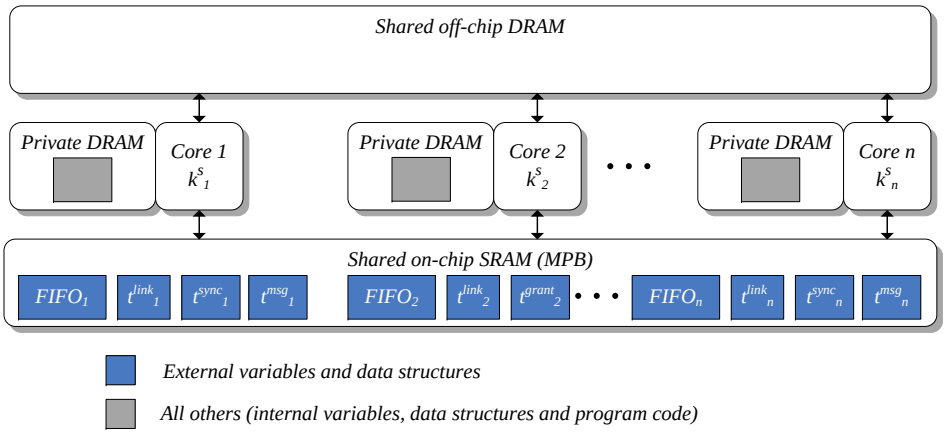


Abbildung 4.31.: Speichernutzung bei adaptiver Synchronisation

4.5.6. Vollautomatisierte Werkzeugunterstützung

Die vollautomatisierte Werkzeugkette wurde speziell für den adaptiven parallelen SystemC Kernel entwickelt. Volle Automatisierung bedeutet, dass Schritt II der Simulationssynthese aus Abschnitt 4.2.3 komplett versteckt vor dem Modellersteller erfolgt. Dies ist insofern wünschenswert, als dass ein solcher Ansatz unabhängig vom Simulationsmodell ist. Er ist daher unmittelbar auch auf andere RTL u.ä. Modelle anwendbar. Vollautomatisierung ist darüber hinaus notwendig, weil eine manuelle Modellanalyse und Konfiguration, wie sie im Fall des Kernels aus Abschnitt 4.4 noch ausreichend war, im Hinblick auf die komplexen Zusammenhänge zwischen Modelleigenschaften und Synchronisationsmethoden als nicht mehr handhabbar betrachtet werden kann. Ein weiterer Vorteil ist die Aufhebung der Partitionierungsbeschränkung auf dem Toplevel, wodurch Modelle flexibel verteilt werden können.

4.5.6.1. Überblick

Der gesamte Ablauf ist in Abb. 4.32 illustriert. Da der HeMPS Editor infolge der vollen Automatisierung nicht mehr modifiziert werden muss, kann für die Generierung des HeMPS Simulationsmodells der originale Editor verwendet werden. Als Voraussetzung für die Abbildung des Modells auf eine parallele Ausführungsplattform muss dann zunächst eine Modellanalyse erfolgen. Anhand der Analyse werden relevante Eigenschaften des Simulationsmodells identifiziert und gespeichert. Auf Basis dieser gespeicherten Informationen kann die parallele Simulation, bestehend aus Modell und parallelem Simulator, für die Ausführung auf der Zielplattform konfiguriert werden.

Die vorgelagerte Modellanalyse gliedert sich in einen statischen und einen dynamischen Teil. Die statische Analyse dient der Extraktion von deklarativer Information und Aspekten, die das Modellverhalten betreffen. Sie nutzt dazu das Compiler Frontend Clang [88], das Teil der LLVM Compiler Infrastruktur [186] ist. Mit [158] existiert seit Kurzem ein weiteres Werkzeug auf Basis von Clang für die Analyse von SystemC Modellen, das ungefähr im gleichen Zeitraum zu dieser Arbeit entstanden ist. In [158] wird allerdings nur eine rein statische Analyse unterstützt. Da sich die statische Analyse in dieser Arbeit auf die Extraktion deklarativer Informationen beschränkt, kann sie deutlich einfacher aufgebaut werden. Die restlichen Informationen werden in der dynamischen Analyse gewonnen.

Die dynamische Analyse dient der Extraktion von Information, die zur Laufzeit unmittelbar verfügbar ist und schwer oder gar nicht statisch extrahiert werden kann. Dazu gehört beispielsweise die vollständige parametrisierte Struktur der Instanz der modellierten Hardwarearchitektur. Außerdem integriert die dynamische Analyse direkt ein dynamisches Profiling, das zur Optimierung der Partitionierung genutzt wird. Die dynamische Analyseroutine kann nativ in die ausführbare Datei der Simulation integriert werden, sodass die Analyse, anders als bei [193], ohne virtuelle Maschine und Just-in-Time Kompilierung durchgeführt werden kann. Durch die Kombination aus statischer und dynamischer Analyse erhält man ein möglichst vollständiges Abbild des Simulationsmodells. Im Folgenden werden die statische und die dynamische Analyse näher betrachtet.

4.5.6.2. Statische Analyse

Ausgehend vom ursprünglichen Modell wird zunächst die statische Analyse durchgeführt (siehe Abb. 4.32). Diese besteht aus der Quellcodeanalyse selbst (*Generate SAR*), in der die SystemC-spezifische *Static Abstract Representation (SAR)* generiert wird und einer Codegenerierungsphase (*Generate Extended Model Code*).

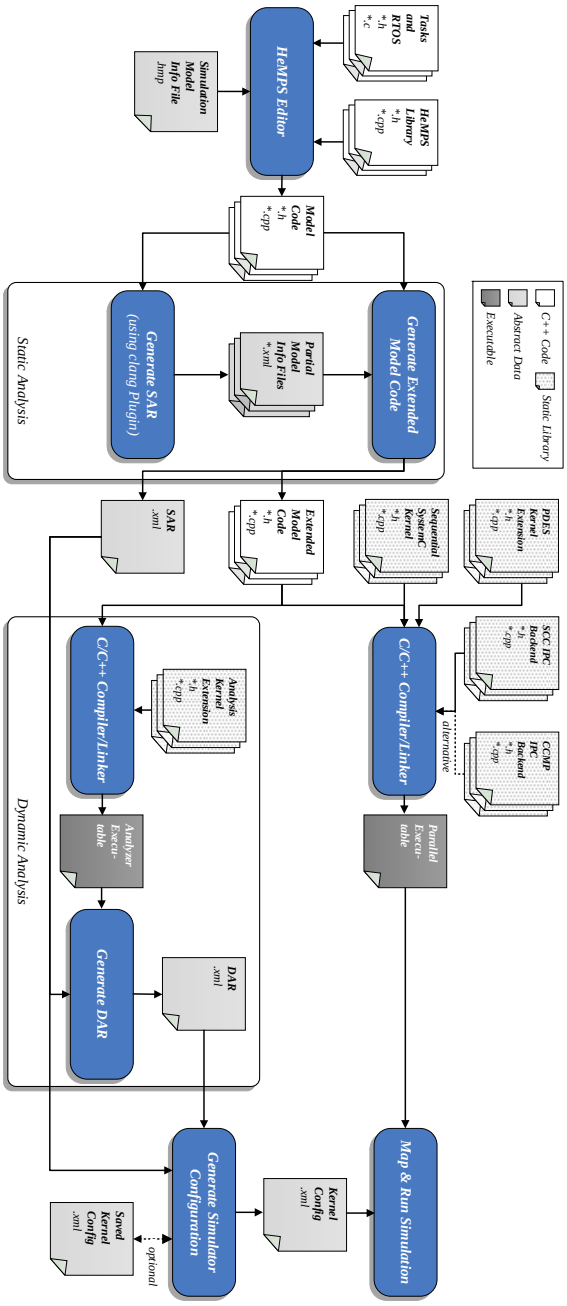


Abbildung 4.32.: Vollautomatisierte Werkzeugkette

Generierung der statischen abstrakten Darstellung

Die statische Quellcodeanalyse nutzt das bereits erwähnte Clang LLVM Compiler Frontend. Clang bietet eine Programmierschnittstelle zur Integration von Plug-ins und bildet damit eine flexible Grundlage für die Implementierung der Software zur Quellcodeanalyse, auch im Hinblick auf zukünftige Erweiterungen.

Ein Clang Plug-in arbeitet auf dem *Abstract Syntax Tree (AST)* des eingelesenen Quellcodes. Dieser ist eine vollständige Darstellung der im C++ Quelltext vorgefundenen Sprachkonstrukte. Eine vollständige Darstellung ist notwendig, da der AST den Ausgangspunkt für alle weiteren Schritte zur Erzeugung von Maschinencode bildet. Im Clang AST sind jedoch bereits die Ergebnisse einiger Verarbeitungsschritte wie der Namensauflösung und der Klassifikation von Ausdrücken und Deklarationen enthalten. Zur Speicherung der Sprachkonstrukte basiert der Clang AST auf einer geeigneten Datenstruktur.

Neben der Datenstruktur zur Speicherung des AST stellt Clang eine Programmierschnittstelle zur Verfügung, anhand derer innerhalb eines Plug-ins über Typen von Deklarationen und Ausdrücken im AST iteriert werden kann. Mit Hilfe dieser Schnittstelle werden folgende Informationen extrahiert und in der SystemC-spezifischen SAR gespeichert:

- **Modul-Deklarationen**: Für jede gefundene Moduldeklaration wird deren Position in der Quelldatei für die spätere Code-Modifikation identifiziert.
- **Ports und Channels**: Alle innerhalb von Modulen deklarierten SystemC Ports und Channels werden ermittelt.
- **Prozesse**: Definitionen von SystemC Prozessen innerhalb von Modulkonstruktoren, deren Typ (z.B. *SC_THREAD* oder *SC_METHOD*) sowie deren Sensitivitätslisten werden aktuell identifiziert. Auch Aufrufe der *wait()* und *next_trigger()* Methoden innerhalb von Prozessen werden ermittelt. Diese können entweder eine dynamische Sensitivität verursachen oder eine Timed-/Delta Notification erzeugen, wodurch der den Aufruf beinhaltende Prozess zu einem späteren Zeitpunkt wieder aktiviert wird. Im ersten Fall wird die Notification wie ein Eintrag in der Sensitivitätsliste behandelt. Im zweiten Fall wird der Prozess als Quelle für Notifications markiert.
- **Lese- und Schreibzugriffe**: Voraussetzung für die Extraktion sämtlicher Lese- und Schreibzugriffe auf Channels oder Ports innerhalb von identifizierten SystemC Prozessen ist, dass die verwendeten Port bzw. Channel Variablen direkt aus dem Quellcode ermittelt werden können. Da bei der Verwendung von Zeigern die verwendete Variable im Allgemeinen erst zur Laufzeit feststeht, dürfen solche Zugriffe im Modell aktuell nicht auftreten. Im Fall von Arrays aus Ports oder Channels können variable Indizes verwendet werden. Das Analysewerkzeug geht dann jedoch von einem Zugriff auf alle Einträge des Arrays aus.

Die durch die SAR extrahierten Informationen sind in Abb. 4.33 anhand eines *Unified Modeling Language (UML)* Klassendiagramms dargestellt. Da das Clang Plug-in in den Compiler eingebunden ist, wird die Analyse für jede Kompilier-Einheit und damit für jede Quelldatei einzeln ausgeführt. Als Ausgabe sind nach einem Analysedurchlauf über die verschiedenen Quelldateien mehrere XML (Extensible Markup Language) Dateien vorhanden, über die die extrahierte SAR Information modulweise verteilt gespeichert ist. In Abb. 4.32 werden diese Dateien als *Partial Model Info Files* bezeichnet.

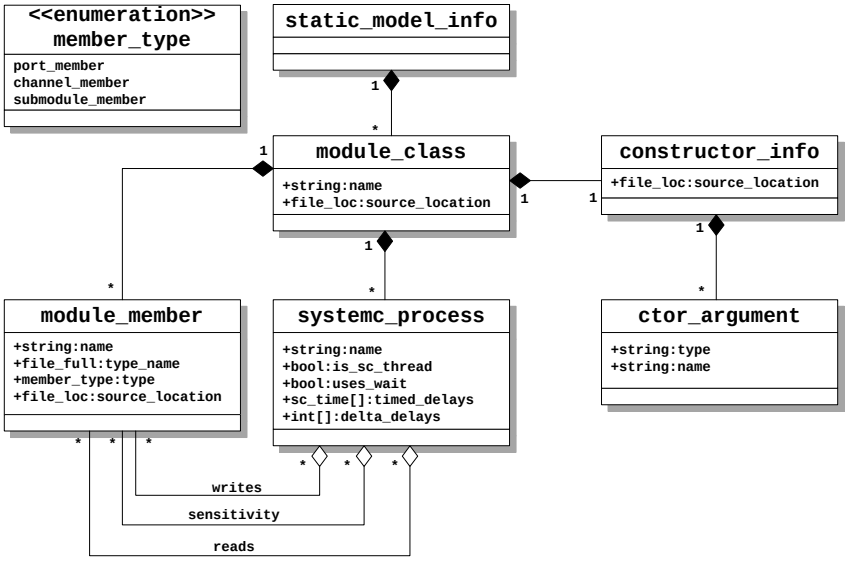


Abbildung 4.33.: Klassendiagramm der statischen abstrakten Darstellung (Quelle: [Red14])

Generierung des erweiterten Modellcodes

Die mit dem Clang Plug-in erzeugten XML Dateien dienen als Eingangsdaten für das Werkzeug zur Generierung des erweiterten Modellcodes (*Generate Extended Model Code* Block in Abb. 4.32). Dieses fasst zunächst die statischen Informationen, die auf die verschiedenen Partial Model Info Files verteilt sind, zusammen und speichert sie in einer einzelnen Datei namens *SAR.xml*. Die extrahierte SAR wird anschließend zur Generierung der folgenden drei Artefakte verwendet:

- **Modulwrapper**: Für jedes Modul der gesamten Modulhierarchie wird ein Wrapper generiert. Wie bei den Wrappern aus Abschnitt 4.4.6 ist eine bedingte Instanziierung von gekapselten Moduls anhand von Flags und ei-

ne Modellverteilung mit Modulgranularität möglich. Im Unterschied zu den Wrappern aus Abschnitt 4.4.6 ist die Verteilung allerdings nicht mehr auf das Toplevel beschränkt. Module niedrigerer Hierarchieebenen können unabhängig von Modulen höherer Hierarchieebenen auf unterschiedliche Kerne der Ausführungsplattform verteilt werden. Eine Erweiterung von Modul- auf Prozessgranularität ist leicht integrierbar: Dazu müsste ein Modul auf verschiedenen Kernen mehrfach instanziiert werden. Die Prozesse innerhalb des Moduls müssten dann selektiv ausgeführt werden.

- **Helfermethoden und Helferklassen**: Durch die Helfermethoden und Helferklassen können statisch gewonnene und in der SAR hinterlegte Informationen in der nachfolgenden dynamischen Analyse oder Ausführung verwendet werden. Die Helfermethoden sind für eine typspezifische Inspizierung von SystemC Modulen in der dynamischen Analyse notwendig. Die Helferklassen werden für die dynamische Instanziierung der Modulwrapper und SystemC Channels während der Laufzeit benötigt, wenn das übergeordnete Modul auf einem anderen Kern ausgeführt wird.
- **Modifikation der Header**: Die Modifikation der originalen Quelltexte beschränkt sich auf das Hinzufügen von Typdefinitionen und Präprozessorkonstrukten, wodurch die originalen Module durch die Modulwrapper ausgetauscht werden. Da der in Abschnitt 4.5.3 beschriebene Kommunikationsmechanismus direkt in die Basisklasse für Primitive Channels integriert wurde, werden notwendige Modifikationen am originalen Quelltext insgesamt auf ein Minimum reduziert.

4.5.6.3. Dynamische Analyse

Während der dynamischen Analyse werden alle Aspekte extrahiert, die von einer konkreten Modellausführung abhängen und in der sog. *Dynamic Abstract Representation (DAR)* gespeichert. Dies hat den Vorteil, dass jegliche Parametrierung, die von einem bestimmten Simulationslauf abhängt, automatisch erkannt werden kann. Abb. 4.34 illustriert das einer DAR zugrundeliegende Klassendiagramm. Um die DAR zu erzeugen, muss der im Rahmen der statischen Analyse generierte/modifizierte Quellcode zunächst kompiliert und mit den für die Analyse notwendigen Bibliotheken gelinkt werden. Letztere beinhalten den sequentiellen SystemC Kernel sowie eine sog. *Analyzer Kernel Extension*, die sämtlichen für die dynamische Analyse notwendigen Analysecode beinhaltet (siehe Abb. 4.32).

Als Ergebnis erhält man das *Analyzer Executable*, durch dessen Ausführung die dynamische Analyse erfolgt. Dabei wird die Elaboration Phase des SystemC Kernels zur Extraktion der simulierten Architektur und die Simulation Phase zur Extraktion von Profilinginformation genutzt.

Extraktion der Modellarchitektur

In der Elaboration Phase werden Objekte der Module, Ports und Channels eines SystemC Designs instanziiert und die Verknüpfungen zwischen diesen hergestellt. Mit dem Ende der Elaboration Phase ist das komplette Modell instanziiert und bereit zur Ausführung. Der SystemC Kernel speichert dabei die genannten SystemC Objekte in einer internen Datenstruktur. Diese Datenstruktur wird zur Iteration über die Objekte und zu deren Inspektion genutzt.

Da die Objekte in der Datenstruktur nur durch Zeiger auf polymorphe Basisklassen (*sc_module*, *sc_prim_channel* oder *sc_port_base*) referenziert werden (vgl. [27]), müssen für die Extraktion der Verknüpfungen von Objekten zunächst die spezialisierten Datentypen der einzelnen Objekte identifiziert werden. Dazu werden die während der statischen Analyse generierten Helfermethoden eingesetzt (vgl. Abschnitt 4.5.6.2).

Mit Unterstützung der Helfermethoden werden die Klassennamen der Modulinstanzen und die Membervariablen von Modulen inklusive Typ und Name erkannt. Anhand der Namen von Modulklassen und Membervariablen können den einzelnen Instanzen dann die zugehörigen Daten aus der statischen Analyse zugeordnet werden. Darauf basierend ist es schließlich möglich, Verbindungsstrukturen zwischen Prozessen, Port- und Channelinstanzen zu identifizieren, indem die an Ports angebundenen Channelinstanzen ermittelt und Zugriffe von Prozessen auf Channels und Ports den jeweiligen Instanzen zugeordnet werden.

Die Helfermethoden setzen sich aus einer Methode namens *get_module_members()* und einer Menge von überladenen Methoden namens *get_members()* zusammen. Von letzteren existiert jeweils eine Version für jede Modulklasse des Modells. Innerhalb von *get_module_members()* wird der Klassenname einer Modulinstanz über deren dynamischen Typ mit Hilfe von Typecasts bestimmt. Wurde der Typ identifiziert, so können mit der zugehörigen *get_members()* Methode alle im Modul instanziierten Member wie Ports, Channels inklusive Typ und Name ermittelt werden.

Insgesamt wird mit den extrahierten Informationen eine abstrakte Beschreibung der dynamisch erzeugten Modellstruktur generiert. Diese wird mit Hilfe der Klassenstruktur aus Abb. 4.34 abgespeichert. Die Beschreibung umfasst die Modulhierarchie mit allen Ports, Channels und Prozessen und deren Relationen. Speziell für *sc_clock* Channels werden aktuell zusätzlich noch Informationen über deren Zeitverhalten erfasst und für die Nutzung in den nachfolgenden Analyseschritten abgespeichert.

Extraktion von Profilingdaten

In der Simulation Phase werden für SystemC Prozesse und für SystemC Channels Profilinginformationen aufgezeichnet. Für jeden SystemC Prozess $p_i \in P$ wird die Anzahl der Ausführungen $N_i^{eval}(\theta)$ und die mittlere Ausführungszeit

T_i^{eval} in Abhängigkeit des Deltacyclezählers θ erfasst. In der Implementierung muss mit θ^{max} eine Grenze festgelegt werden, bis zu der $N_i^{eval}(\theta)$ maximal bestimmt werden soll. Mit Hilfe von

$$\bar{n}_i^{eval}(\theta) = \begin{cases} N_i^{eval}(\theta), & \theta < \theta^{max} \\ \sum_{k=\theta^{max}}^{\infty} N_i^{eval}(k), & \theta = \theta^{max} \end{cases} \tag{4.18}$$

kann für jeden SystemC Prozess p_i ein Vektor $\vec{n}_i^{eval}$ angegeben werden, der die Ausführungshäufigkeiten für $\theta \leq \theta^{max}$ wie folgt annähert:

$$\vec{n}_i^{eval} = \left[\bar{n}_i^{eval}(0), \bar{n}_i^{eval}(1), ..., \bar{n}_i^{eval}(\theta^{max})\right]^T \tag{4.19}$$

Auf Basis von $\vec{n}_i^{eval}$ wird der Lastvektor $\vec{w}_i^{load}$ definiert, welcher die Verteilung der mittleren Ausführungszeit eines SystemC Prozesses p_i auf die Deltacycles beschreibt. Für $\vec{w}_i^{load}$ gilt:

$$\vec{w}_i^{load} = T_i^{eval} \times \begin{pmatrix} \hat{n}_i^{eval} \\ \vec{n}_i^{eval} \end{pmatrix}. \tag{4.20}$$

Dabei ist die Gesamtzahl der Evaluationen $\hat{n}_i^{eval}$ eines Prozesses p_i näherungsweise durch

$$\hat{n}_i^{eval} = \sum_{\theta=0}^{\theta^{max}} \bar{n}_i^{eval}(\theta) \tag{4.21}$$

gegeben. Neben den Profilingdaten für SystemC Prozesse wird aktuell für jeden Channel $ch_j \in CH$ die Gesamtanzahl der Aktualisierungen N_j^{update} in der Update Phase aufgezeichnet.

Die Profilingdaten werden, neben den weiter oben bereits genannten Informationen über das Modell, ebenfalls in der DAR hinterlegt. Die vollständige DAR kann abschließend in einer XML Datei namens *DAR.xml* exportiert werden. DAR und SAR liefern somit ein Abbild des Modells, das Verhalten, Struktur und Per-

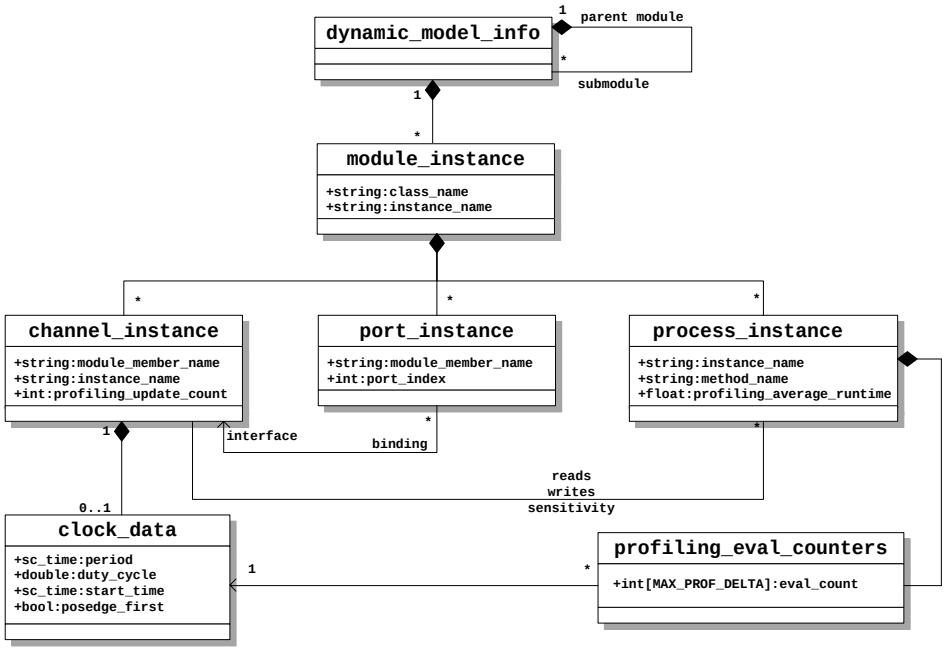


Abbildung 4.34.: Klassendiagramm der dynamischen abstrakten Darstellung (Quelle: [Red14], erweitert)

formanzinformation beinhaltet. Diese Informationen dienen als Grundlage für die Modellpartitionierung und die Kernelkonfiguration im nun folgenden Schritt.

4.5.6.4. Abbildung auf die Ausführungsplattform

Die Abbildung des Simulationsmodells auf die Ausführungsplattform geschieht mit Hilfe eines gesonderten Werkzeugs. In Abb. 4.32 wird dieses Werkzeug durch den Block *Generate Simulator Configuration* repräsentiert. Der Prozess der Abbildung lässt sich in die drei Teilschritte Vorverarbeitung, Partitionierung und Nachverarbeitung gliedern.

Vorverarbeitung
Das Werkzeug liest zunächst die XML Dateien ein, welche die statische und dynamische abstrakte Darstellung enthalten und erzeugt daraus den Prozess-Signal-Graphen $G_{PS}(P, S)$ und den Sensitivitätsgraphen $G_S(P, E)$ des Simulationsmodells. Auf Basis einer Analyse von G_S werden Knoten P aus G_{PS} mit Aktivitätsmustern (vgl. Abschnitt 4.5.2.4) annotiert. Knoten P und Kanten S werden zudem mit Profilingdaten (vgl. Abschnitt 4.5.6.3) aus der DAR annotiert.

Partitionierung
Ausgehend vom Graphen G_{PS} wird der logische Prozessgraph $G_{LP}(LP, L)$ synthetisiert. Letzterer entspricht mit Definition 4.4 einer Partitionierung aller vorhandenen SystemC Prozesse $p \in P$ in logische Prozesse $lp \in LP$.

Aktuell existiert für die Partitionierung die zusätzliche Nebenbedingung, dass alle SystemC Prozesse einer Modulinstanz in derselben Partition liegen müssen (vgl. Abschnitt 4.5.6.2). Diese Bedingung kann dadurch eingehalten werden, dass Prozesse derselben Modulinstanz vor der eigentlichen Partitionierung zu einem Knoten zusammengefasst werden. Damit erhält man ein Graphenpartitionierungsproblem ohne Nebenbedingungen, für dessen Lösung bereits viele unterschiedliche Algorithmen existieren. Geeeignete Lösungsalgorithmen sind in der METIS Bibliothek [155] implementiert, weshalb METIS in dieser Arbeit verwendet wird. METIS wurde speziell für die Partitionierung unregelmäßiger Graphen oder großer Mesh-Netzwerke entwickelt. Die Algorithmen basieren auf dem Paradigma der sog. Multilevel Graphpartitionierung [156, 157].

Als Eingabedaten benötigt die Bibliothek Gewichte für die Knoten und Kanten des zu partitionierenden Graphen. Der Partitionierungsalgorithmus versucht dann, die Knoten so auf die Partitionen zu verteilen, dass die Summe aller Knotengewichte in den einzelnen Partitionen möglichst nahe an eine vorgegebene Verteilung heran kommt. Gleichzeitig wird versucht die Summe der Gewichte aller Kanten, die zwischen zwei Partitionen verlaufen, zu minimieren.

Als Knotengewichte können in METIS Vektoren verwendet werden. Dies ist die Basis für eine Optimierung bzgl. eines jeden Vektorelements (Mehrzieloptimierung). Für einen SystemC Prozess werden die an die Knoten $p_i \in G_{PS}$ annotierten Lastvektoren $\vec{w}_i^{load}$ als vektorielle Knotengewichte verwendet. Daneben kann ein Toleranzvektor $\vec{\vartheta}$ definiert werden, der das erlaubte Ungleichgewicht bzgl. einzelner Elemente des Knotengewichtsvektors zwischen Partitionen spezifiziert [155]. Als Kantengewicht für eine Kante $s_j \in G_{PS}$ wird unmittelbar die Summe aus der Anzahl der Aktualisierungen N_k^{update} all der SystemC Channels ch_k verwendet, die durch die Kante repräsentiert sind.

Nachverarbeitung

Ist eine Partitionierung gefunden, werden mit Hilfe der Klassifikation aus Abschnitt 4.5.2) und der Graphen G_{PS} und G_S die Eigenschaften der logischen Links und deren Kritikalitätstypen extrahiert. Als Ergebnis erhält man G_{LP}^{crit}. Über die Suche von stark zusammenhängenden Komponenten [152] in G_{LP}^{crit} werden Domänen identifiziert und logische Prozesse in Master und Slaves eingeteilt. Schlussendlich wird die komplette Konfiguration inkl. Modellpartitionierung, Aktivitätsmuster, Position von Domänen und die Master/Slave Einteilung logischer Prozesse als XML Datei (*KernelConfig.xml*) exportiert. Das Werkzeug ist außerdem in der Lage, eine vereinfachte Repräsentation der generierten Partitionierung zu exportieren (*SavedKernelConfig.xml* Datei). Die abgespeicherte Partitionierung kann daraufhin gegebenenfalls modifiziert und bei einem späteren Aufruf des Werkzeugs wieder importiert werden. Dadurch ist es möglich, auch manuell erstellte oder modifizierte Partitionierungen anstelle einer automatisch generierten Partitionierung zu verwenden.

Ausführung

Für die parallele Ausführung muss zunächst das *ParallelExecutable* erzeugt werden. Dazu muss der in Abb. 4.32 oberhalb zur dynamischen Analyse parallel verlaufende Pfad durchschritten werden. Das *ParallelExecutable* wird dann unter Verwendung der zuvor generierten *KernelConfig.xml* Datei gestartet. Diese konfiguriert das *ParallelExecutable* zur Simulationslaufzeit mit den vorher erstellten Konfigurationsdaten. Mit Hilfe unterschiedlicher Backends ist eine Ausführung dabei sowohl auf dem SCC als auch auf gewöhnlichen cachekohärenten SHM Multiprozessoren möglich.

4.5.7. Bewertung

4.5.7.1. Adaptive Synchronisation und Lastverteilung

In diesem Abschnitt soll untersucht werden, welchen Einfluss die adaptive Synchronisation auf die Beschleunigung der Simulation hat. Dies beinhaltet eine Betrachtung des Einflusses der dynamischen Verklemmungserkennung als auch einen Vergleich mit einem rein synchronen Verfahren.

Für die Messungen wurden für verschieden große HeMPS Modelle unterschiedliche Partitionierungen manuell erstellt. Die HeMPS Modelle führten dabei die bereits bekannte MPEG Applikation aus. Die gewählten Partitionierungen waren frei von Zyklen kritischer logischer Links, so dass eine dynamische Erkennung von Verklemmungen und die Einführung von Domänen prinzipiell eigentlich nicht notwendig war. Ausgehend von diesem Basisfall wurden die logischen Prozesse anschießend dennoch auf gleich große *künstliche* Domänen aufgeteilt. Innerhalb dieser Domänen blieb die Verklemmungserkennung aktiviert. Beginnend bei einer maximalen Anzahl von drei logischen Prozesen pro Domäne wurde die Anzahl an logischen Prozessen innerhalb einer Domäne sukzessive erhöht. Im Umkehrschluss reduzierte sich dadurch die Gesamtanzahl vorhandener Domänen. Im letzten Schritt befanden sich dann alle logischen Prozesse in einer einzigen Domäne. Dies entspricht einer globalen Synchronisation für einen Zeitschritt bzgl. τ.

Für den ersten Teil des Experiments wurden ein 8x8 und ein 6x8 HeMPS RTL Modell manuell in jeweils 48 Partitionen aufgeteilt. Da das 6x8 Modell aus genau 48 Tiles besteht, konnten alle Submodule eines Tiles jeweils in einer separaten Partition zusammengefasst waren. Weil alle Partitionen einen Satz gleichartiger Modulinstanzen enthielten, war die Partitionierung des 6x8 Modells homogen. Bei der Partitionierung des 8x8 Modells enthielten 32 Partitionen je ein Processing Element und die restlichen 16 Partitionen je vier Router und vier Network Interfaces. Aufgrund dieser ungleichen Zusammensetzung war die Partitionierung inhomogen. Abb. 4.35 illustriert die Messergebnisse[12].

Am auffälligsten an den Messergebnissen ist die Tatsache, dass die Beschleunigung der Simulation des 6x8 Modells trotz der geringeren Modellgröße generell deutlich höher ausfällt als die Beschleunigung der Simulation des 8x8 Modells. Die Ursache ist die weniger gleichmäßige Verteilung der Rechenlast bei inhomogener Partitionierung. Dadurch kann es mit höherer Wahrscheinlichkeit zu permanent größeren Differenzen in der Ausführungszeit von Deltazyklen auf unterschiedlichen logischen Prozessen kommen.

[12] Um statistische Schwankungen auszugleichen, wurden die Messungen mehrfach wiederholt und Mittelwerte gebildet.

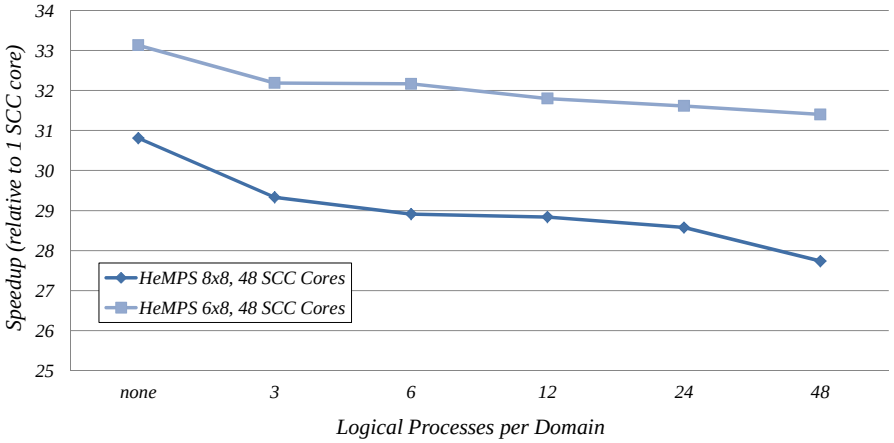


Abbildung 4.35.: Beschleunigung bei fester Partitionierung und variabler Domänengröße

Des Weiteren ist zu erkennen, dass die Beschleunigung mit zunehmender Domänengröße abnimmt. Der größte Abfall ist bei maximaler Domänengröße zu verzeichnen. Ein Grund für den Abfall ist der durch die dynamische Verklemmungserkennung verursachte zusätzliche Synchronisationssaufwand: Innerhalb der künstlichen Domänen entstehen nicht notwendige domänenweite Wartezeiten. Logische Prozesse müssen beim Zeitfortschritt u.U. auf andere logische Prozesse warten, obwohl keine kausalen Abhängigkeiten bestehen.

Bei inhomogener Partitionierung (8x8 Modell) fällt der relative Verlust, d.h. die relative Verlängerung der Ausführungsdauer, bei Verwendung einer globalen Domäne mit 11.07% nahezu doppelt so stark aus wie bei homogener Partitionierung (6x8 Modell) mit 5,95%. Die auftretenden Wartezeiten werden offensichtlich durch die ungleiche Lastverteilung weiter vergrößert, da die schnelleren logischen Prozesse länger auf die langsameren warten müssen.

Tab. 4.2 beinhaltet einige zusätzliche Messergebnisse, um den Zusammenhang zwischen der Lastverteilung und dem Verlust durch globale Synchronisation weiter zu verdeutlichen. Sie zeigt für verschiedene Modellkonfigurationen (ein 2x2, ein 4x4 und die beiden bekannten Modelle) den Verlust, der durch eine globale Domäne und durch vollständig synchrone Ausführung entsteht[13].

Wie erwartet verursacht eine globale Domäne weiterhin bei den inhomogenen Partitionierungen stärkere Verluste. Im Unterschied dazu ist bei voll synchroner Simulation nicht die Homogenität der Partitionierung der dominierende Ein-

[13] Die Prozentangaben beziehen sich auf die Ausführungsdauer. Angegeben ist jeweils das Verhältnis zur Ausführungsdauer bei adaptiver Synchronisation ohne künstliche Domänen.

flussfaktor. Hier ist eher ein Zusammenhang zwischen der Modellgröße/SCC Kernen und dem Verlust erkennbar: Letzterer steigt mit der Anzahl an SCC Kernen. Bei gleichbleibender Anzahl an SCC Kernen (48) sinkt er mit der Modellgröße. Die Verluste sind bei synchroner Ausführung zudem insgesamt deutlich größer als im Fall von asynchroner Ausführung und einer globalen Domäne.

Tabelle 4.2.: Laufzeitverlust durch globale Synchronisation

Modellgröße/SCC Kerne	2x2/9	4x4/16	6x8/ 48	8x8/48
Homogene Partitionierung	nein	ja	ja	nein
Verlust durch globale Domäne	9.47 %	2.87 %	5.95 %	11.07 %
Verlust durch synchrone Ausführung	24.94 %	36.05 %	76.05 %	67.07 %

4.5.7.2. Feingranulare Partitionierung

Bei der teilautomatisierten Werkzeugkette aus Abschnitt 4.4.8 ist die Partitionierung auf die oberste Ebene der Modulhierarchie beschränkt. Dies limitiert die maximale Anzahl logischer Prozesse. Beispielsweise können im Fall eines 2x2 HeMPS Modells bestehend aus zwölf Toplevel Modulen (PEs, NIs und ROs) maximal zwölf logische Prozesse erzeugt werden. Da die Processing Elements den Hauptanteil der Rechenlast erzeugen, kann eine weitere Partitionierung dieser Module zu zusätzlichen Performanzverbesserungen beitragen. Dies ist in Abb. 4.36 illustriert.

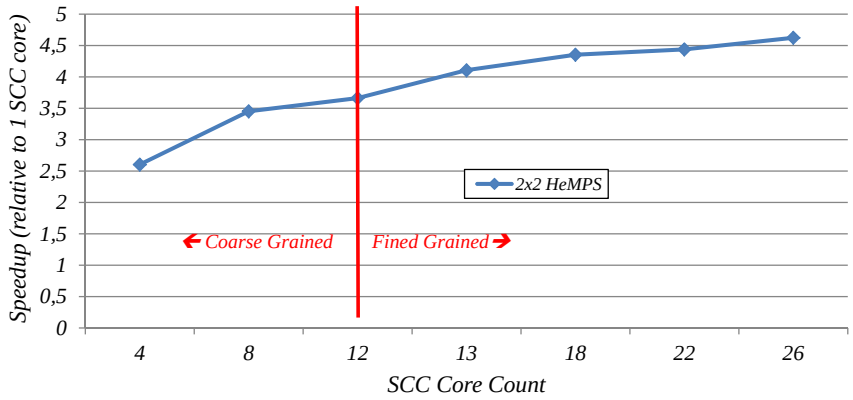


Abbildung 4.36.: Grobgranulare versus feingranulare Partitionierung

Durch die feingranulare Partitionierung ist es möglich ein 2x2 HeMPS Modell über zwölf Partitionen hinaus auf bis zu 26 Partitionen zu verteilen. (alle Mess-

punkte rechts der roten Linie). Dadurch kann die Beschleunigung ausgehend von 3.63x bei 12 Partitionen auf bis zu 4.62x bei 26 Partitionen gesteigert werden.

Darüber hinaus eröffnet die Möglichkeit zur feineren Partitionierung noch weitere Freiheitsgrade zur Optimierung. Im Fall von HeMPS lässt sich beispielsweise der Kommunikationsaufwand zwischen logischen Prozessen in Form der Deltaschranken von existierenden beschränkten logischer Links durch Umverteilen einzelner Submodule reduzieren. Um dies zu demonstrieren, wurden zunächst verschiedene homogene Toplevel Partitionierungen mit einer Granularität von Tiles erzeugt. Dabei wurden nur Signale geschnitten, die zwischen benachbarten Routern verlaufen. Die HeMPS Router verfügen über einen separaten Empfangspuffer für jeden benachbarten Router. Diese Puffer sind in einem eigenen Submodul implementiert. Im Zuge der Optimierung wurden diese Empfangspuffer jeweils in die Partition des sendenden Routers verschoben.

Messergebnisse, welche die optimierte Variante der nicht optimierten Variante für verschiedene Modellgrößen gegenüberstellen, sind in Abb. 4.37 dargestellt. Die Optimierung bewirkt generell eine höhere Beschleunigung. Wie man der Abbildung auch entnehmen kann, konnte die höchste beim adaptiven Verfahren gemessene Beschleunigung von 33.13x beim 6x8 Modell auf 48 SCC Kernen nur mit dieser Art der Optimierung erreicht werden.

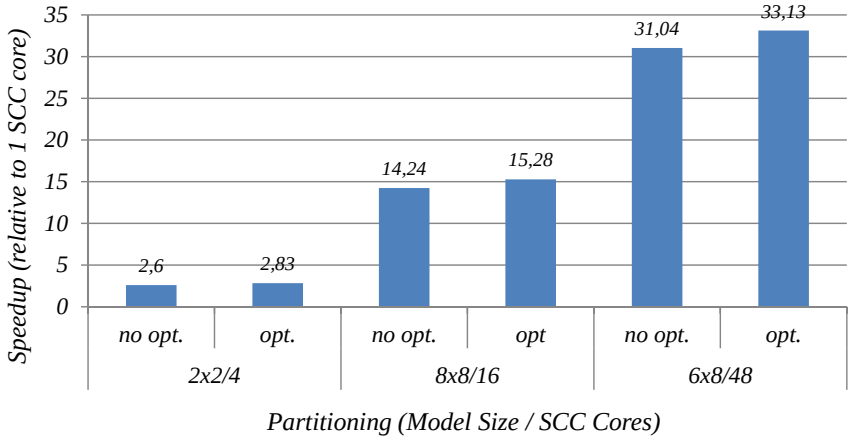


Abbildung 4.37.: Optimierte versus Toplevel Partitionierung

4.5.7.3. Automatische Partitionierung

Zur Demonstration von Funktion und Anwendbarkeit der automatischen Partitionierung wurden abschließend mehrere umfangreiche Messreihen mit ver-

schiedenen HeMPS Modellen (unter Verwendung von reinen RTL und *Cycle-Approximate Level (CAL)* PEs) durchgeführt. Unter Vorgabe der Anzahl zu verwendender SCC Kerne wurden die Modelle mit der Werkzeugkette aus Abschnitt 4.5.6 jeweils automatisch analysiert, partitioniert und anschließend ausgeführt. Für die automatische Partitionierung wurden zwei unterschiedliche Ansätze verwendet:

- **Strategie I**: Knoten- und Kantengewichte werden durch Compileranalyse ermittelt. Für $\vec{n}_i^{eval}$ von Prozess p_i aus Ausdruck 4.19 werden dessen Aktivitätsmuster (vgl. Abschnitt 4.5.2.4) in Form eines binären Vektors verwendet. T_i^{eval} wird auf 1 und $\hat{n}_i^{eval}$ auf 0 gesetzt. Als Toleranzwert für das Lastungleichgewicht wird generell 1.2 angenommen. Für SystemC Signale wird der Betrag des Aktivitätsmustervektors des jeweils schreibenden Prozesses als Kantengewicht benutzt. Aktive Signale werden zusätzlich mit einem Faktor $3x$ gewichtet. Als METIS Partitionierungsmethode wird *Multilevel k-Way Partitioning* gewählt.
- **Strategie II**: Knoten- und Kantengewichte werden durch Profiling ermittelt. Diese entspricht der in Abschnitt 4.5.6.3 beschriebenen Methode. Zudem wird das erste Element der Lastvektoren $\vec{w}_i^{load}$ mit einer Toleranz von 1.15 behandelt, die restlichen Vektorelemente mit einer Toleranz von 1.4. Als METIS Partitionierungsmethode wird *Multilevel Recursive Bisectioning* gewählt.

Für alle Untersuchungen wurde $\theta^{max} = 8$ gesetzt. Die erzielten Beschleunigungen bei Verwendung von Strategie I und adapativer Synchronisation sind in den Abb. 4.38 und 4.39 in Abhängigkeit der verwendeten SCC Kerne und der verschiedenen Modellkonfigurationen dargestellt[14]. Bei den kleineren Modellen ist wegen der geringeren Rechenkomplexität generell ein schnellere Sättigung der Beschleunigung zu verzeichnen. Vergleicht man den Verlauf der Kurven der RTL und CAL basierten Simulationen, so liegen die Gewinne durchweg in ähnlichen Größenordnungen, trotz der geringeren Rechenkomplexität der CAL PEs.

Im Mittel steigt die Beschleunigung mit der Anzahl der SCC Kerne an. Es sind allerdings deutliche Schwankungen und Ausreißer zu erkennen. Beim RTL Modell sind diese noch stärker als beim CAL basierten Modell. Dabei fällt auf, dass meist nur in einigen Sonderfällen, insbesondere beim 4x4, 5x5 und 6x6 RTL Modell, wenn die Anzahl der HeMPS Tiles identisch zur Anzahl der SCC Kerne ist, gute Beschleunigungen erzielt werden. Dies ist mit der $3x$ Gewichtung der aktiven Signale zu erklären. Zwischen den HeMPS Tiles existieren wenige aktive Signale, so dass eine Partitionierung an Tile Grenzen bevorzugt wird.

[14] Bei Verwendung von Strategie I in Kombination mit dem 2x2 Modell und bei Vorgabe vieler SCC Kerne > 25 resultierte die Partitionierung mit METIS teilweise in der Auslassung von SCC Kernen. Dabei blieben maximal fünf SCC Kerne ungenutzt.

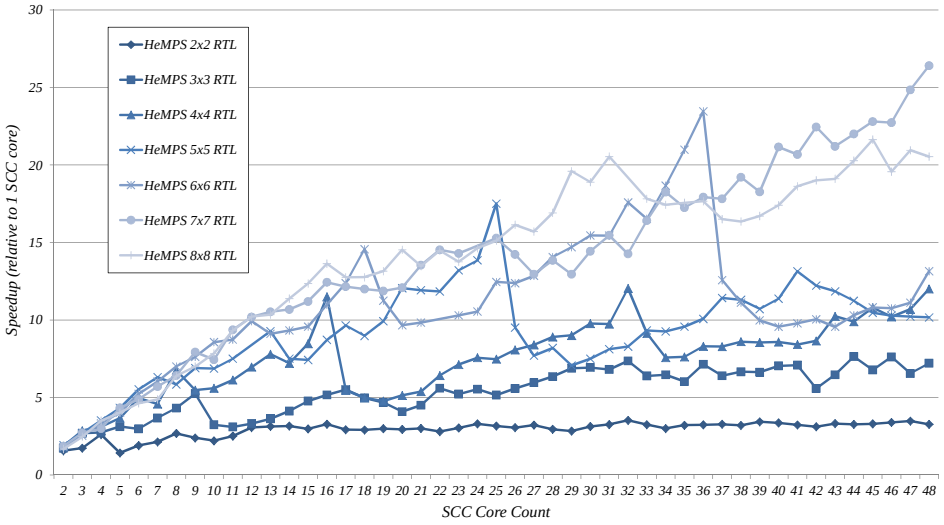


Abbildung 4.38.: Beschleunigung von HeMPS (RTL PEs) mit Strategie I und adaptiver Synchronisation

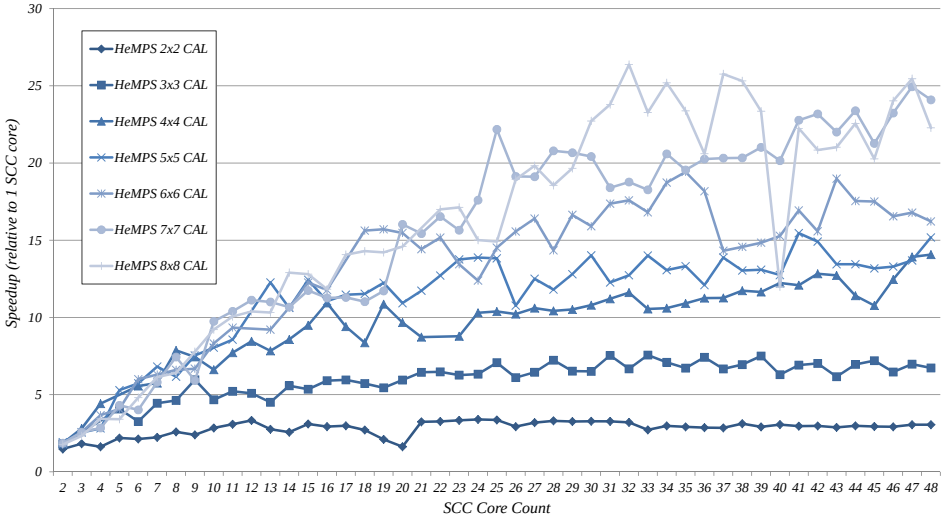


Abbildung 4.39.: Beschleunigung von HeMPS (CAL PEs) mit Strategie I und adaptiver Synchronisation

In Abb. 4.40 sind die Messergebnisse der parallelen RTL Simulation mit Strategie II und adaptiver Synchronisation dargestellt. Vergleicht man Abb. 4.40 mit Abb. 4.38, so kommen die Kurven viel näher an den durch das Amdahlsche Gesetz (vgl. Abschnitt 2.4.4.5) theoretisch beschriebenen Verlauf heran.

Durch den Austausch der Partitionierungsmethode und der per Compileranalyse ermittelten Schätzwerte mit realen Profilingdaten ist es offensichtlich möglich, die Schwankungen in der Beschleunigung bis zu einem gewissen Grad auszugleichen. Auffällig ist außerdem, dass die starken Maxima, die bei Partitionierung des RTL Modells mit Strategie I entstanden sind, mit Strategie II nicht mehr auftreten. Allerdings kann mit Strategie II auch außerhalb der Sonderfälle eine relativ gute Lastverteilung erzielt werden.

Sowohl bei Strategie I als auch bei Strategie II fällt die Beschleunigung bei automatischer Partitionierung im Durchschnitt niedriger als bei manueller Optimierung aus. Beispielsweise konnte das 8x8 HeMPS RTL Modell mit manueller Partitionierung und manueller Optimierung maximal um einen Faktor 30.81x beschleunigt werden. Bei automatischer Partitionierung wurde dagegen nur eine maximale Beschleunigung von 20.54x (Strategie I) bzw. 29.28x (Strategie II) erreicht. Dies ist auf die suboptimale Gewichtung von Signalen und Prozessen während der automatischen Partitionierung zurückzuführen. Offensichtlich ist noch weiteres Optimierungspotential vorhanden.

Um den Vorteil der adaptiven Synchronisation weiter zu verdeutlichen, ist in Abb. 4.41 schließlich die Beschleunigung illustriert, wenn vollsynchrone Ausführung auf Basis globaler Barriers verwendet wird[15]. Insgesamt liegt die maximal erzielte Beschleunigung deutlich unter $20x$, was signifikant kleiner ist als die mit adaptiver Synchronisation maximal erreichten Werte.

Des Weiteren ist ein deutlicher Rückgang der Beschleunigung mit steigendem Parallelisierungsgrad zu erkennen, der bei adaptiver Synchronisation nicht so deutlich auftritt. Die vollsynchrone Ausführung skaliert offensichtlich nicht so gut mit der Anzahl an SCC Kernen wie die adaptive Synchronisation. Dieser Effekt wird durch die Implementierung der Barrier verstärkt, welche einen zentralen Master nutzt.

4.5.7.4. Diskussion

Anhand der verschiedenen Fallstudien konnte gezeigt werden, dass es durch einen erhöhten Grad an Automatisierung möglich ist, asynchrone PDES für die deltazyklengenaue Parallelisierung von SystemC RTL Modellen nutzbar zu ma-

[15] Die Implementierung des vollsynchronen Ansatzes, der hier als Vergleichsfall verwendet wird, entspricht weitgehend dem Verfahren, das innerhalb einer großen globalen Domäne angewendet wird. Der Unterschied besteht darin, dass alle Slaves vor jedem neuen Timed- oder Deltacycle immer mit dem Master synchronisieren und blockieren, während sie auf den Master warten.

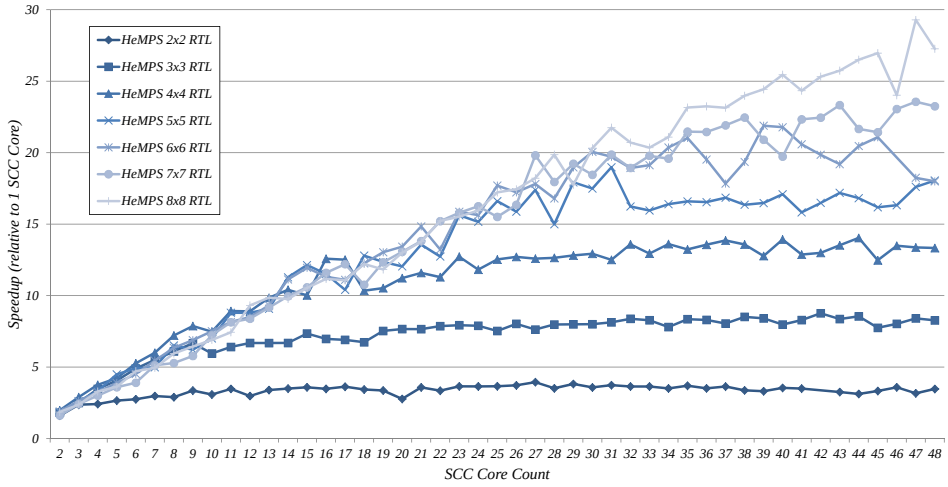


Abbildung 4.40.: Beschleunigung von HeMPS (RTL PEs) mit Strategie II und adaptiver Synchronisation

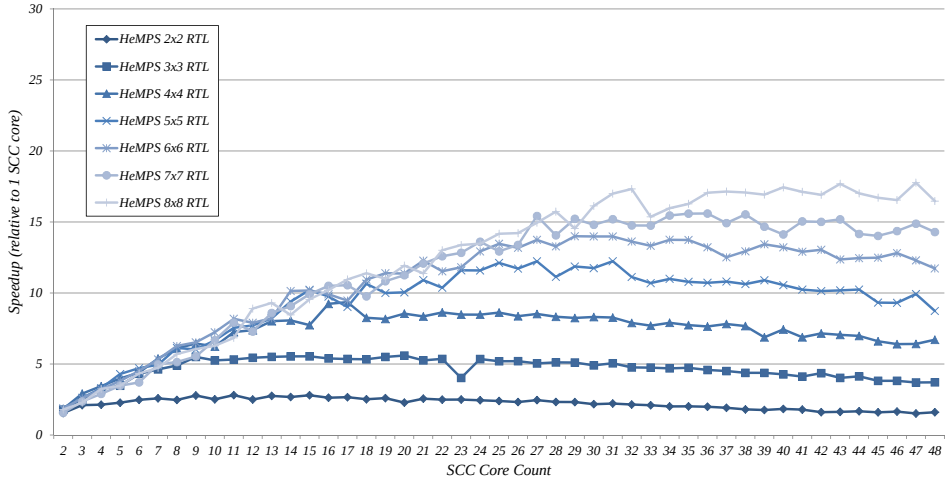


Abbildung 4.41.: Beschleunigung von HeMPS (RTL PEs) mit Strategie II und globalen Barriers

chen. Die erzielten Beschleunigungswerte liegen trotz feinerer Synchronisation in etwa in der gleichen Größenordnung wie beim Verfahren aus Abschnitt 4.4. Durch gezielte Optimierungen konnten sie sogar übertroffen werden.

Die automatische Partitionierung bietet noch Potential für Verbesserungen. Insbesondere für die Kanten des Graphen G_{PS} können u.U. bessere Gewichtungen gefunden werden, um die Qualität der mit METIS erzeugten Partitionierungen weiter zu erhöhen. Insbesondere muss das Gewicht einer Kante besser mit der Klassifikation (aktiv/passiv, delta-beschränkt/unbeschränkt, etc.) der Kante korrelieren, um so den Effekt auf die Performanz detaillierter zu beschreiben, der durch die Klassifikation entsteht. Dazu ist der Einfluss der Klassen von Kanten auf die Leistungsfähigkeit der Simulation genauer zu untersuchen und die Klassen anhand einer skalaren Kostenmetrik zueinander in Beziehung zu setzen.

Die Haupteinschränkung des beschriebenen Ansatzes ist die Limitierung auf RTL und ähnliche Modelle. Um die Anwendbarkeit auch auf andere Modellierungsstile (z.B. TLM) auszudehnen, müssen spezifische Modellierungsartefakte und Modellcharakteristika von den Analyse- und Transformationswerkzeugen sowie der Laufzeitumgebung unterstützt werden. Ein Grund für die Limitierung auf das RTL o.ä. ist die Voraussetzung, dass Partitionen um mindestens einen Deltacycle entkoppelt sind und die durch die Aktion *nextEdgetime*() zurückgegebene Zeit τ^{edge} bei der Berücksichtigung delta-beschränkter Links lokal bekannt ist. Bei RTL Modellen kann dieser Wert unmittelbar lokal vorhergesagt werden. Kommen im Modell aber beliebige Aufrufe von *wait*() oder *next_trigger* mit Zeitargument vor, so sind die jeweiligen Verzögerungszeiten im Allgemeinen erst zur Laufzeit bekannt. Darüber hinaus ist die Verzögerung zunächst nur dem logischen Prozess bekannt, in dem der Aufruf stattfindet.

Um trotz des beschriebenen Sachverhalts nicht auf dezentrale Synchronisation verzichten zu müssen, ist ein möglicher Ansatz, die Zeitpunkte zukünftiger Notifications im Rahmen der Synchronisation zwischen (statisch oder dynamisch zu bestimmenden) ausgewählten logischen Prozessen auszutauschen. Auf Basis der ausgetauschten Informationen könnte dann in jedem logischen Prozess eine allgemeingültigere *nextTime*() Aktion realisiert werden.

Neben der Optimierung von Partitionierung und Synchronisation könnten extrahierte Modellinformationen auch zur Optimierung des lokalen Schedulings genutzt werden. Beispielsweise könnte die Reihenfolge, in der aktive SystemC Prozesse während der Evaluation Phase abgearbeitet werden, mit Hilfe der Analysedaten verbessert werden [16]. SystemC Prozesse, die nicht von Datenabhängigkeiten betroffen sind, könnten bereits vor dem Prüfen der ELOCC ausführen.

[16] Da laut SystemC Standard die Ausführungsreihenfolge in der Evaluations Phase nicht festgelegt ist, ist das Umsortieren der Prozesse grundsätzlich erlaubt. Dies kann aber u.U. mit einem Verlust von Determinismus einhergehen. In diesem Fall wäre z.B. vom Anwender zu entscheiden, ob er dies in Kauf nehmen kann oder nicht.

Umgekehrt könnte man die Ausführung von SystemC Prozesse priorisiert behandeln, wenn diese in der Lage sind, Daten an andere logische Prozesse zu versenden. Auf diese Weise können sich Wartezeiten auf benachbarte logische Prozesse gegebenenfalls verringern.

4.6. Strategie zur Simulation auf Transaktionsebene

Eine Hauptanforderung an die in den Abschnitten 4.3 bis 4.5 vorgestellten Parallelisierungsstrategien war die Unterstützung des RTL Subsets von SystemC und eine zyklenakkurate Ausführung. Es wurde unter anderem gezeigt, dass die Effizienz durch geschickte Ausnutzung von Modelleigenschaften gesteigert werden kann. Allerdings war die erzielbare Simulationsbeschleunigung aufgrund der engen kausalen Kopplung zyklenakkurater Teilmodelle grundsätzlich beschränkt.

Im Folgenden soll daher untersucht werden, inwieweit es möglich ist, die Performanz durch gezielte Abstraktion weiter zu steigern, jedoch den Verlust an zeitlicher Genauigkeit möglichst gering zu halten. Dazu wird zunächst eine neue Methode für die Modellierung und Simulation von NoC-basierten MPSoCs auf Transaktionsebene entwickelt, welche typische mikroskopische Effekte solcher NoC-basierten Architekturen wiedergeben kann. Gleichzeitig wird die kausale Kopplung von Prozessen von Deltacycles auf Timedcycles reduziert[17]. Anschließend wird ein Ansatz zur Integration dieser Methode mit einem parallelen SystemC Kernel vorgestellt. Schließlich wird die Performanz der Gesamtmethodik bewertet. Im Rahmen einer vom Autor betreuten Bachelorarbeit [Buc12] wurden einige Vorarbeiten zum hier beschriebenen Verfahren gemacht. Teile der folgenden Unterabschnitte sind publiziert in [RBR+13a] und [RBR+13b].

4.6.1. Allgemeine Anforderungen

Die zu entwickelnde Modellierungstechnik soll hauptsächlich zur Modellierung von Verbindungsstrukturen und Elementen von Verbindungsstrukturen wie z.B. Routermodulen eines NoCs dienen. Sie sollte aber nicht notwendigerweise auf diesen Bereich beschränkt sein und bei Bedarf auch die Modellierung anderer Elemente einer MPSoC Architektur ermöglichen. Die wiederholte Ausführung ein und desselben TL Modells soll identische Ergebnisse liefern und damit für entsprechende Verifikationsfälle geeignet sein.

[17] Eine Kopplung mit einer Genauigkeit von einem Takt wird im Folgenden, in Abgrenzung zur deltazyklenweisen Synchronisation, als zyklenweise Synchronisation bezeichnet.

Die Implementierung eines Packet-Switched NoC Routers [50] wie dem Hermes Router [203] umfasst üblicherweise die Schichten 1 bis 3 des ISO/OSI Referenzmodells (Bitübertragungs-, Sicherungs- und Vermittlungsschicht) [270]. Die grundlegende Dateneinheit auf der Vermittlungsschicht ist ein Paket, auf der Sicherungsschicht ein Flit und auf der Bitübertragungsschicht ein Signal. Um bei der Modellierung der Kommunikation zwischen Routermodulen mikroskopische Effekte wie Pufferkongestion oder Ressourcenkonflikte akkurat reproduzieren zu können, muss die Modellierung mindestens mit einer Granularität von Flits erfolgen und daher auf der Sicherungsschicht ansetzen. Dabei genügt es, wenn Effekte der Bitübertragungsschicht durch synchrone Ausführungs- und Kommunikationsmechanismen in Erscheinung treten.

4.6.2. Basismethode

Den Kern der Modellierungstechnik bilden sog. leichtgewichtige Module, Scheduler und Prozesse. In Abb. 4.42 ist ein Szenario für eine parallele Simulation bestehend aus zwei logischen Prozessen lp_0 und lp_1 dargestellt, in dem die Modellierungstechnik verwendet wird. Jeder der beiden logischen Prozesse kapselt einen sequentiellen SystemC Kernel. Jeder Kernel führt wiederum jeweils ein Teilmodell, bestehend aus zwei Modulen aus. Das Teilmodell auf Kernel k_0 besteht aus einem leichtgewichtigen Modul μ_0 und einem konventionellen Modul m_0.

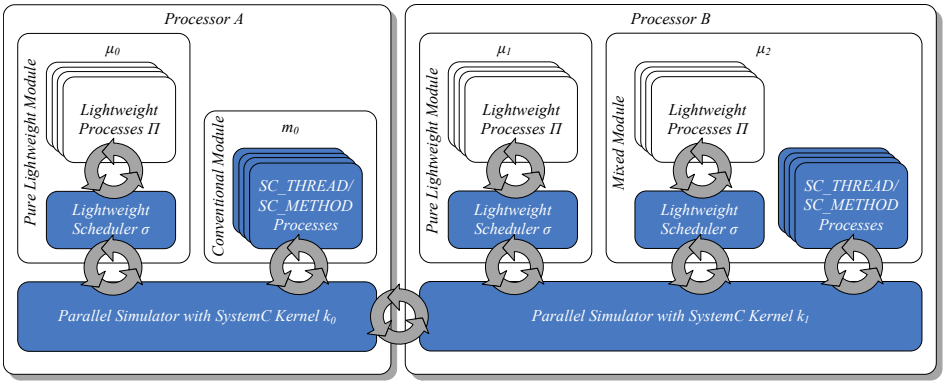


Abbildung 4.42.: Prinzip der TL Modellierungsstrategie

Ein leichtgewichtiges Modul ist ein spezieller Typ eines normalen SystemC Moduls, das auf Basis eines leichtgewichtigen Schedulers und leichtgewichtigen Prozessen ein hierarchisches Scheduling implementiert. Der Aufruf eines leichtgewichtigen Schedulers liegt in der Verantwortung des SystemC Kernels, der

Aufruf eines leichtgewichtigen Prozesses liegt in der Verantwortung des leichtgewichtigen Schedulers. Kommunikation und Synchronisation zwischen leichtgewichtigen Modulen erfolgt anhand von Transaktionen. Dabei werden für die parallele Simulation sowie für die Reproduktion mikroskopischer NoC Effekte geeignete Mechanismen für das Scheduling und die Kommunikation benutzt.

Im Gegensatz zu μ_0 besteht m_0 ausschließlich aus gewöhnlichen SystemC Prozessen. Auf Kernel k_1 befindet sich ein leichtgewichtiges Modul μ_1 und ein gemischt modelliertes Modul μ_2. Eine gemischte Modellierung kann dann notwendig sein, wenn verschiedene Modellierungstechniken miteinander gekoppelt werden müssen. Ein gemischtes Modul dient dann als Adapter, welcher zwischen Syntax und Semantik der beiden Modellierungstechniken übersetzt.

Vor dem Hintergrund des in Abschnitt 4.2.2 eingeführten Schichtenmodells wird durch das Konzept prinzipiell eine weitere Schicht oberhalb der traditionellen SystemC Modellebene eingeführt, wodurch die zusätzliche Hierarchisierung entsteht. Dies hat verschiedene Vorteile: Kommunikation zwischen leichtgewichtigen Prozessen innerhalb eines Moduls kann vollständig lokal über den leichtgewichtigen Scheduler abgewickelt werden. Wie später gezeigt wird, können in Modulen enthaltene leichtgewichtige Prozesse vom SystemC Kernel und von anderen leichtgewichtigen Prozessen zudem (kontrolliert) temporär entkoppelt werden. Eine solche temporäre Entkopplung reduziert den Synchronisationsaufwand sowohl zwischen leichtgewichtigen Schedulern und lokalem SystemC Kernel (vertikal), als auch zwischen verteilten logischen Prozessen (horizontal) und kann so zu einer allgemeinen Performanzsteigerung beitragen.

4.6.2.1. Komponenten

Die zur Modellierung notwendigen Komponenten sind in der folgenden Definition eines leichtgewichtigen Moduls zusammengefasst:

Definition 4.22 *(**Leichtgewichtiges Modul**): Ein leichtgewichtiges Modul $\mu \in M$ ist ein SystemC Modul, das den Regeln der Modellierungsmethodik folgt. μ ist ein Tupel $(\sigma, \tau, P, V, FI, FO, C, S)$, wobei folgendes gilt:*

- *σ ist ein leichtgewichtiger Scheduler,*
- *τ ist eine lokale Variable zur Speicherung der lokalen Zeit eines leichtgewichtigen Moduls,*
- *P ist die Menge aller leichtgewichtigen Prozesse des Moduls,*
- *V ist die Menge der Variablen (Variablencontainer),*
- *FI ist die Menge der Eingangsartefakte aller logischen Puffer,*

- *FO ist die Menge der Ausgangsartefakte aller logischen Puffer,*
- *C ist die Menge der Kontrollpuffer,*
- *S ist die Menge aller eingehenden TLM Socketverbindungen über Target-Sockets S^{target} und ausgehenden TLM Socketverbindungen über Initiator-Sockets S^{init}.*

Abb. 4.43 zeigt ein Beispiel, anhand dessen die Bedeutung der einzelnen Komponenten deutlich gemacht werden soll. In der folgenden Beschreibung wird zunächst auf die Modellierung von Verhalten und Kommunikation innerhalb eines leichtgewichtigen Moduls und anschließend auf die Modellierung von Kommunikation zwischen Modulen eingegangen. Anschließend wird ein Ausführungsmechanismus beschrieben, der beides miteinander kombiniert.

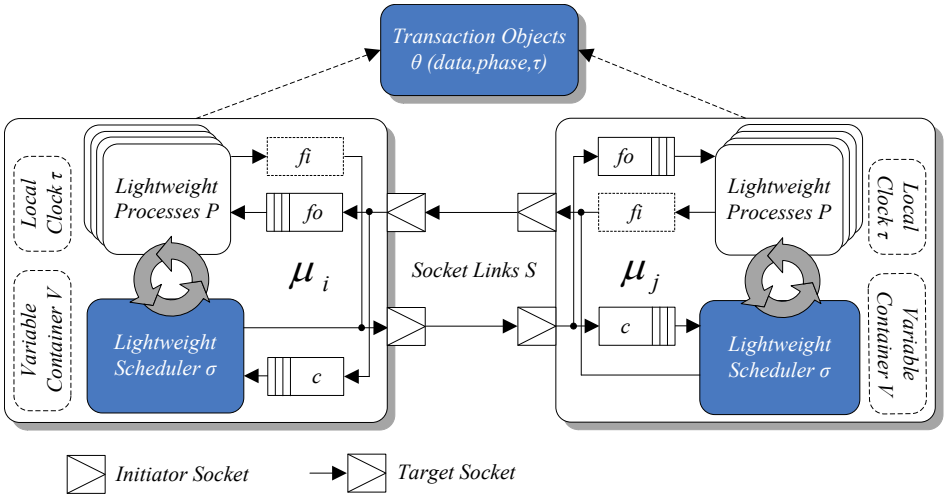


Abbildung 4.43.: Grundlegende Bausteine der Modellierungsstrategie

4.6.2.2. Verhalten und Kommunikation innerhalb leichtgewichtiger Module

Für die Verhaltensmodellierung innerhalb von Modulen stehen die Artefakte σ, P, V und τ zur Verfügung. Ein leichtgewichtiger Scheduler σ ist ein herkömmlicher SystemC Prozess und zugleich der einzige richtige SystemC Prozess innerhalb eines Moduls. Er kann entweder als *SC_METHOD* oder *SC_THREAD* implementiert werden. Ein Scheduler σ ist verantwortlich für die Ausführung von leichtgewichtigen Prozessen P.

Leichtgewichtige Prozesse werden als normale C++ Klassen realisiert, die mit einer speziellen dem Scheduler bekannten Schnittstelle ausgestattet sind. Leicht-

gewichtige Prozesse ersetzen konventionelle *SC_METHOD* oder *SC_THREAD* basierte SystemC Prozesse. Durch sie kann eine prozessbasierte Strukturierung von Verhaltensbeschreibungen innerhalb von Modulen erfolgen. Der Scheduler σ und die Prozesse P implementieren zusammen ein kooperatives Multitasking: Die Prozesse werden regelmäßig vom Scheduler über ihre Schnittstelle getriggert und sind anschließend selbst dafür verantwortlich, den Kontext wieder an den Scheduler zurückzugeben.

Die Variablen V des Variablencontainers dienen zur Implementierung einer effizienten modulinternen Kommunikation zwischen leichtgewichtigen Prozessen. Der Variablencontainer kann beispielsweise als simpler C++ Container realisiert werden. Er dient als Ersatz für SystemC Channels wie z.B. *sc_signal*. Die lokale Zeitvariable τ ist eine separate (ganzzahlige) C++ Variable. Sie dient zur Speicherung der aktuellen lokalen Zeit eines leichtgewichtigen Moduls.

Die Verteilung der Verhaltensbeschreibung auf mehrere leichtgewichtige Prozesse ist die Grundlage für die Modellierung von modulinterner Parallelität. Die Verwendung von leichtgewichtigen Schedulern gestattet prinzipiell eine Verhaltensmodellierung auf Basis unterschiedlicher Berechnungsmodelle. Für die Modellierung parallel ausführender und kommunizierender Zustandsautomaten ist eine synchrone dynamische Semantik mit zyklenweiser Synchronisation gut geeignet.

Zur Realisierung zyklenweiser Synchronisation besitzt jeder leichtgewichtige Prozess $p \in P$ drei öffentliche Schnittstellenmethoden mit dem Namen *pre()*, *exec()* und *post()*. Diese dienen als Callback-Methoden und werden vom leichtgewichtigen Scheduler innerhalb einer leichtgewichtigen Iteration sukzessive jeweils genau einmal aufgerufen, d.h. zunächst alle *pre()* Methoden, dann alle *exec()* Methoden und zuletzt alle *post()* Methoden. Ein Simulationslauf umfasst dann viele solcher Iterationen nacheinander. Nach jeder Iteration wird die lokale Zeit τ um ein $\Delta\tau$ erhöht, welches dem Taktzykluszeit entspricht. Die Callback-Methoden eines Prozesses p haben folgende Aufgaben:

- *pre()*: Durch diese Methode wird sichergestellt, dass p auf einem rein lokalen Zustandsspeicher arbeitet. Relevante Variablen aus V werden dazu als Parameter (Kopie) übergeben. Die Werte der übergebenen Parameter werden entweder direkt in internen Eingangsvariablen von p gespeichert oder in eine geeignete Darstellung transformiert (Vorverarbeitung anhand von Teilen einer modellierten kombinatorischen Eingangslogik) und dann erst intern gespeichert.
- *exec()*: Diese Methode implementiert eine Verhaltensbeschreibung in Form eines endlichen Zustandsautomaten. Jeder Aufruf der *exec()* Methode bewirkt die Berechnung eines Schritts (Taktes) des Zustandsautomaten auf Basis der in *pre()* gelesenen Eingangsvariablen und evtl. zusätzlicher in-

terner Variablen. Neu berechnete Ausgangswerte werden in internen Ausgangsvariablen von Prozess p abgelegt.

- *post()*: Mit dieser Methode werden die Werte der internen Ausgangsvariablen von p in entsprechende Variablen in V kopiert. Ausgangsvariablen werden dabei als Parameter übergeben. Als Alternative kann diese Methode, äquivalent zur *pre()* Methode, auch zur Modellierung kombinatorischer Ausgangslogik genutzt werden, wodurch interne Ausgangsvariablen in eine geeignete Darstellung transformiert werden.

Stellt man sicher, dass jede Variable $v \in V$ nur von genau einem leichtgewichtigen Prozess geschrieben werden kann, so ist das Ergebnis einer Iteration unabhängig von der Ausführungsreihenfolge der Prozesse P während der Iteration. Grundsätzlich könnte für synchrone Ausführung auch die *pre()* mit der *exec()* Methode zu einer Methode kombiniert werden. In Verlauf der Implementierung hat sich die Verschiebung von Teilen der kombinatorischen Eingangslogik in eine separate *pre()* Methode allerdings als vorteilhaft für die Strukturierung herausgestellt.

4.6.2.3. Kommunikation zwischen leichtgewichtigen Modulen

TLM Socketverbindungen und Transaktionen

In einem TL Modell kommunizieren ein Initiator und ein Target-Modul im Allgemeinen über eine Socketverbindung, wobei der Initiator Zugriff auf ein Initiator Socket und das Target Zugriff auf ein Target Socket hat. Der Initiator übermittelt durch *nb_transport_fw()* Aufrufe Transaktionen auf dem Vorwärtspfad, das Target antwortet durch Aufrufe von *nb_transport_bw()* auf dem Rückwärtspfad. Dadurch wird der Initiator über den neuen Zustand im Target informiert (vgl. Abschnitt 2.3.2.3).

In einem NoC ist jeder Router Initiator und Target zugleich. Diese Unterscheidung ist damit (zumindest für diesen Anwendungsfall) hinfällig. Daher wurde in dieser Arbeit eine Alternative gewählt: Zwischen Routermodulen μ_i und μ_j werden grundsätzlich zwei Socketverbindungen auf Basis von Initiator und Target Sockets von entgegengesetzter Orientierung instanziiert. Auf jeder Socketverbindung werden ausschließlich *nb_transport_fw()* Methodenaufrufe verwendet. Ein solcher Aufruf kann dann sowohl Transaktionen mit vorwärtsgerichteter als auch rückwärtsgerichteter Information übertragen. in Anlehnung an [27] ist eine Transaktion wie folgt aufgebaut:

Definition 4.23 *(**Transaktion**): Eine Transaktion θ dient zum Informationsaustausch auf einer bestimmten Protokollschicht zwischen leichtgewichtigen Modulen. θ ist ein Tupel $(data, phase, \tau)$ mit:*

- *data: Bezeichnet eine Datenstruktur zur Übertragung von Nutzdaten der modellierten Protokollschicht.*
- *phase: Bezeichnet eine Variable, die die Protokollphase spezifiziert.*
- τ*: Bezeichnet eine Variable, die den Zeitpunkt spezifiziert, an dem* θ *im Empfänger verarbeitet werden muss.*

Im Allgemeinen können Transaktionen zur Simulationslaufzeit dynamisch erzeugt und wieder gelöscht werden. Dabei bezeichnet Θ *die Menge der zu einer bestimmten Simulationszeit in einem logischen Prozess erzeugten und existierenden Transaktionen* θ*. Dynamisch erzeugte Transaktionen werden auch als Transaktionsobjekte bezeichnet.*

Zur Modellierung auf der Sicherungsschicht wird zwischen zwei Phasen unterschieden, einer *Kontrollphase* und einer *Datenphase*: $\theta.phase \in (CTRL, DATA)$. Falls $\theta.phase \equiv CTRL$, so heißt θ *Kontrolltransaktion*, falls $\theta.phase \equiv DATA$, so heißt θ *Datentransaktion*. In der Kontrollphase werden mit $\theta.data$ Kontrollinformationen wie z.B. Pufferfüllstände oder Zählerstände zur Datenflusskontrolle übertragen, in der Datenphase Flits.

Der Zeitstempel $\theta.\tau$ dient als Hilfsmittel für die zeitliche Synchronisation miteinander kommunizierender leichtgewichtiger Module. Der Zeitstempel ist Teil der Modellierungstechnik und nicht Teil der modellierten Funktion. Seine Verwendung ist dennoch optional. Sie hängt im Allgemeinen vom Synchronisationsverfahren ab, das in einem leichtgewichtigen Scheduler implementiert ist (vgl. Abschnitt 4.6.3).

Pufferung von Flits auf der Sicherungsschicht

Die Artefakte aus den Mengen FI und FO dienen zur Modellierung eines verteilten Pufferungsmechanismus für Datentransaktionen bzw. Flits:

Definition 4.24 *(**Logischer Puffer**): Ein logischer Puffer besteht aus einem Puffereingangsartefakt* fi *und einem Pufferausgangsartefakt* fo*. Das* fi *Artefakt befindet sich im Sendermodul und ist die Schnittstelle zum Schreiben von Datentransaktionen. Das* fo *Artefakt befindet sich im Empfängermodul und ist die Schnittstelle zum Lesen. Eingangs- und Ausgangsartefakt verwalten gemeinsam den Zustand des logischen Puffers. Nur das Pufferausgangsartefakt* fo *speichert tatsächlich Datentransaktionen in einer internen FIFO Struktur.*

Zur gemeinsamen verteilten Verwaltung des Pufferzustands verfügen beide Artefakte über jeweils drei Zustandsvariablen. In einem fi Artefakt sind dies f^s,

f^{fi} und $\overline{f^{fo}}$. In einem fo Artefakt sind dies f^s, f^{fo} und $\overline{f^{fi}}$. Die Variablen haben folgende Bedeutung:

- f^s ist der synchronisierte Füllstandwert.
- f^{fi} ist der im fi Artefakt sichtbare Füllstandswert.
- f^{fo} ist der im fo Artefakt sichtbare Füllstandswert.
- $\overline{f^{fi}}$ ist der letzte bekannte Wert von f^{fi} auf Empfängerseite.
- $\overline{f^{fo}}$ ist der letzte bekannte Wert von f^{fo} auf Senderseite.

Außer f^s können sich die Variablen im Laufe der Simulationsausführung sowohl innerhalb eines Artefakts als auch zwischen den Artefakten unterscheiden. Nur an bestimmten Synchronisationspunkten zwischen Modulen ist die Identität aller Variablen gegeben:

Definition 4.25 *(**Synchronisationspunkt eines logischen Puffers**): Gegeben seien die Module μ_i und μ_j. μ_i enthält ein fi Artefakt und μ_j ein fo Artefakt. Ein Simulation bestehend aus μ_i und μ_j hat den Synchronisationspunkt τ^{sync} überschritten, wenn $\tau^{sync} = \tau_i = \tau_j$ und alle sechs Zustandsvariablen in den fi und fo Artefakten nach Durchführung der folgenden Berechnungen den gleichen Wert haben:*

Auf der Senderseite (in fi von μ_i):

$$f^{fi} \leftarrow f^{fi} + \overline{f^{fo}} - f^s \tag{4.22}$$

$$f^s \leftarrow f^{fi} \tag{4.23}$$

Auf der Empfängerseite (in fo von μ_j):

$$f^{fo} \leftarrow f^{fo} + \overline{f^{fi}} - f^s \tag{4.24}$$

$$f^s \leftarrow f^{fo} \tag{4.25}$$

Vor der Überschreitung eines Synchronisationspunktes muss der Wert von $\overline{f^{fi}}$ vom Sender zum Empfänger übermittelt worden sein. Dies kann z.B. dadurch geschehen, dass der Empfänger alle seit dem letzten Synchronisationspunkt eingegangenen Datentransaktionen zählt. Eine Alternative ist die Übermittlung von f^{fi} in einer Kontrolltransaktion. Genauso muss der gestrichene Wert $\overline{f^{fo}}$ vom Empfänger zum Sender per Kontrolltransaktion übermittelt worden sein. Ein undirektionaler Datenfluss hat so einen bidirektionalen Kontrollfluss zur Folge.

Im Fall eines Eingangsartefakts fi werden an einem Synchronisationspunkt alle Datentransaktionen sichtbar, die seit dem letzten Synchronisationspunkt vom logischen Puffer gelesen wurden. Im Fall eines Ausgangsartefakts fo werden an einem Synchronisationspunkt alle Datentransaktionen sichtbar, die seit dem letzten Synchronisationspunkt auf den logischen Puffer geschrieben wurden. Dieses Verhalten ist vergleichbar mit dem E/U Paradigma und fundamental für die Realisierung von zyklenweise synchronen Modellen sowie deterministischem Zeitverhalten im parallelisierten Fall.

Synchronisierte Kommunikation

Mit Hilfe von Kontrolltransaktionen ist es möglich, einen in zeitlicher Genauigkeit und Performanz skalierbaren Mechanismus für die Kommunikation über logische Puffer zu implementieren. Beim hier zunächst beschriebenen Basisverfahren kann dazu die Kommunikationslatenz zwischen leichtgewichtigen Modulen vor Beginn der Simulation statisch festgelegt werden. Legt man die Kommunikationslatenz gleichzeitig global fest, so entspricht dies einem globalen Quantum, ähnlich dem globalen Quantum aus dem SystemC/TLM Standard (vgl. [27]):

Definition 4.26 *(**Globales Quantum**): Das **globale Quantum** q ist ein maximales, global festgelegtes Zeitintervall, das den zeitlichen Abstand von Synchronisationpunkten zwischen leichtgewichtigen Modulen spezifiziert.*

Durch das globale Quantum ist die Anzahl N der Synchronisationspunkte, die auf jedem verteilten Puffer während eines Simulationslaufs auftreten, gleich groß und lässt sich mit der maximalen Simulationszeit τ^{max} über $N = \tau^{max}/q$ berechnen. Um eine zyklenweise Synchronisation zu erreichen, muss das globale Quantum einem Taktzyklus entsprechen, d.h. $q = \tau^{cycle}$. Durch größere Werte von q kann eine zeitliche Entkopplung bei gleichzeitigem Erhalt des Determinismus erzielt werden.

Angenommen, ein leichtgewichtiges Modul μ hat gerade mit allen benachbarten Modulen den n-ten Synchronisationsvorgang erfolgreich abgeschlossen und befindet sich bei der lokalen Zeit $\tau(n)$. Nach Abarbeitung des globalen Quantums q und Erreichen des Zeitpunkts $\tau(n+1) = \tau(n) + q$ beinhaltet die Durchführung des nächsten Synchronisationsvorgangs drei Schritte:

1. μ sendet jeweils genau eine Kontrolltransaktion an alle benachbarten Module. Die Kontrolltransaktionen signalisieren, dass das letzte globale Quantum von μ durchschritten wurde, μ alle Datentransaktionen für das letzte Quantum verschickt hat und die empfangenden Module aus Sicht von μ autorisiert sind, mit der Abarbeitung des nächsten globalen Quantums zu beginnen.

2. Anschließend wartet μ selbst so lange, bis sich von jedem benachbarten Modul mindestens eine Kontrolltransaktion in jedem Kontrollpuffer $c \in C$ befindet. Dies autorisiert μ zur Aktualisierung der Füllstände aller logischen Puffer.
3. Auf Basis der gestrichenen Füllstandswerte, die mit den Kontrolltransaktionen empfangen wurden, kann μ den Zustand der eingehenden und ausgehenden logischen Puffer durch Anwendung der Ausdrücke 4.22 und 4.24 aktualisieren. Dann kann μ das nächste globale Quantums q abarbeiten.

Kontrolltransaktionen erfüllen damit drei Aufgaben gleichzeitig: Übermittlung von Kontrollinformation, Ausschluss transienter Transaktionen und Zeitsynchronisation. Da wegen des konstanten globalen Quantums ein Zeitfortschritt unmittelbar durch den Empfang einer Kontrolltransaktion signalisiert wird, ist im Basisverfahren der Zeitstempel der Transaktion prinzipiell überflüssig.

Globales Quantum versus statischer Lookahead

Der zeitliche Fehler gegenüber zyklenweiser Synchronisation zwischen Modulen ist durch die Größe des globalen Quantums beschränkt. Wird für das globale Quantum die minimale zwischen leichtgewichtigen Modulen auftretende Kommunikationslatenz gewählt, so wird der Genauigkeitsverlust im Vergleich zum RTL Modell auf ein Minimum reduziert. Das globale Quantum kommt dann einem statischen Lookahead gleich. Wenn das globale Quantum größer als der statische Lookahead gewählt wird, so hat das Verfahren die Eigenschaft, die tatsächliche Latenz einzelner Datentransaktionen (Flits) um ein oder mehrere Quanta zu verschmieren.

Betrachtet man das originale signalbasierte RTL Modell des Hermes NoC, so ist die Kontrollinformation immer um einen halben Takt verzögert zu den Daten verfügbar. Dadurch wird eine verlustlose taktweise Übertragung der Daten realisiert. Die halbtaktweise Verschiebung hatte zur Folge, dass in Abschnitt 4.4.8.3 ein statischer Lookahead für das Hermes NoC von $\Delta l^{\tau} < \frac{\tau^{cycle}}{2}$ hergeleitet werden konnte. Bei transaktionsbasierter Modellierung des Hermes NoC kann der statische Lookahead für eine zyklenweise Datenübertragung aus zwei Gründen auf τ^{cycle} erhöht werden: Zum einen führen die Daten- und die Kontrollphase prinzipbedingt immer abwechselnd aus, weshalb eine halbtaktweise Synchronisation bereits explizit modelliert ist. Zum anderen wartet ein leichtgewichtiger Scheduler immer den Empfang aller Kontrolltransaktion für die aktuelle leichtgewichtige Iteration ab, bevor er die Datentransaktionen verarbeitet. Der Lookahead muss daher nicht mehr kleiner als τ^{cycle} gewählt werden, um garantieren zu können, dass für τ^{cycle} alle Datentransaktionen eingetroffen sind.

4.6.2.4. Kombinierter Ausführungsmechanismus beim Basisverfahren

Zur Beschreibung der kombinierten Ausführungsmechanimus in einem leichtgewichtigen Modul μ_i werden folgende Zeitvariablen verwendet:

- τ_i ist die aktuelle lokale Simulationszeit eines leichtgewichtigen Moduls.
- τ^{cycle} ist die Taktzykluszeit.
- τ^{max} ist die maximale Simulationszeit.
- τ^{sync} ist der nächste Synchronisationszeitpunkt.

Innerhalb eines leichtgewichtigen Schedulers sind folgende Aktionen definiert:

- *sendCT()*: Sende eine Kontrolltransaktion an jedes benachbarte Modul. Eine Kontrolltransaktion signalisiert, dass ein global und statisch spezifiziertes Zeitquantum q erreicht wurde. Daneben überträgt eine Kontrolltransaktion von μ_i nach μ_j die aktuellen lokalen Füllstandswerte jedes logischen Puffers zwischen μ_i und μ_j.
- *checkCT()*: Gib die Anzahl der Kontrollpuffer aus C zurück, die im aktuellen Zyklus mindestens eine Kontrolltransaktion speichern.
- *sleep($\Delta\tau^s$)*: Gib die Kontrolle an den SystemC Kernel zurück. Dies ist notwendig, um den Ausführungskontext zu wechseln oder um den SystemC Kernel um die spezifizierte Zeit $\Delta\tau^s$ voranschreiten zu lassen. Im Fall einer Simulation auf gemischten Abstraktionsebenen (TLM und RTL) werden dadurch die RTL Anteile ausgeführt.
- *update()*: $\forall fi \in FI \wedge \forall fo \in FO$: Aktualisiere den Zustand der lokal vorhandenen fi bzw. fo Instanz entsprechend der Ausdrücke 4.22 und 4.24. Die gestrichenen Werte wurden mit der letzten Kontrolltransaktion empfangen.
- *popCT()*: $\forall c \in C$: Lösche die oberste Kontrolltransaktion.
- *pre(P)*: $\forall p \in P$: Führe die *pre()* Methode des leichtgewichtigen Prozesses p aus. Übergebe zusätzlich einen Zeiger auf alle $fo \in FO$ bzw. $fi \in FI$ Artefakte, auf die p lesenden bzw. schreibenden Zugriff hat.
- *exec(P)*: $\forall p \in P$: Führe die *exec()* Methode des leichtgewichtigen Prozesses p aus. Führe bei Bedarf Lese- bzw. Schreibzugriffe von p auf fo bzw. fi Artefakte unter Verwendung von Datentransaktionen durch.
- *post(P)*: $\forall p \in P$: Führe die *post()* Methode des leichtgewichtigen Prozesses p aus.
- *incLocal()*: Inkrementiere τ_i um τ^{cycle}.

In Abbildung 4.44 ist die Zustandsmaschine des leichtgewichtigen Schedulers illustriert. Sie besteht aus zwei Teilen:

In den beiden Toplevel-Zuständen s^{sync} und s^{update} ist die Synchronisation mit benachbarten Modulen implementiert. Synchronisation erfolgt mit einer Schrittweite q, dem globalen Quantum. Zwischen den Zuständen s^{sync} und s^{update} kann der Scheduler optional durch Aufruf von *sleep()* dem SystemC Kernel-Scheduler den Kontext übergeben, um diesen ebenfalls um q voranschreiten zu lassen.

der hierarchische $s^{schedule}$ Zustand ist verantwortlich für das lokale Scheduling der leichtgewichtigen Prozesse. Während der Ausführung eines Quantums q vollzieht der Scheduler der $\frac{q}{\tau^{cycle}}$ Iterationen durch die drei Zustände $s^{preprocess}$, $s^{execute}$ und $s^{postprocess}$. Während einer Iteration werden die drei Callback Methoden *pre()*, *exec()* und *post()* eines jeden leichtgewichtigen Prozesses genau einmal aufgerufen.

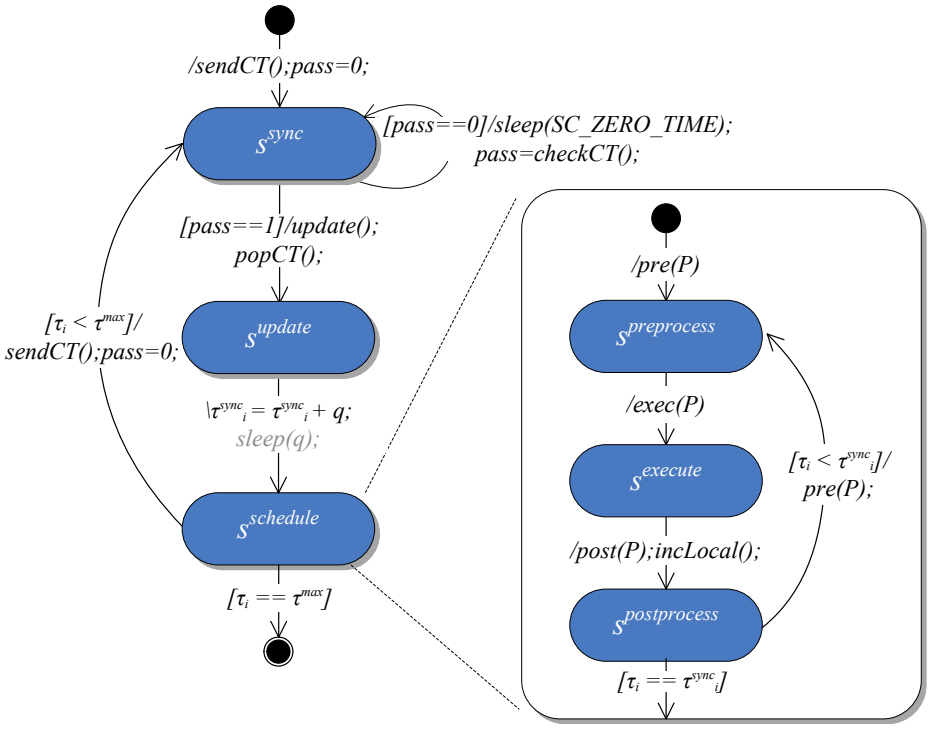


Abbildung 4.44.: Zustandsmaschine Basisvariante

4.6.3. Dynamische Latenzprädiktion

Das im vorigen Abschnitt beschriebene Basisverfahren ermöglicht eine temporäre Entkopplung auf Kosten der zeitlichen Genauigkeit. Ein Kommunikationsvorgang wird mit größer werdendem globalen Quantum über ein oder evtl. auch mehrere Quanta verschmiert. In diesem Kapitel wird daher eine optionale Variante vorgestellt, die den Grad der temporären Entkopplung zwischen leichtgewichtigen Modulen zur Laufzeit dynamisch anpassen kann. Diese *dynamische Latenzprädiktion (DLP)* verhindert den Verlust von zeitlicher Präzision im Vergleich zu nicht entkoppelter zyklenweiser Ausführung mit dem Basisverfahren.

Anstatt ein immer gleich großes Zeitquantum global und statisch zu definieren, existieren bei DLP für jede Modulverbindung anpassbare lokale Zeitquanta. Auf Basis von Latenzannotationen innerhalb von Modulen sowie aktueller Laufzeitparameter wie Zuständen von Automaten oder Zählern, können Kommunikationslatenzen zwischen Modulen dynamisch vorhergesagt werden. Diese Vorhersagen dienen als Garantien für temporäre kausale Unabhängigkeit. Eine temporäre Entkopplung ohne Genauigkeitsverlust wird erreicht, wenn die lokalen Zeitquanta entsprechend der bereitgestellten Prädiktionen angepasst werden. Durch die Integration des Prädiktionsmechanismus in die Modellspezifikation ist ein einfacher Zugriff auf Modellparameter möglich.

4.6.3.1. Modellierung der Latenz

Bei DLP wird der Lookahead nicht mehr nur statisch bestimmt, sondern dynamisch. Dies ist möglich, da Module nicht mehr nur als Blackboxes betrachtet werden. Vielmehr werden funktionsabhängige Latenzen, die innerhalb von Modulen auftreten, für eine dynamische Berechnung des Lookaheads genutzt. Im Kontext eines NoC Routers werden dadurch insbesondere Latenzen unterschiedlicher Protokollschichten für die kontrollierte Adaption der temporären Entkopplung nutzbar gemacht. Zur Beschreibung von DLP eignet sich die Darstellung anhand eines *dynamischen Latenzgraphen*:

Definition 4.27 *(**Dynamischer Latenzgraph**: Ein **dynamischer Latenzgraph** ist ein gerichteter Graph $G_L(FI, FO, E, D(\tau))$ mit Knoten $fi \in FI$ und $fo \in FO$, Kanten $e \in E$ und Kantengewichten $d(\tau) \in D(\tau)$. Er beschreibt den Zustand eines Netzwerks leichtgewichtiger Module zu diskreten äquidistanten Zeitpunkten $\tau(n)$ mit $n \in \mathbb{N}$. Jeder Knoten repräsentiert genau ein fi oder ein fo Artefakt eines leichtgewichtigen Moduls. Zwei Knoten fi_i und fo_j sind über eine gerichtete Kante e_{ij} bzw. e_{ji} verbunden, wenn eine kausale Abhängigkeit von fi_i nach fo_j bzw. von fo_j nach fi_i existiert. Das*

Gewicht d_{ij} einer Kante e_{ij} spezifiziert die Latenz ab dem Zeitpunkt $\tau(n)$, nach deren Ablauf der Knoten mit Index j von Knoten mit Index i kausal beeinflusst wird.

Mit Hilfe des dynamischen Latenzgraphen kann das zeitliche Verhalten von Modulen im Sinne von auftretenden zeitlichen Verzögerungen zwischen Pufferartefakten des gleichen Moduls oder benachbarter Module modelliert werden. Durch die Limitierung des Latenzgraphen auf Knoten, die Pufferartefakte repräsentieren, richtet sich der Fokus automatisch auf das für die Modellierung von Routern in einem NoC wichtige reaktive Eingangs- und Ausgangsverhalten. Die Modellierung anhand zweier Typen von Knoten erlaubt die getrennte Betrachtung von statischen und dynamisch erzeugten Anteilen des Lookahead.

4.6.3.2. Latenzgraph für das Hermes NoC

Um die Beschreibung des Prädiktionsmechanismus zu erleichtern, wird im Folgenden das bereits bekannte Hermes NoC [203] betrachtet. Dessen TL Modell ist in Abbildung 4.45 links illustriert (Kontrollpuffer wurden im Bild aus Gründen der Übersichtlichkeit ausgelassen). Das Routermodell besteht aus typischen Elementen wie

- **Puffer**: Diese dienen zur Zwischenspeicherung von Flits. Puffer existieren für jede der vier Himmelsrichtungen Norden, Osten, Süden, Westen und für die lokale Schnittstelle. Sie werden durch logische Puffer modelliert.
- **Crossbar**: Die Crossbar erlaubt es, Eingänge auf beliebige Ausgänge durchzuschalten. Dies geschieht entsprechend einem XY Routing. Die Crossbar ist mit Hilfe von simplen C++ Variablenfeldern modelliert.
- **Input Controller**: Diese sind verantwortlich für die Flusskontrolle beim Transport von Flits. Die Input Controller werden durch leichtgewichtige Prozesse modelliert.
- **Switch Controller**: Der Switch Controller steuert den Datenfluss durch den Router anhand eines Round Robin Schedulings. Der Switch Controller wird ebenfalls durch einen leichtgewichtigen Prozess modelliert.

Puffer und Crossbar repräsentieren die Bitübertragungsschicht, Input Controller repräsentieren die Sicherungsschicht und der Switch Controller die Vermittlungsschicht des ISO/OSI Referenzmodells [270].

Abb. 4.45 rechts zeigt den Latenzgraphen innerhalb eines einzelnen Hermes Routers. Die sog. Vorwärtskanten von fo zu fi Artefakten sind schwarz gezeichnet, sog. Rückwärtskanten von fi zu fo Artefakten rot. Mit Vorwärtskanten wird die zeitliche Vorwärtswirkung modelliert. Die Richtung der Vorwärtswirkung entspricht dem Fluss von Flits durch das NoC. Mit den Rückwärtskanten wird die

zeitliche Rückwärtswirkung modelliert, welche dem Datenfluss bzw. dem Fluss von Flits entgegengerichtet ist. Die Rückwärtswirkung wird durch Pufferkongestion oder Zugriffskonflikte ausgelöst und hat großen Einfluss auf die Performanz der vorwärtsgerichteten Datenübertragung.

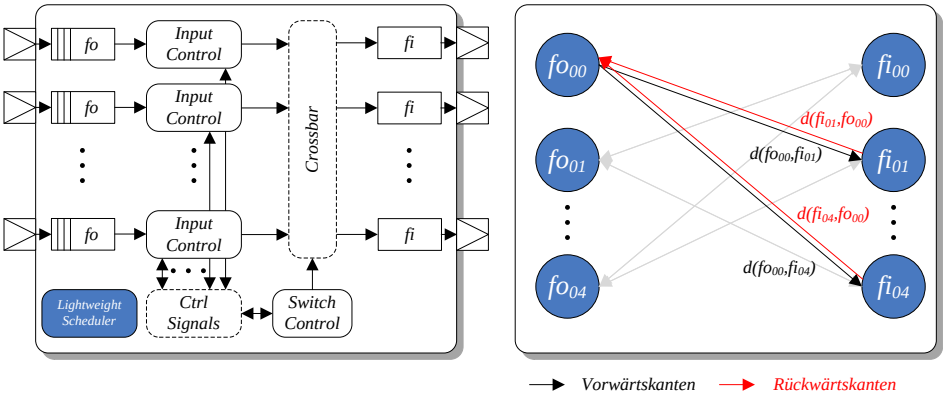


Abbildung 4.45.: TL Modell des Hermes Routers (links) und dessen Latenzgraph G_L (rechts)

Eine Verknüpfung zwischen Routermodulen über logische Puffer beliebiger Anzahl und Orientierung wird im Folgenden als Verbindung con_{ij} oder con_{ji} bezeichnet, wobei $con_{ij} \equiv con_{ji}$. Eine Verbindung con_{ij} repräsentiert damit alle zwischen μ_i und μ_j verlaufenden Vorwärts- und Rückwärtskanten des Graphen G_L. Beim Hermes Router gilt für jede Verbindung zu einem benachbarten Router $|con| = 4$.

Pfade von Vorwärtskanten (Vorwärtspfade) und Pfade von Rückwärtskanten (Rückwärtspfade) durch ein Netzwerk von Routermodulen können separat anhand der Teilgraphen $G_L^v(P, E_L^v)$ und $G_L^r(P, E_L^r)$ beschrieben werden. Dabei ist $E_L \equiv E_L^v \cup E_L^r$ mit $E_L^v \cap E_L^r \equiv \emptyset$. Abb. 4.46 illustriert beispielhaft die beiden Graphen G_L^v und G_L^r für einen Auschnitt aus einem kompletten NoC Modell.

An den Graphen G_L^v und G_L^r ist die Vorwärts- bzw. Rückwärtswirkung durch das komplette NoC zu erkennen. Die Vorwärtswirkung verläuft zwischen Modulen von fi in Richtung fo Artefakten und innerhalb von Modulen von fo in Richtung fi Artefakten. Die Rückwärtswirkung durch das NoC verläuft reziprok dazu, d.h. entgegen der vorwärtsgerichteten Übertragung von Flits: Sie verläuft zwischen Modulen von fo in Richtung fi Artefakten und innerhalb von Modulen von fi in Richtung fo Artefakten.

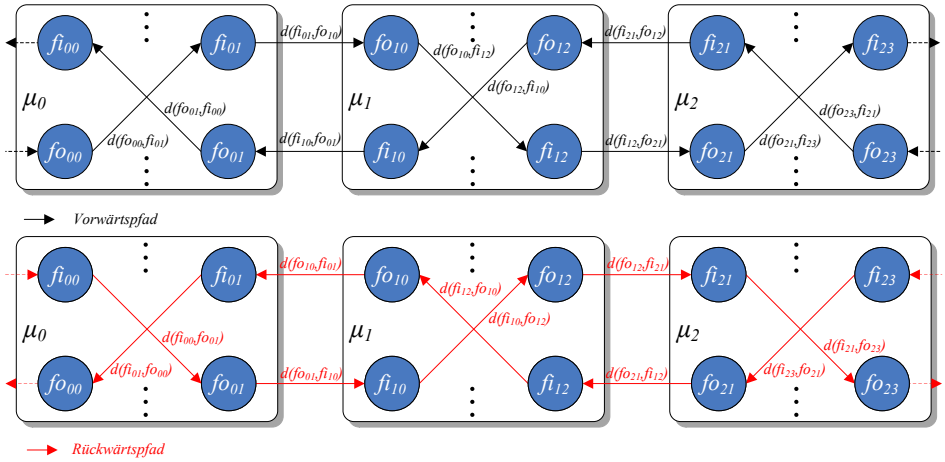


Abbildung 4.46.: Graph G_L^v für die Vorwärtspfade (oben) und Graph G_L^r für die Rückwärtspfade (unten)

4.6.3.3. Statische Latenzmodellierung

Zwischen Knoten in G_L, die unterschiedlichen Routermodulen μ_i und μ_j zugeordnet sind, kann eine *externe Pfadlatenz* für Verbindung con_{ij} definiert werden. Diese entspricht dem im Basisverfahren bereits verwendeten globalen Quantum q und ist daher für jede Verbindung $con \in CON$ identisch und *statisch* festgelegt.

Wird für das globale Quantum die zwischen Modulen minimal auftretende Kommunikationslatenz auf dem Vorwärts- und dem Rückwärtspfad des Graphen G_L gewählt, so entspricht das globale Quantum q einem statischen Lookahead (vgl. Abschnitt 4.6.2.3) bzw. der Latenz der Bitübertragungsschicht. Der statische Lookahead wird im Folgenden auch als *minimale externe Pfadlatenz* d_{ij}^{ext} bezeichnet:

$$d_{ij}^{ext} = q^{min}. \tag{4.26}$$

4.6.3.4. Dynamische Latenzmodellierung

Neben der Modellierung der statischen Latenz der Bitübertragungsschicht kann der Latenzgraph G_L zur Modellierung der dynamischen Latenzen höherer Protokollschichten genutzt werden. Dies ist die Grundlage für die Vergrößerung der temporären Entkopplung über den statischen Lookahead hinaus, ohne die Kausalität zu verletzen. Zur Beschreibung der Methode werden zunächst die Teilgraphen G_L^v und G_L^r von G_L separat betrachtet.

Mit Hilfe des Graphen G_L^v können für ein Routermodul μ_i zum lokalen Zeitpunkt τ_i die *minimalen internen Vorwärtspfadlatenzen* $d^{fwd,int}(\tau_i)$ bzgl. der Pufferartefakte $fi_{ij} \in FI_i$ berechnet werden. Unter der Annahme, dass generell von jedem Pufferartefakt $fo_{ik} \in FO_i$ eine kausale Abhängigkeit zu jedem $fi_{ij} \in FI_i$ existiert, berechnet sich $d_{ij}^{fwd,int}(\tau_i)$ für ein Artefakt fi_{ij} über das Minimum der Latenzen aller in fi_{ij} eingehenden Kanten des Graphen G_L^v:

$$d_{ij}^{fwd,int}(\tau_i) = \min_{\forall fo_{ik} \in FO_i} \{d(fo_{ik}, fi_{ij})(\tau_i)\}. \tag{4.27}$$

Äquivalent dazu können mit G_L^r die *minimalen internen Rückwärtspfadlatenzen* $d_{ij}^{bwd,int}(\tau_i)$ für Artefakte fo_{ij} über

$$d_{ij}^{bwd,int}(\tau_i) = \min_{\forall fi_{ik} \in FI_i} \{d(fi_{ik}, fo_{ij})(\tau_i)\} \tag{4.28}$$

berechnet werden. Mit der für einen NoC Router typischen Eigenschaft, dass Flits, die über einen logischen Puffer einer Verbindung con_{ij} in μ_i eintreffen, nicht wieder direkt über diese Verbindung zurückgeschickt werden können, gilt für $j \equiv k$:

$$d(fo_{ik}, fi_{ij}) \equiv d(fi_{ik}, fo_{ij}) \equiv \infty.$$

Unter der Voraussetzung das alle anderen Latenzen endlich sind, reduziert sich die Menge der in Ausdruck 4.27 zu berücksichtigenden ausgehenden Pufferartefakte FO_i auf die Teilmenge $FO_i \backslash \{fo_{ik}\}_{i=k}$. Äquivalent reduziert sich die Menge der in Ausdruck 4.28 zu berücksichtigenden Puffereingangsartefakte auf $FI_i \backslash \{fi_{ik}\}_{i=k}$.

Bei alleiniger Betrachtung von μ_i ist eine Beeinflussung des für μ_i *sichtbaren Zustands* der logischen Puffer genau dann vorhanden, wenn μ_i selbst eine beliebige Zustandsvariable seiner Pufferartefakte modifiziert. Mit den für die Pufferartefakte einer Verbindung con_{ij} bestimmten internen Latenzen auf dem Vorwärts- und auf dem Rückwärtspfad kann das diskrete Zeitintervall ab dem lokalen Zeitpunkt τ_i *dynamisch* bestimmt werden, für das auf beiden Pfaden eine solche Beeinflussung des Zustands der Verbindung con_{ij} ausgeschlossen ist. Für dieses Zeitintervall gilt:

$$d_{ij}^{int}(\tau_i) = \min\{d_{ij}^{fwd,int}(\tau_i), d_{ij}^{bwd,int}(\tau_i)\} \tag{4.29}$$

$d_{ij}^{int}(\tau_i)$ wird als *minimale interne Pfadlatenz* von Router μ_i in Richtung von Verbindung con_{ij} zum lokalen Zeitpunkt τ_i bezeichnet. Diese entspricht den Latenzen, welche durch die Sicherungs- und die Vermittlungsschicht erzeugt werden.

4.6.3.5. Dynamischer Lookahead und lokales Quantum

Mit Hilfe der internen und externen Pfadlatenzen kann in einem Modul μ_i für jede Verbindung, con_{ij} auf die μ_i Zugriff hat, ein *dynamischer Lookahead* berechnet werden. Der dynamische Lookahead einer Verbindung con_{ij} im Modul μ_i spezifiziert das diskrete Zeitintervall ausgehend von einem diskreten lokalen Zeitpunkt τ_i, für das auf dem Vorwärts- und auf dem Rückwärtspfad eine Beeinflussung des für Modul μ_j *sichtbaren Zustands* durch Modul μ_i ausgeschlossen werden kann. Der dynamische Lookahead einer Verbindung con_{ij} im Modul μ_i entspricht damit der Summe aus der internen Pfadlatenz von μ_i in Richtung con_{ij} und der externen Pfadlatenz von con_{ij}:

$$\Delta l_{ij}(\mu_i, \tau_i) = d_{ij}^{int}(\tau_i) + d_{ij}^{ext} = d_{ij}^{int}(\tau_i) + q. \tag{4.30}$$

Unter der Voraussetzung, dass zwei Module μ_i und μ_j, die über con_{ij} miteinander verbunden sind, die gleiche lokale Zeit $\tau = \tau_i = \tau_j$ erreicht haben, kann der diskrete Zeitpunkt τ_{ij}^{next}, nach dessen Ablauf der Zustand von con_{ij} zwischen μ_i und μ_j das nächste Mal synchronisiert werden muss, über die Summe aus der aktuellen Zeit τ und dem sog. *lokalen Quantum* $q_{ij}(\tau)$ berechnet werden:

$$\tau_{ij}^{next} = \tau + q_{ij}(\tau) = \tau + \min\{\Delta l_{ij}(\mu_i, \tau), \Delta l_{ij}(\mu_j, \tau)\}. \tag{4.31}$$

Das lokale Quantum $q_{ij}(\tau)$ muss generell durch μ_i und μ_j kooperativ bestimmt werden. Mit kooperativer Berechnung ist hier gemeint, dass benachbarte Module sich gegenseitig zunächst alle notwendigen Informationen zur Verfügung stellen müssen, um dann *unabhängig voneinander* ein *identisches* lokales Quantum bestimmen zu können. In Ergänzung zum Basisverfahren aus Abschnitt 4.6 muss dabei der Zeitstempel einer Kontrolltransaktion immer auf den lokal vorhandenen dynamischen Lookahead gesetzt werden. In Analogie zum Basisverfahren

kann die kooperative Bestimmung zwischen μ_i und μ_j in drei Schritten geschehen:

1. Das Modul μ_i sendet nach Abarbeitung des letzten lokalen Quantums genau eine Kontrolltransaktion an μ_j. Diese signalisiert μ_j, dass das letzte lokale Quantum durchschritten wurde.
2. μ_i wartet so lange, bis sich von μ_j mindestens eine Kontrolltransaktion im zugehörigen Kontrollpuffer $c_{ij} \in C$ befindet.
3. Auf Basis der gestrichenen Füllstandswerte, die mit der Kontrolltransaktion empfangen wurden, kann μ_i den Zustand der eingehenden und ausgehenden logischen Puffer von con_{ij} durch Anwendung der Ausdrücke 4.22 und 4.24 aktualisieren. Dann berechnet μ_i das nächste lokale Quantum $q_{ij}(\tau)$ entsprechend Ausdruck 4.31. Danach kann μ_i **aus Sicht von** μ_j das lokale Quantum $q_{ij}(\tau)$ bis zur Erreichung von τ_{ij}^{next} abarbeiten.

4.6.3.6. Kombinierter Ausführungsmechanismus bei Latenzprädiktion

Im Detail kann der kombinierte Ausführungsmechanismus mit Latenzprädiktion für Modul μ_i anhand folgender neuer oder modifizierter Aktionen beschrieben werden:

- *calcDL()*: $\forall con_{ij} \in CON_i | \tau_{ij}^{next} = \tau_i$: Berechne den dynamischen Lookahead $q_{ij}(\tau)$ entsprechend Ausdruck 4.30.
- *sendCT()*: $\forall con_{ij} \in CON_i | \tau_{ij}^{next} = \tau_i$: Sende eine Kontrolltransaktion an das Modul μ_j über con_{ij}. Setze dabei den Zeitstempel der Kontrolltransaktion auf den zuletzt mit *calcDL()* für con_{ij} berechneten dynamischen Lookahead $\Delta l_{ij}(\mu_i, \tau_i)$. Daneben überträgt die Kontrolltransaktion nach μ_j die aktuellen lokalen Füllstandswerte jedes logischen Puffers zwischen μ_i und μ_j.
- *checkCTD()*: $\forall con_{ij} \in CON_i | \tau_{ij}^{next} = \tau_i$: Gib eine 0 zurück, falls ein Kontrollpuffer $c_{ij} \in C_i$ existiert mit $|c_{ij}| = 0$. Ansonsten gibt eine 1 zurück.
- *updateD()*: $\forall con_{ij} \in CON_i | \tau_{ij}^{next} = \tau_i$: Berechne den nächsten Synchronisationspunkt τ_{ij}^{next} für alle logischen Puffer von con_{ij} entsprechend Ausdruck 4.31. Aktualisiere anschließend den Zustand von fi_{ij} bzw. fo_{ij} entsprechend der Ausdrücke 4.22 und 4.24. Die gestrichenen Werte wurden mit der letzten Kontrolltransaktion empfangen.
- *popCTD()*: $\forall con_{ij} \in CON_i | \tau_{ij}^{next} = \tau_i$: Lösche die oberste Kontrolltransaktion aus c_{ij}.
- *setNextModuleSyncTime()*: Setze den nächsten Synchronisationspunkt des Moduls τ_i^{sync} auf $\min_{\forall con_{ij} \in CON_i} \{\tau_{ij}^{next}\}$.

Abb. 4.47 illustriert den modifizierten leichtgewichtigen Scheduler. Im neuen Scheduler existiert prinzipiell die gleiche Zustandsfolge wie beim Basisverfahren. Insbesondere die Implementierung des hierarchischen $s^{schedule}$ Zustands ist identisch zum Basisverfahren. Für den neuen Ausführungsmechamismus ist daher ausschließlich die neue Funktionalität innerhalb der Aktionen auf dem Toplevel verantwortlich.

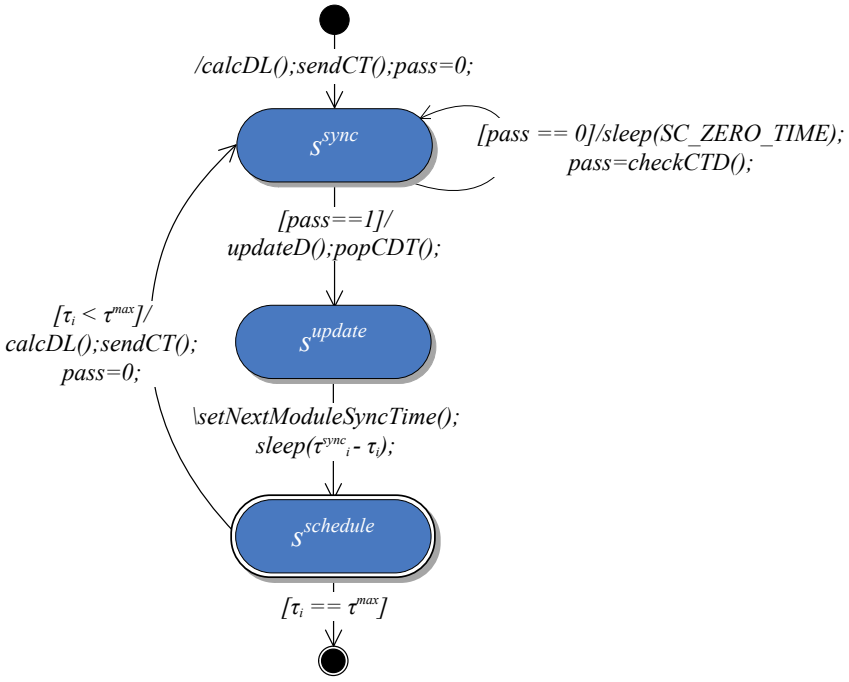


Abbildung 4.47.: Zustandsmaschine bei Latenzprädiktion

4.6.4. Integration transaktionsbasierter Kommunikation

Die Modellierungsstrategie wurde mit dem symmetrischen asynchronen Kernel aus Abschnitt 4.4 kombiniert. Dazu waren einige wenige Ergänzungen am parallelen SystemC Kernel notwendig.

Während die Partitionierung eines RTL Modells das Schneiden von externen Signalen und eine Duplikation der Datenstrukturen geschnittener Signale zur Folge hat (vgl. Abschnitte 4.4.2 und 4.4.4), resultiert die Partitionierung eines TLM Modells hingegen im Schneiden von Socketverbindungen, die bereits aus zwei separaten Artefakten in Form eines Initiator und eines Target Sockets bestehen.

Folglich ist keine Duplikation notwendig, sondern nur eine Adaption. In der aktuellen Implementierung existiert dazu eine verteilte Version der *nb_ transport_fw()* Methode. Bei Aufruf von *nb_transport_fw()* auf einem TLM Initiator Socket wird der Inhalt der übergebenen Transaktion zunächst in eine *Transaktionsnachricht* kopiert:

Definition 4.28 *(**Transaktionsnachricht**): Eine Transaktionsnachricht, die von lp_i an lp_j über l_{ij} übertragen wird, enthält folgende Informationen:*

- *eine id, welche die TLM Socketverbindung identifiziert,*
- *ein Datenfeld data, welches die Transaktion speichert.*

Die Nachricht wird dann in das Ausgangssocket eines logischen Links kopiert. Der durch die Transaktion im Modell dynamisch reservierte Speicher wird dabei deallokiert. Beim Empfang wird die Transaktionsnachricht im Target Socket vom Eingangssocket des logischen Links gelesen. Anschließend wird eine neue Transaktion lokal allokiert. Diese Transaktion wird zum zugehörigen Modul anhand der dort lokal implementierten *nb_transport_fw()* Methode direkt übermittelt.

Für die Umsetzung des Verfahrens wurde die *insert_message()* Methode des Adapters aus Abb. 4.19 so erweitert, dass sie neben dem synchronisierten Modus einen weiteren unsynchronisierten Modus unterstützt. Während im synchronisierten Modus Nachrichten entsprechend ihres Zeitstempels weitergeleitet werden, ist im neuen unsynchronisierten Modus die Zeitsynchronisation vollständig deaktiviert. Daten werden dann direkt weitergeleitet, sobald der Kernel den Kontext vom Modell erhalten hat und das nächste Mal die *dispatch()* Aktion ausführt (z.B. bei Aufruf von *sleep()*). Die ist möglich, weil TL Modelle, die entsprechend der beschriebenen Modellierungstechnik beschrieben sind, sich selbst synchronisieren.

Aktuell ist für Signalnachrichten standardmäßig die synchronisierte Weiterleitung aktiviert. Transaktionsnachrichten werden hingegen standardmäßig unsynchronisiert weitergeleitet. Ein logischer Link, der Nachrichten ausschließlich im unsynchronisierten Modus überträgt, kann als deadlock-unkritisch klassifiziert werden, da die Vermeidung von Deadlocks nicht in der Verantwortung des Kernels liegt. Ansonsten ergibt sich die Kritikalität aus Definition 4.10.

Die Implementierung von *sleep()* ist vollständig identisch zu Implementierung der originalen *wait()* bzw. *next_trigger()* Methoden des sequentiellen Kernels. Wegen der Reduktion der kausalen Kopplung von Deltacycles auf Timedcyles genügt eine rein lokale Implementierung, eine Synchronisation mit anderen logischen Prozessen erfolgt nicht.

4.6.5. Fallstudie I

In der ersten Fallstudie wurde die Basisvariante des Modellierungsansatzes hinsichtlich der zwei Aspekte I) Genauigkeit und II) Performanz betrachtet. In Bezug auf die Genauigkeit wurde zunächst untersucht, inwieweit es mit dem Modellierungsansatz überhaupt möglich ist, spezielle NoC Charakteristika wiederzugeben. Darauf basierend wurde der Trade-off zwischen erzielbarer Genauigkeit und erzielbarer Performanz näher beleuchtet. Als Grundlage für die Untersuchung diente das in Abb. 4.6.3.2 bereits illustrierte und neu entwickelte TL Modell des Hermes Routers [203] sowie das um die TL Router ergänzte HeMPS Modell [75].

4.6.5.1. Wiedergabe typischer Network-on-Chip Charakteristika

Ein Effekt, der typischerweise auftritt, sobald ein NoC-basiertes MPSoC an die Grenzen seiner Leistungsfähigkeit stößt, ist die Kongestion der Puffer im NoC. Diese tritt dann auf, wenn Nachrichten mit einer höheren Datenrate in das Netzwerk injiziert werden, als sie von diesem übertragen werden können. Eine limitierte Übertragungsrate kann sowohl durch die empfangenden Verarbeitungseinheiten, als auch durch die Implementierung des NoCs selbst hervorgerufen werden. Im ersten Fall bilden die Empfänger aufgrund ihrer hohen Verarbeitungslatenz den Flaschenhals. In zweiten Fall bildet das NoC den Flaschenhals. Ursache ist beispielsweise ein ineffizientes Routingverfahren, welches zu Zugriffskonflikten während der Übertragung führt. Im Rahmen dieser Untersuchung lag der Fokus auf dem NoC.

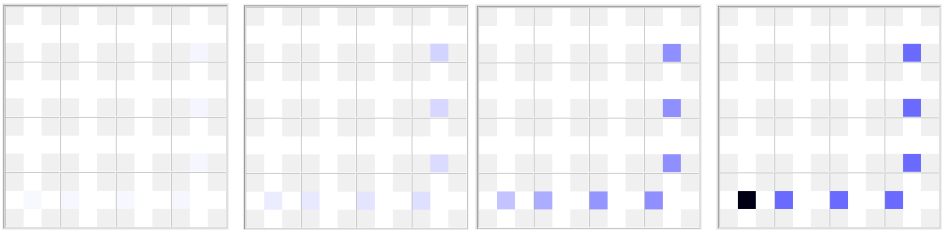
Kongestion der Routerpuffer

In einem ersten Experiment wurden in jeden Hermes Router eines 4x4 NoCs Nachrichten mit unterschiedlichen Flit-Injektionsraten (Flit-IR) injiziert[18]. Bei den Messungen wurden zwei unterschiedliche Traffic Patterns verwendet. Beim sog. Single Sender Pattern wurden nur Pakete in Router 0/0 injiziert mit Router 3/3 als Zieladresse. Beim sog. Transpose Pattern wurden in jeden Router Pakete mit folgendem Adressierungsschema injiziert: $destX = sourceY$ und $destY = sourceX$.

In den Empfängerroutern wurden eingehende Pakete generell ohne Verarbeitungslatenz direkt wieder ausgelesen. Die Erzeugung des Netzwerkverkehrs erfolgte anhand abstrakter verzögerungsfreier Prozessoreinheiten. Für die Messungen wurden Puffergröße und Paketlänge jeweils auf 16 Flits gesetzt. Das glo-

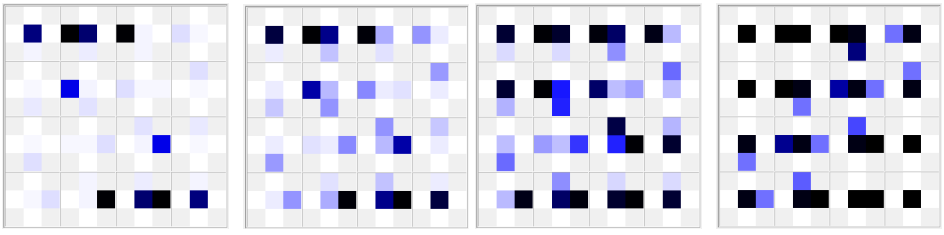
[18] Eine Injektionsrate von 100% (50%, 25%...) entspricht der Generierung eines Flits in jedem (jedem 2., jedem 4., ...) Taktzyklus und dessen sofortiger Injektion, sobald der Ausgangspuffer des Network Interfaces genügend Speicherplatz bietet.

bale Quantum wurde nicht variiert und entsprach der Taktzykluszeit. Die Messergebnisse sind in den Abbildungen 4.48 und 4.49 illustriert.



(a) 10% Flit-IR (b) 25% Flit-IR (c) 50% Flit-IR (d) 100% Flit-IR

Abbildung 4.48.: Kongestion der Puffer beim Single Sender Pattern



(a) 10% Flit-IR (b) 25% Flit-IR (c) 50% Flit-IR (d) 100% Flit-IR

Abbildung 4.49.: Kongestion der Puffer beim Transpose Pattern

Die farblich unterlegten Quadrate repräsentieren die Eingangspuffer der Router. Je dunkler die Farbgebung eines Puffers, desto größer ist der gemessene durchschnittliche Füllstand über die gesamte Simulationsausführung. In beiden Abbildungen ist deutlich der Zusammenhang zwischen steigender Injektionsrate und steigenden Pufferfüllständen in Abhängigkeit des Traffic Patterns zu erkennen. Aufgrund der Übertragungslatenz des Hermesrouters ist es nicht möglich, die Puffer zwischen den Routern mit einem einzigen Datenstrom und 100% Injektionsrate voll auszulasten (siehe Abb. 4.48 (d)). Erst mit dem Vorhandensein mehrerer gleichzeitiger Datenströme (vgl. Abb. 4.49), die sich gegenseitig kreuzen, ist dies möglich.

Genauigkeitsverlust bei Aufzeichung typischer NoC Charakteristika

In einem zweiten Experiment wurden in jeden Hermes Router eines 8x8 NoCs Nachrichten mit einer 50% Flit-Injektionsrate injiziert. Die Adressen der Zielrouter wurden dabei per Zufall gewählt und waren gleichverteilt (Random Pat-

tern). Für die Messungen wurden Modellparameter wie Puffergröße, Paketgröße und das globale Quantum variiert. Die abstrakten Prozessoreinheiten wurden zur Aufzeichnung von Performanzdaten wie der Verzögerung einzelner Pakete und dem Datendurchsatz genutzt.

Die Abb. 4.50 und 4.51 zeigen die gemessene durchschnittliche Abweichung der Paketverzögerung und des Durchsatzes, die entsteht, wenn anstatt einer zyklenweisen Synchronisation mit einem globalen Quantum von $q = \tau^{cycle} = 10$ ns größere Werte für q gewählt werden. In den Abbildungen sind einzelne vertikale Segmente, welche unterschiedliche Paketgrößen repräsentieren, nichtkumulativ.

Wie erwartet steigt die durchschnittliche Abweichung in beiden Fällen mit der Größe des globalen Quantums an. Für eine Puffergröße von acht Flits und einer Paketgröße von 32 Flits resultiert eine Vergrößerung des globalen Quantums von 20 ns auf 80 ns in einem Anstieg der Abweichung von Verzögerung und Durchsatz von ±4.0% auf ±57.4% bzw. ±3.8% auf ±35.6%. Die Ursache für den Anstieg ist die Generierung zusätzlicher *synthetischer Kongestion*, welche durch die temporäre Entkopplung ausgelöst wird. Da die Pufferfüllstände nur einmal pro Quantum synchronisiert werden, werden bereits volle Puffer mit erhöhter Wahrscheinlichkeit zu spät ausgelesen. Umgekehrt werden bereits leere oder nicht mehr volle Puffer mit erhöhter Wahrscheinlichkeit zu spät beschrieben.

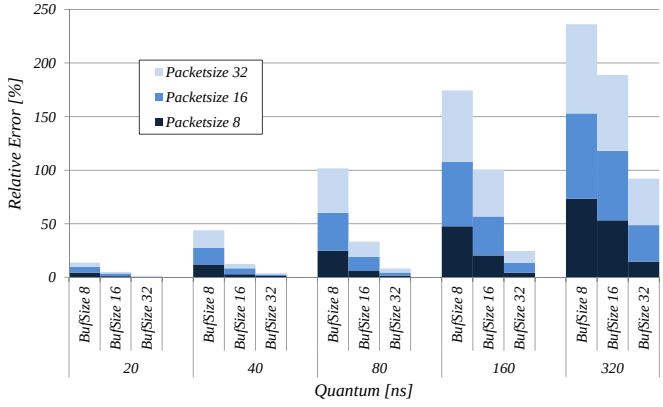


Abbildung 4.50.: Durchschnittliche Abweichung der Paketverzögerung (Random Pattern)

Wie sich den Abbildungen auch entnehmen lässt, sinkt die durchschnittliche Abweichung mit steigender Pufferkapazität. Offensichtlich reduzieren größere Puffer die Gefahr von (synthetischer) Kongestion. Beispielsweise reduziert sich die durchschnittliche Abweichung der Verzögerung, im Fall eines globalen Quan-

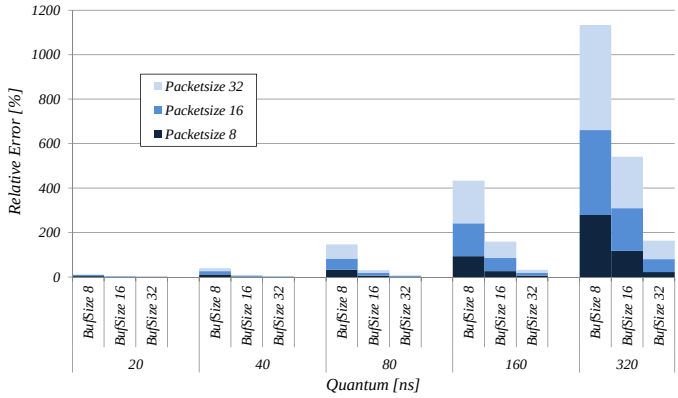


Abbildung 4.51.: Durchschnittliche Abweichung des Durchsatzes (Random Pattern)

tums von $q = 20$ ns und einer Paketgröße von 32 Flits, von $\pm 4.0\%$ auf nur $\pm 0.49\%$, wenn man die Pufferkapazität von acht auf 32 Flits anhebt. Entsprechend reduziert sich die Abweichung des Durchsatzes von $\pm 3.8\%$ auf $\pm 0.37\%$.

Mit den gegebenen Paketgrößen wurden kleinere durchschnittliche Abweichungen im einstelligen Prozentbereich für $F_{max} < \frac{F_b}{2}$ gemessen. Dabei ist F_{max} die Anzahl an Flits, die maximal zwischen zwei Routern innerhalb eines Quantums q übertragen werden können und F_b die Kapazität der Puffer. Kleinere Paketgrößen resultierten meist in kleineren Abweichungen. Im speziellen Fall des Hermes NoC gilt:

$$F_{max} = \frac{q}{\tau^{cycle}} < \frac{F_b}{2}. \tag{4.32}$$

4.6.5.2. Performanz und zeitliche Genauigkeit

Um den Trade-off zwischen Performanz und zeitlicher Genauigkeit im Fall einer vollständigen MPSoC Simulation zu untersuchen, wurden unterschiedliche Varianten eines 8x8 HeMPS Modells auf einer unterschiedlichen Anzahl an Kernen einer bestimmten Zielplattform ausgeführt. Die Module und deren verfügbare Abstraktionsebenen (RTL, TL, PA-CAL und *Mixed Level (ML)*) sind in Tab. 4.3 zusammengefasst.

Sowohl für den Router als auch für das Network Interface existieren transaktionsbasierte Modelle. Die ML Beschreibung des Network Interfaces dient als Ad-

apter zur Übersetzung zwischen einem signalbasierten PE und dem TL Router. Dabei werden intern zwei *SC_METHOD* Prozesse zur Übersetzung von Signalen in Transaktionen verwendet. Der leichtgewichtige Scheduler dient ausschließlich zur Synchronisation.

Modul	Abstraktionsebene
Processing Element (PE)	RTL, PA-CAL
Network Interface (NI)	RTL, ML
Router (RO)	RTL, TL

Tabelle 4.3.: Modellelemente und Abstraktionebenen

Die Module wurden soweit wie möglich in gleich große Partitionen aufgeteilt. Die Simulation wurde für 10^6 Taktzyklen bei einer simulierten Taktfrequenz von 100 MHz und entsprechend einer Zykluszeit von $\tau^{cycle} = 10$ ns ausgeführt. Die PEs führten dabei mehrere MPEG Decoder Pipelines aus.

Als Zielplattformen kamen sowohl der SCC als auch ein cachkohärenter SHM Multiprozessor (Core i7 930) mit vier physikalischen und acht virtuellen (Hyperthreading) Kernen zum Einsatz. Der Core i7 930 war mit 2.8GHz getaktet und verfügte im Unterschied zum SCC über Hardware Cachekohärenz. Da die Kernelimplementierung auf dem asynchronen Kernel aus Abschnitt 4.4 basiert, wurde für die Konfiguration der Simulation die teilautomatisierte Werkzeugkette aus Abschnitt 4.4.8 verwendet.

Zur Quantifizierung der zeitlichen Genauigkeit unterschiedlich konfigurierter Modelle im Vergleich zur RTL Referenz wurden Zeitstempel signifikanter Ereignisse wie z.B. Empfangszeiten von Nachrichten aufgezeichnet. Diese Zeitstempel wurden dann zur Bestimmung der relativen zeitlichen Abweichung der abstrakteren Simulationsmodelle von der RTL Referenz verwendet. Dazu wurde der zeitliche Fehler jeweils identischer Ereignisse berechnet und daraus der Durchschnitt gebildet.

Die Messergebnisse auf dem SCC und dem Core i7 930 sind in den Abb. 4.52 und 4.53 dargestellt. In beiden Fällen ist sowohl die Beschleunigung rein sequentieller Simulation als auch die durch Parallelisierung zusätzlich erzielte Beschleunigung dargestellt. Bei der Core i 930 Workstation wird außerdem zwischen einer Simulation auf vier Kernen (ohne Hyperthreading) und acht Kernen (mit Hyperthreading) unterschieden.

Sowohl auf dem SCC als auch auf dem Core i 930 wird mit reiner RTL Simulation maximale Genauigkeit aber generell die geringste Beschleunigung erzielt (22.6x/1.0 auf dem SCC respektive 4.6x/2.9/1.0 auf dem Core i7 930). Der Grund für die signifikant bessere Beschleunigung auf dem SCC ist die vergleichsweise

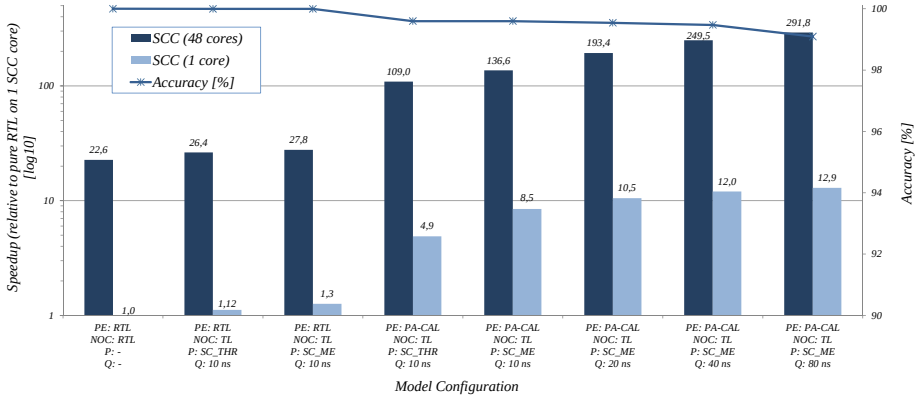


Abbildung 4.52.: Charakteristika von Performanz und Genauigkeit der Simulation eines 8x8 HeMPS Modells auf dem SCC

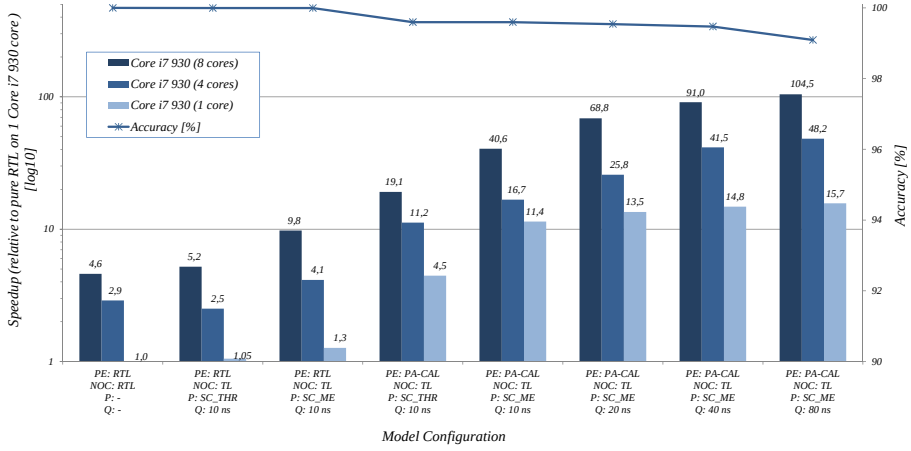


Abbildung 4.53.: Charakteristika von Performanz und Genauigkeit der Simulation eines 8x8 HeMPS Modells auf dem Core i7 930

geringe Rechenleistung der SCC Kerne in Verbindung mit der insgesamt höheren Anzahl an Kernen.

Tauscht man die RTL Beschreibungen der Router und Network Interfaces durch deren TL bzw. ML Beschreibungen aus und belässt das Quantum bei der Zykluszeit, so tritt ein leichter durchschnittlicher Genauigkeitsverlust von 0.005% auf. Dieser ist auf die synchrone Semantik der leichtgewichtigen Scheduler zurückzuführen, welche keine Deltacycles unterstützt, wodurch es evtl. zu kleinen Timingfehlern im Vergleich zur RTL Referenz kommen kann.

Mit *SC_THREAD* Schedulern ist in fast allen Fällen ein leichter Performanzgewinn zu verzeichnen. Nur auf dem Core i 930 geht die Beschleunigung bei vier Kernen von 2.9x auf 2.5x zurück. Eine Ursache dieses Rückgangs ist der hohe Overhead für das Scheduling der *SC_THREAD* Co-Routinen. Bei Verwendung der effizienteren *SC_METHOD* Callbacks konnte hingegen in allen Fällen eine signifikant bessere Beschleunigung gemessen werden (27.8x/1.3x auf dem SCC respektive 9.8x/4.1x/1.3x auf dem Core i7 930). Die Zuwächse im rein sequentiellen Fall sind generell beschränkt, da die Hauptrechenlast weiterhin durch die RTL Plasmakerne generiert wird. Dieser Overhead ist um einiges Größer als die durch den Austausch der Router und Network Interfaces erzeugte Varianz in der Rechenlast.

Tauscht man die RTL PEs durch PA-CAL Beschreibungen aus, so resultiert dies in einem Rückgang der Genauigkeit auf ca. 99.6% und in einem generellen Zuwachs der Beschleunigung, selbst wenn in den Routern und Network Interfaces *SC_THREAD* Scheduler eingesetzt werden. Ein Austausch von *SC_THREAD* Schedulern durch *SC_METHOD* Scheduler fällt wegen der geringeren Rechenlast der PA-CAL Beschreibungen nun viel stärker ins Gewicht: Während beispielsweise die parallele Beschleunigung mit RTL PEs auf dem SCC nur von 26.4x auf 27.8x zunimmt, steigt sie mit PA-CAL PEs von 109.0x auf 136,6x.

Wendet man nun zusätzlich temporäre Entkopplung an und vergrößert das globale Quantum, so wird im Fall von $q = 80$ ns eine maximale Beschleunigung von 291.8x auf dem SCC und 104.5x auf dem Core i7 930 erreicht. Dabei sinkt die Genauigkeit im betrachteten Szenario im Vergleich zur RTL Referenz lediglich auf 99.09% ab. Der nur geringe Verlust ist auf die eher rechenintensiven MPEG Pipelines zurückzuführen. Aufgrund der relativ seltenen Kommunikation über das NoC ist der zeitliche Genauigkeitsverlust im Vergleich zur RTL Referenz auch bei einem Quantum von $q = 80$ ns stark limitiert.

4.6.6. Fallstudie II

In der zweiten Fallstudie wurde die Anwendbarkeit der Methode zur dynamischen Latenzprädiktion untersucht. Zur Anwendung der Methode auf das Hermes NoC müssen folgende drei Schritte durchgeführt werden:

1. Analyse des Verhaltens.
2. Annotation von Verzögerungen an die Verhaltensbeschreibungen der leichtgewichtigen Prozesse.
3. Entwicklung einer Funktion zur Berechnung des dynamischen Lookaheads auf Basis der annotierten Verzögerungen.

Dabei ist insbesondere hervorzuheben, dass der dynamische Lookahead, im Gegensatz zur allgemeinen Beschreibung aus den vorigen Abschnitten, nicht akkurat berechnet, sondern *konservativ geschätzt* wird. Die mögliche Performanzsteigerung durch temporäre Entkopplung ist dadurch zusätzlich limitiert, im Vergleich zum Basisverfahren werden Kausalitätsverletzungen jedoch vollständig verhindert.

4.6.6.1. Analyse des Verhaltens im Hermes Router

Die Aufgaben der einzelnen Elemente des Hermes Routers (Puffer, Crossbar, Input Controllern und Switch Controller) wurden bereits in Abschnitt 4.6.3.2 grob erläutert. Abb. 4.54 illustriert auf das Notwendigste reduzierte Versionen der beiden Controller FSMs. Im Detail ist das Verhalten wie folgt:

Angenommen, ein Input Controller befindet sich im *S_IDLE* Zustand. Sobald sich ein Header Flit eines Pakets im zugehörigen Eingangspuffer befindet, sendet der Input Controller eine Anfrage zur Weiterleitung aller Flits des Pakets zum Switch Controller. Der Input Controller wechselt dann in den *S_WAIT* Zustand und wartet auf eine Bestätigung durch den Switch Controller.

Ein Arbitrierungs- und Routingzyklus für einen beliebigen Input Controller ist erst nach mindestens einer vollständigen Iteration von *S_IDLE* nach *S_END* des Switch Controllers abgeschlossen: Der Switch Controller kann sich zum Zeitpunkt der Anfrage eines Input Controllers *A* in einem beliebigen Zustand zwischen *S_IDLE* und *S_END* befinden. Dies ist dann der Fall, wenn der Switch Controller bereits eine Anfrage eines Input Controllers *B* bearbeitet, während die Anfrage des Input Controllers *A* eintrifft. Mehr als eine Iteration des Switch Controller ist möglich, wenn andere Input Controller trotz der Anfrage des Input Controllers *A* aufgrund des Schedulingschemas mit höherer Priorität ausgewählt werden. Ein weiterer Grund für mehr als eine Iteration ist ein bereits belegter Ausgang, was in einem Abbruch des Routingvorgangs im Switch Controller im Zustand *S_ROUTE* resultiert.

Sobald ein Input Controller erfolgreich arbitriert wurde, werden die Header und Payload Flits in den Zuständen *S_HEAD* und *S_PAYLOAD* übertragen. Im Fall von Pufferkongestion ist es möglich, dass der Input Controller längere Zeit in einem dieser Zustände verharrt. In den Zuständen *S_END* und *S_END2* signalisiert der Input Controller das Ende der Übertragung an den Switch Controller und wechselt schließlich zurück in *S_IDLE*.

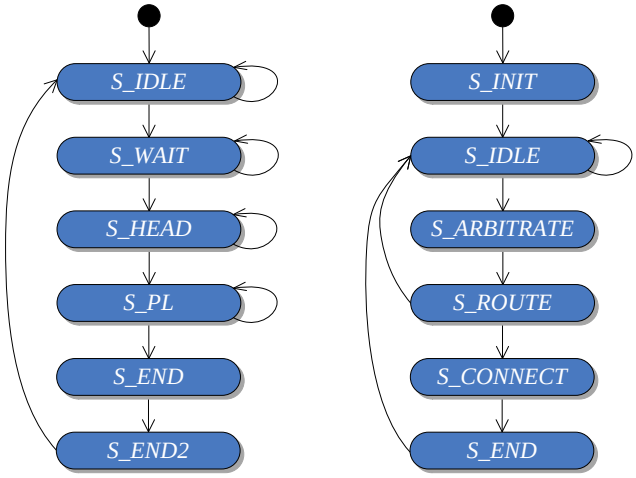


Abbildung 4.54.: Input Controller FSM (links) und Switch Controller FSM (rechts)

4.6.6.2. Annotation von Verzögerungen

Wie bereits erwähnt, können die Elemente des Hermes Routers den unteren drei Schichten des OSI Referenzmodells zugeordnet werden. In diesem Kontext wird eine annotierte Latenz an eine der genannten Komponenten als ein Zeitfenster in der Größenordnung von Taktzyklen definiert, für das eine Interaktion mit dem darunterliegenden OSI Layer (im Fall der untersten Schicht mit dem benachbarten Router) ausgeschlossen ist. Annotationen auf einer konkreten Schicht haben folgende Bedeutung:

- **Bitübertragungsschicht**: Diese Schicht wird durch eine konstante Verzögerung d^{buf} von einem Taktzyklus modelliert. Eine zustandsabhängige Annotation ist nicht notwendig.
- **Sicherungsschicht**: Angenommen, die FSM eines Input Controllers befindet sich in einem beliebigen der in Abb. 4.54 gezeigten Zustände. Dann repräsentiert eine Annotation an eine Transition die konservativ geschätzte

minimale Anzahl an Taktzyklen, bis zur nächsten verlustlosen Flitübertragung (Übergang nach *S_HEAD* oder *S_PL*). Arbitrierungsverzögerungen werden noch nicht berücksichtigt. Der aktuelle Schätzwert der Verzögerung $d^{ic}(\tau)$ einer Input Control FSM ist durch den Wert der Annotation an der nächsten zu nehmenden Transition gegeben.

- **Vermittlungsschicht**: Angenommen, die FSM des Switch Controllers befindet sich in einem beliebigen der in Abb. 4.54 gezeigten Zustände. Dann repräsentiert eine Annotation an eine Transition die konservativ geschätzte minimale Anzahl an Taktzyklen, bis der aktuelle Arbitrierungszyklus vorbei ist (Übergang nach *S_END*). Der aktuelle Schätzwert der Verzögerung $d^{sc}(\tau)$ der Switch Control FSM ist durch den Wert der Annotation an der nächsten zu nehmenden Transition gegeben.

Die Layer 1 Verzögerung d^{buf} ist identisch zur externen Pfadlatenz d^{ext} bzw. zum statischen Lookahead (vgl. Abschnitt 4.6.3.3). d^{buf} sowie die Schätzwerte $d^{ic}(\tau)$ und $d^{sc}(\tau)$ können widerum zur Schätzung der Pfadlatenzen bzw. des dynamischen Lookaheads genutzt werden (vgl. Abschnitt 4.6.3.4).

4.6.6.3. Schätzung des dynamischen Lookaheads

Alg. 4.1 illustriert die in der Fallstudie verwendete Schätzfunktion. Die Funktion ist Teil des leichtgewichtigen Schedulers und wird innerhalb der Aktion *calcDL()* angewendet (vgl. Abschnitt 4.6.3.6). Die Funktion schätzt ausschließlich die Latenz auf dem Vorwärtspfad. Auf eine separate Schätzung des Rückwärtspfades kann verzichtet werden. Der Grund sind folgende spezielle Eigenschaften des Hermes Routers:

Zum einen werden im Switch Controller Entscheidungen bzgl. Arbitrierung und Routing grundsätzlich unabhängig von den Pufferfüllstandswerten gefällt. Diese haben keinen Einfluss auf die Funktion des Switch Controllers. Des Weiteren stützt ein Input Controller die Entscheidung, ob im aktuellen Takt ein Flit übertragen werden kann, nur auf die im unmittelbar vorangegangenen Taktzyklus empfangene binäre Zustandsinformation über die Pufferfüllstände (voll/nicht voll).

Für die Berücksichtigung des Rückwärtspfades genügt es daher, wenn man dafür sorgt, dass auf dem Vorwärtspfad nicht nur nach jedem tatsächlich durchgeführten Schreibzugriff auf ein fi Artefakt, sondern zusätzlich vor jedem potentiellen Schreibzugriff eine Synchronisation erfolgt. Zu diesem Zweck wurden die Annotationen in der Input Controller FSM insgesamt um einen Takt zusätzlich dekrementiert. Damit kann die Berechnung der minimalen internen Pfadlatenz in Ausdruck 4.29 wie folgt vereinfacht werden:

$$d_{ij}^{int}(\tau_i) = \min\{d_{ij}^{fwd,int}(\tau_i), d_{ij}^{bwd,int}(\tau_i)\} = d_{ij}^{fwd,int}(\tau_i). \tag{4.33}$$

Algorithm 4.1 Schätzfunktion für den dynamischen Lookahead

```
1:  function ESTIMATE_LOOKAHEAD(InputArtifact fi)
2:      lookahead l = ∞
3:      for all x ∈ InputController do
4:          d^int ← 0
5:          if s(x) = S_WAIT then
6:              d^int ← d^sc(τ)
7:          else if s(x) = S_HEAD ∨ s(x) = S_PL then
8:              if out(x) = fi then
9:                  d^int ← d^ic_x(τ)
10:             else
11:                 d^int ← pc(x) + d^sc_min
12:             end if
13:         else
14:             //all other states:
15:             d^int ← d^ic_x(τ) + d^sc_min
16:         end if
17:         l ← min(l, d^int)
18:     end for
19:     l ← l + d^buf
20: end function
```

Auf dieser Basis führt die Funktion in Alg. 4.1 eine Reduktion des Lookaheads von ∞ auf einen gültigen Wert durch. Zunächst wird durch Iteration über alle Input Controller mit Ausdruck 4.33 die minimale interne Vorwärtspfadlatenz wie folgt berechnet:

Falls sich der Input Controller x im Zustand *S_WAIT* befindet, so wird für $d_{ij}^{int}(\tau_i)$ die konservative Schätzung der aktuellen Latenz des Switch Controllers $d^{sc}(\tau)$ herangezogen. In den Zuständen *S_HEAD* bzw. *S_PL* hängt der geschätzte Latenzwert vom betrachteten fi Artefakt ab. Ist dieses identisch zu dem logischen Puffer, auf den der Input Controller x gerade schreibt (identifiziert durch $out(x)$), so wird als Latenz $d_x^{ic}(\tau)$ angenommen. Andernfalls, ist die Latenz $pc(x) + d_{min}^{sc}$. $pc(x)$ gibt dabei die Anzahl der zur Übertragung ausstehenden Payload Flits zurück und d_{min}^{sc} entspricht der minimalen Dauer eines Routingvorgangs durch den Switch Controller (vier Taktzyklen). In allen anderen Zuständen (*S_IDLE*, *S_END* und *S_END2*) wird die Latenz zu $d_x^{ic}(\tau) + d_{min}^{sc}$ angenommen: In diesen

Fällen kann eine aktive Übertragung ausgeschlossen werden. Für die nächste Übertragung eines Flits muss mindestens ein kompletter Routingvorgang erfolgen. Zu guter Letzt wird in Zeile 19 die Berechnung durch Addition der externen Pfadlatenz bzw. d^{buf} vervollständigt.

4.6.6.4. Experimentelle Evaluation

Zur experimentellen Bewertung wurden mehrere Simulationsläufe mit unterschiedlichen Varianten des HeMPS durchgeführt. Zunächst wurden dazu verschiedene Konfigurationen einer Applikation, die ein Transpose Pattern implementiert, auf die PEs des Modells abgebildet. Dabei wurde jedem PE ein Task mit einem festen Kommunikationspartner auf einem anderen Kern zugeordnet. Mit Hilfe eines Zählers war es möglich, die Wartezyklen zwischen der Übertragung aufeinanderfolgender Pakete einzustellen. Anschließend wurde die Modellierungstechnik für die verwendeten Module ausgewählt. Die Umsetzung des Prädiktionsmechanismus beschränkte sich auf das Routermodell.

Die Module wurden mit einer Granularität von Tiles soweit wie möglich in gleich große Partitionen gruppiert. Jeder dieser Partitionen wurde einem Kern der Zielplattform zugeordnet. Als Zielplattform dienten der SCC und eine Core i7 930 Workstation mit bis zu acht virtuellen Kernen mit jeweils 2.8 GHz. Die Simulation wurde für 10^6 Zyklen bei einer simulierten Taktfrequenz von 100 MHz ausgeführt.

Plattformunabhängige Charakteristika

Tabelle 4.4 fasst gemessene plattformunabhängige Charakteristika zusammen. Sobald ein Synchronisationspunkt einer Verbindung erreicht ist, warten benachbarte Module auf den gegenseitigen Austausch von genau einer Kontrolltransaktion. Dies resultiert darin, dass auch mit aktivierter Prädiktion die Anzahl der insgesamt in der Simulation ausgetauschten Kontrolltransaktionen unabhängig von der Zielplattform und dem Ausführungsmodus (sequentiell oder parallel) ist.

Der Tabelle ist zu entnehmen, dass die durchschnittliche Anzahl der Kontrolltransaktionen θ^{ctrl} die bei deaktivierter Prädiktion pro Router und Socket Link verschickt wird, exakt der Anzahl der simulierten Taktzyklen (1×10^6) entspricht. Die Anwendung von DLP resultiert im Allgemeinen in einer Reduktion der durchschnittlichen Anzahl an Kontrolltransaktionen um 57% bis 66%. Damit steigt die durchschnittliche Anzahl an Taktzyklen, die ein Modul ausführen kann, ohne mit einem benachbarten Modul synchronisieren zu müssen, von 1.0 auf Werte ≥ 2.38.

Tabelle 4.4.: Plattformunabhängige Charakteristika

Modell	Wartezyklen	Prädiktion	$\theta.ctrl$/Router/Link	Zyklen/$\theta.ctrl$
4x4	10^0	aus	1×10^6	1.0
4x4	10^0	an	4.21×10^5	2.38
4x4	10^2	aus	1×10^6	1.0
4x4	10^2	an	4.06×10^5	2.46
4x4	10^5	aus	1×10^6	1.0
4x4	10^5	an	3.58×10^5	2.79
8x8	10^0	aus	1×10^6	1.0
8x8	10^0	an	3.90×10^5	2.56
8x8	10^2	aus	1×10^6	1.0
8x8	10^2	an	3.83×10^5	2.61
8x8	10^5	aus	1×10^6	1.0
8x8	10^5	an	3.37×10^5	2.97

Die tatsächliche Effektivität hängt von der auf dem HeMPS ausgeführten Applikation ab. Im Fall von geringem Aufkommen an Datentransaktionen (größere Anzahl Wartezyklen) ist die durchschnittliche Anzahl an Kontrolltransaktionen generell kleiner als im Fall von hohem Aufkommen (kleinere Anzahl Wartezyklen). Viele Datentransaktionen einer aktiven Übertragung über einen bestimmten routerinternen Pfad erhöhen die Wahrscheinlichkeit, dass sich die betroffenen Input Controller im Zustand *S_HEAD* oder *S_PL* befinden und die Latenz dieses Pfades über Zeile neun in Alg. 4.1 berechnet wird. Da sich die Werte der Annotationen an Transitionen im Input Controller in den Zuständen *S_HEAD* oder *S_PL* nur zwischen null und einem Taktzyklus bewegen, führt dies zu einem erhöhten Aufkommen von Kontrolltransaktionen.

Für alle anderen internen Pfade, die nicht von der aktiven Übertragung betroffen sind und über die aktuell keine Datentransaktionen übermittelt werden, berechnet sich die interne Pfadlatenz ausgehend von einem aktiven Input Controller, über Zeile elf. Dadurch bewegen sich die geschätzten Latenzen in der Größenordnung des Payloads. Allerdings werden diese verhältnismäßig hohen Schätzwerte in nahezu allen Fällen durch geringere Schätzwerte anderer passiver Input Controller im Zustand *S_ILDE*, *S_END* oder *S_END2* wieder reduziert: In

diesem Fall bestimmt sich die Latenz über Zeile 15. Aufgrund der annotierten Werte, die sich in jeder FSM nur im Bereich von ein bis maximal vier Zyklen bewegen, resultiert dies in Gesamtverzögerungen ≤ 7 Zyklen.

Die durchschnittliche Anzahl an Kontrolltransaktionen ist bei aktiver Prädiktion beim 8x8 Modell generell kleiner als beim 4x4 Modell. Der Grund ist die Initialisierungsphase des HeMPS Modells, in der die Tasks zunächst von einem Master PE auf Slave PEs verteilt werden, bevor letztere mit der Ausführung beginnen. In dieser Phase wird die Transpose Anwendung nicht vollständig ausgeführt. Da die Taskverteilung auf einem 8x8 System länger dauert als auf einem 4x4 System, befinden sich die Input Controller der Router länger im *S_IDLE* Zustand. Dadurch entstehen größere dynamische Lookaheads und lokale Quanta.

Plattformabhängige Charakteristika

Zur Bewertung plattformabhängiger Charakteristika wird zunächst die Ausführung auf der Core i7 930 Workstation betrachtet. In Abb. 4.55 ist die gemessene Beschleunigung von paralleler im Vergleich zu sequentieller Simulation illustriert. Bei der sequentiellen Simulation wurde zyklenweise Synchronisation ohne Prädiktion verwendet. Im parallelen Fall wurde die Ausführungsdauer mit und ohne Prädiktion gemessen.

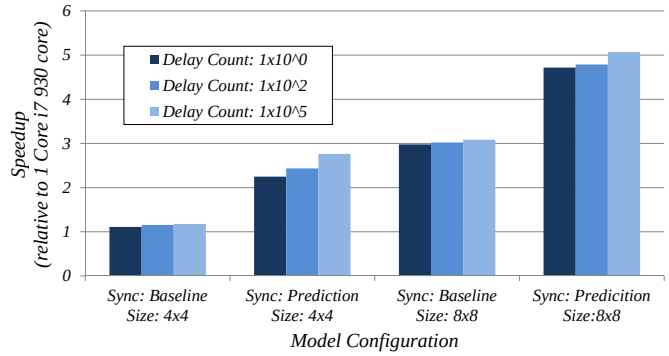


Abbildung 4.55.: Beschleunigung auf dem Core i7 930

Wegen des höheren parallelisierbaren Berechnungsaufwands ist die Beschleunigung beim 8x8 Modell generell höher als beim 4x4 Modell. In beiden Fällen hat DLP eine merkliche Verbesserung der Beschleunigung zur Folge. Beispielsweise beträgt sie beim 8x8 Modell mit zyklenweiser Synchronisation, ohne DLP und 1×10^5 Wartezyklen, $3.08x$. Mit aktivierter DLP steigt die Beschleunigung auf $5.07x$ an. Dies ist gleichzeitig der höchste Wert, der auf der Core i7 930 Workstation gemessen wurde.

Insbesondere bei aktivierter DLP ist eine Abhängigkeit der Beschleunigung vom Datenaufkommen zu erkennen, das durch die Transpose Applikation generiert wird. Beispielsweise reduziert sich die Beschleunigung beim 8x8 Modell und aktivierter DLP von $5.07x$ auf $4.79x$ bzw. $4.72x$, wenn die Anzahl der Wartezyklen in der Applikation von 1×10^5 auf 1×10^2 bzw. 1×10^0 verringert wird.

Für die Untersuchungen auf dem SCC konnten die Modelle stärker zerlegt werden, da der SCC mehr Kerne als die SHM Workstation besitzt. Insgesamt kamen für das 4x4 Modell 16 Partitionen und für das 8x8 Modell 48 Partitionen und damit alle SCC Kerne zum Einsatz. Da mit Tile Granularität partitioniert wurde, bestanden beim 8x8 Modell 16 Partitionen aus zwei Tiles. Abb. 4.56 illustriert die Messergebnisse.

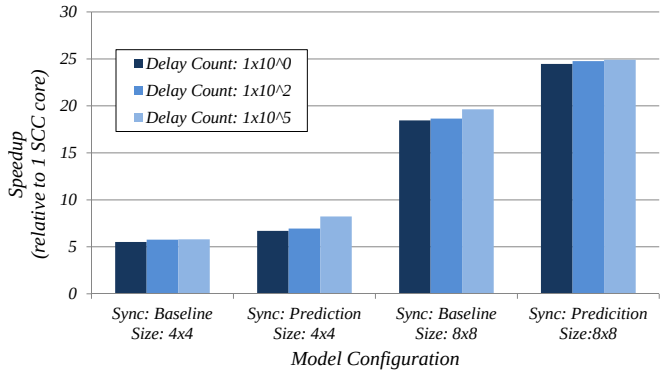


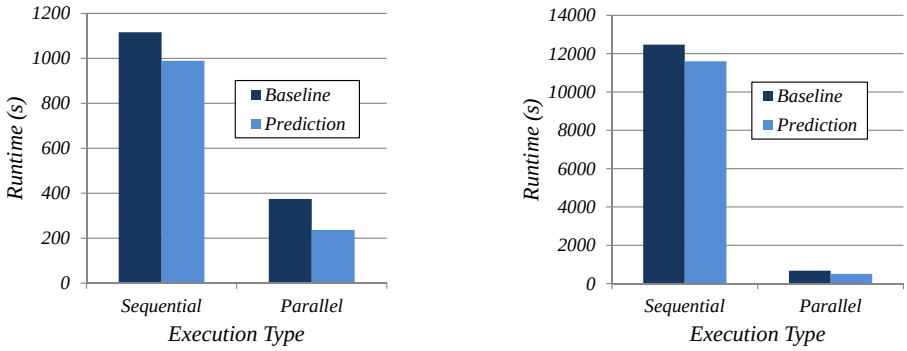
Abbildung 4.56.: Beschleunigung auf dem SCC

Auch auf dem SCC führt DLP zu einer merklichen Verbesserung der Beschleunigung im Vergleich zu zyklenweiser Synchronisation. Während die insgesamt erzielten Beschleunigungen um einiges größer sind als auf dem Core i7 930, bewegen sich die durch DLP zusätzlich erzielten Verbesserungen in der gleichen Größenordnung.

Beim 4x4 Modell mit 1×10^5 Wartezyklen steigt die Beschleunigung durch Einsatz von DLP von $5.8x$ auf $8.22x$. Beim 8x8 und gleicher Anzahl Wartezyklen steigt sie von $19.64x$ auf $24.91x$. Wie bereits in Abschnitt 4.6.5.2 erläutert, sind die Beschleunigungen auf dem SCC, wegen der vergleichsweise geringen Rechenleistung in Kombination mit der höheren Kernanzahl, generell höher.

Die Gegenüberstellung in Abb. 4.57 macht deutlich, dass eine höhere Beschleunigung nicht zwangsläufig einen besseren Absolutwert der Laufzeit zur Folge hat. Simuliert wurde ein 8x8 Modell bei 1×10^0 Wartezyklen. Sowohl im sequentiellen als auch im parallelen Fall sind Laufzeitverbesserungen durch DLP zu verzeichnen. Die geringe Taktfrequenz von 533 MHz der SCC Kerne hat aber

vergleichsweise lange Laufzeiten bei sequentieller Ausführung in der Größenordnung von 10^4 s zur Folge (im Vergleich zu nur 10^3 s auf dem Core i7 930). Durch aggressive Parallelisierung wird der Nachteil der geringen Rechenleistung der SCC Kerne allerdings nahezu ausgeglichen: Im betrachteten Szenario kann die Laufzeit auf dem SCC von 12468 s bzw. 11604 s (mit aktiver DLP) auf nur 675 s bzw. 509 s reduziert werden. Die absoluten parallelen Laufzeiten auf dem SCC bewegen sich dann zwischen den absoluten sequentiellen und parallelen Laufzeiten auf dem Core i7 930.



(a) Laufzeit auf dem Core i7 930

(b) Laufzeit auf dem SCC

Abbildung 4.57.: Vergleich der Laufzeiten

4.7. Einordnung in verwandte Arbeiten und Fazit

Die hier vorliegende Arbeit ist die erste Arbeit, in der die Parallelisierung von SystemC auf Manycore Architekturen untersucht wurde. Dem Autor sind darüber hinaus keine Arbeiten bekannt, dies das Problem der Parallelisierung von SystemC mit Hilfe von Schichtenbildung zerlegen. Auch Gegenüberstellungen unterschiedlicher Verfahren und Plattformen (z.B. asymmetrische versus symmetrische Strategie auf logischer Ebene, SCC versus cachkohärentem SHM Multiprozessor auf Ausführungsplattformebene) sind in der Literatur eher selten zu finden. Ausnahmen sind [260] und [61], wo eine GPU Parallelisierung von SystemC RTL Modellen mit einer Parallelisierung auf einem Multiprozessor bzw. FPGA verglichen wird.

Die asymmetrische synchrone Strategie aus Abschnitt 4.3 wurde in ähnlicher Weise in [105] und [232] implementiert, allerdings für konventionelle cachekohärente SHM Multiprozessoren. Im Bereich von SpecC existieren die Arbeiten

von Chen et. al [85, 83], die ebenfalls auf cachekohärente Architekturen spezialisiert sind. Bei den Untersuchungen in dieser Arbeit hat sich die asymmetrische Strategie als nur schwer anwendbar auf dem nicht cachekohärenten SCC erwiesen.

Vergleichbar mit dem symmetrischen asynchronen Ansatz aus Abschnitt 4.4 sind z.B. die Arbeiten in [271] und [82]. In beiden Fällen sind die Zielplattformen allerdings Workstation Cluster. In [271] werden nur synthetische Benchmarks durchgeführt und keine realistischen Simulationen. Das Verfahren in [82] wurde für SpecC TL Modelle implementiert. In [188], [206] und [181] werden Ansätze zur parallelen VHDL und Verilog Simulation beschrieben. Die Arbeiten erlauben aber entweder keine Relaxation der Synchronisation auf die Ebene von Timedcycles [188], fokussieren auf die Gatterebene [206] oder nutzen optimistische Synchronisation [188, 181].

Die größte Ähnlichkeit mit dem adaptiven Synchronisationsverfahren aus Abschnitt 4.5 haben die SystemC-basierten parallelen Simulatoren von Cox [93] und Combes et. al [90] sowie der parallele VHDL Simulator von Lungeanu und Shi [188]. Keine der genannten Arbeiten zielt jedoch auf Manycore Architekturen als Zielplattformen ab. Alle sind auf globale Synchronisation zwischen logischen Prozessen angewiesen. Zudem bieten Cox und Combes keine Werkzeugkette welche es erlaubt, SystemC Modelle automatisch zu analysieren, zu partitionieren und zu verteilen. Weder in [93], [90] noch in [188] wird eine statische Kausalitätsanalyse zur Vermeidung globaler Synchronisation genutzt.

Die einzigen bekannten Arbeiten im Bereich von SystemC, die eine statische compilergestützte Modellanalyse verwenden, sind die GPU basierten Lösungen aus [205], [237] und [260]. Nur die im Kontext von SpecC entstandenen Arbeiten von Chen et. al [85, 83] nutzen ebenfalls eine statische Compileranalyse, allerdings für eine zentralisierte Parallelisierung auf cachekohärenten Multiprozessoren. Keine der bekannten Arbeiten kombiniert eine statische mit einer dynamischen Analyse vor der Simulationslaufzeit, ähnlich dem Ansatz von PinaVM [193].

Die TL Modellierungsstrategie aus dem vorigen Abschnitt ist durch die Softwarearchitektur von Ptolemy II [217] inspiriert (vgl. Abschnitt 2.3.4), insbesondere die hierarchische Verschachtelung von Schedulern. Während die Hierarchisierung in PtII als Grundlage für die strukturierte Komposition von heterogenen Simulationsmodellen dient, wird sie in der beschriebenen TL Methodik allerdings als Basis für die Simulationsbeschleunigung eingesetzt. Ein möglicher Ansatz zur Nutzung von sog. Microschedulern im Kontext von SystemC/TLM mit dem Ziel der Simulationsbeschleunigung wird in [142] und [143] beschrieben. Jedoch wird in diesen Arbeiten keine Methode für die parallele Ausführung solcher Modelle hergeleitet.

In [259, 199] wird eine SystemC TL Modellierungsmethodik namens TLM/T bzw. deren Erweiterung TLM-DT vorgestellt, die auf Basis von Transaktionen ein PDES Protokoll umsetzt. TLM-DT Modelle sind mit einem speziellen Kernel parallel ausführbar. Der Kernel ist allerdings für die parallele Simulation von TLM-DT Modellen auf cachekohärenten SHM Multiprozessoren beschränkt. Weder in [259] noch in [199] werden leichtgewichtige Scheduler eingesetzt, um mikroskopische Effekte wie Kongestion einzelner Puffer in einem NoC Router akkurat zu simulieren. Zudem wird weder eine Lösung zur deterministischen TLM Simulation noch zur kontrollierten Beschränkung von temporärer Entkopplung beschrieben.

Insgesamt konnte in diesem Kapitel gezeigt werden, dass sich asynchrone Synchronisationsverfahren gut für die parallele Simulation von zyklenakkuraten MPSoC Simulationsmodellen eignen. Auf Architekturen wie dem SCC hat sich eine statische Partitionierung des Modells als notwendig erwiesen, um signifikante Beschleunigungen zu erreichen. Durch die Kombination aus Parallelisierung und Abstraktion ist ein weiterer Grad an Beschleunigung möglich. Neben der Partitionierung und dem Abstraktionsgrad hängt die Effizienz stark von der Größe des Simulationsmodells ab. Die Präparation eines Modells kann anhand entsprechender Werkzeugunterstützung erfolgen. Dies wurde anhand ein teil- und einer vollautomatisierten Werkzeugkette demonstriert.

Ziel zukünftiger Arbeiten kann es sein, die Anwendbarkeit der beschriebenen Methoden auf eines größeres Spektrum von Modellierungsstilen sowie eine Kombination aus heterogenen Ausführungsplattformen (GPUs, Intel MIC, FPGAs, ...) zu erweitern. Dazu sind die Eigenschaften von Modellen, Modellierungsstilen und Ausführungsplattformen zu analysieren, um daraus Schlussfolgerungen für notwendige Adaptionen der Laufzeitumgebung und der Modelle zu ziehen. In Kombination mit einer entsprechenden Erweiterung der teil- oder vollautomatisierten Werkzeugkette könnten SystemC Simulationsmodelle automatisiert auf heterogenen Ausführungsplattformen verteilt werden. Die beschriebene Erweiterung ist eng verknüpft mit der Entwicklung von neuen Strategien zur Simulationssynthese auf Basis von Modelltransformationen sowie der Entwicklung von Methoden zur Bewertung verschiedener Parallelisierungsansätze.

5. Interdisziplinäre verteilte Co-Simulation

Neben der Notwendigkeit von Methoden zur Nutzung von Multi- und Manycore Prozessoren für die parallele Simulation, wurde in Kapitel 1 die Unterstützung neuer Methoden zur kooperativen Simulation als elementarer Bestandteil zukünftiger interdisziplinärer Entwicklungsprozesse identifiziert. Einer der Hauptgründe dafür ist der erhöhte Grad an Interaktionen von zukünftigen Anwendungen und zugrundeliegenden Systemkomponenten, nicht nur mit der heterogenen physikalischen Umwelt, sondern auch mit einer weitreichenden und vernetzten IKT Infrastruktur.

In diesem Kapitel wird ein Ansatz zur Unterstützung eines interdisziplinären und simulationsbasierten Verifikationsprozesses für zukünftige eingebettete Systeme vorgestellt. Der Ansatz basiert auf einer Simulatorarchitektur und einer Methode zur Werkzeugkopplung. Am Beispiel automobiler elektrisch/elektronischer (E/E) Architekturen werden allgemeingültige Zusammenhänge zunächst konkretisiert[1]. Darauf aufbauend werden Anforderungen und Konzepte hergeleitet. Anschließend werden implementierte Konzepte beschrieben und die Anwendbarkeit anhand unterschiedlicher Prototypen demonstriert. Grundlagen zu diesem Kapitel wurden in zwei vom Autor betreuten Studienarbeiten [LZ10, Sch13] erarbeitet und sind somit in Zusammenarbeit entstanden. Konzepte und Ergebnisse sind in mehreren Publikationen veröffentlicht [RSHB10a, RSHB10b, RMR+12, BNR+13, RBB+14].

5.1. Beispiel: Automobile E/E Architekturen

Die E/E Architektur eines modernen Fahrzeugs ist heutzutage ein verteiltes Netzwerk, das aus mehreren Bussystemen, duzenden Steuergeräten (engl. *Electronic Control Units (ECUs)*) und hunderten von Sensoren und Aktuatoren besteht. An dieses System werden hohe Anforderungen bzgl. Zuverlässigkeit und Echtzeitfähigkeit gestellt. Die Erfüllung dieser Anforderungen ist die Grundlage für Si-

[1] Die Darstellung der Zusammenhänge basiert auf [71] und [51].

cherheit in verschiedensten Fahr- und Gefahrensituationen. Zur Erfüllung der Anforderungen wird das Zeitverhalten einzelner Teilsysteme sowie die Art und Weise der Kommunikation innerhalb des Fahrzeugs in der Entwurfsphase weitgehend statisch festgelegt. Dies ist möglich, da die E/E Architektur ein nahezu vollständig abgeschlossenes System bildet.

Verursacht durch die evolutionsartige Entwicklung und der dadurch entstandenen funktionalen Fragmentierung, übersteigt die Komplexität heutiger E/E Architekturen deutlich die Komplexität, die eigentlich für die existierende Funktionsdichte ausreichend wäre [51]. Traditionelle E/E Architekturen werden deswegen und aufgrund der stetig steigenden Funktionsdichte immer anfälliger für Fehler und so nach und nach zu einer „*Innovationsbarriere*“ [71]. Darüber hinaus sind Applikationen zunehmend von unterschiedlichsten Daten aus verschiedensten Quellen wie diversen Sensoren oder fahrzeuginternen und -externen Kommunikationsnetzwerken abhängig. Die genannten Defizite der E/E Architektur sind nicht zuletzt auf den stark Bottom-up geprägten Entwicklungsprozess zwischen Zulieferern (Tier 1 und Tier 2 Supplier) und OEMs zurückzuführen [51].

OEMs (z.B. Daimler, VW oder Toyota) produzieren das Gesamtfahrzeug. Tier 1 Zulieferer (z.B. Bosch oder Siemens) produzieren Teilsysteme wie ECUs oder Bussysteme und stellen diese dem OEM zur Verfügung. Tier 2 Zulieferer (z.B. Chiphersteller wie Freescale oder Infineon oder Hersteller von Betriebssystemen wie WindRiver) stellen den Tier 1 Zulieferern wiederum die Basistechnologien zur ECU Entwicklung zur Verfügung. Einzelne Teilsysteme werden weitgehend unabhängig voneinander entwickelt.

Das konkrete Vorgehen während Entwicklung und Test wird durch das sog. V-Modell [140] definiert. Dabei erfolgen Entwicklung und Test heutzutage weitgehend sequentiell. Im Ergebnis tauchen Fehler oft erst in der späten Phase der Integration von ECUs und E/E Architektur auf. Eine anschließende Ursachenforschung kann sehr zeitintensiv sein und erhebliche Kosten verursachen. Des Weiteren finden Optimierungen von Steuergerätefunktionen meist ausschließlich bei einem Zulieferer und losgelöst vom OEM oder anderen Zulieferern statt. Dadurch werden mögliche (implizite) Wechselwirkungen mit anderen Funktionen und Teilsystemen oft nicht berücksichtigt.

5.1.1. Einfluss neuer Technologien auf die E/E Architektur

Die beschriebene Situation im Bereich der E/E Architekturen wird durch die Integration neuartiger Technologien wie beispielsweise *Drive-by-Wire (DbW)* Systemen [146] oder Fahrzeug-zu-Fahrzeug bzw. Fahrzeug-zu-X Kommunikation (engl. *Vehicle-to-X Communication (V2XC)*) [30][227] weiter verschärft. Diese zeich-

nen sich durch eine gesteigerte Abhängigkeit von mehreren oder vielen verteilten Informationsquellen und -senken aus.

DbW Systeme basieren auf voneinander separierten hochintegrierten mechatronischen Antriebsmodulen, die klassische Radnabenmotoren ersetzen [123]. Diese Antriebsmodule sind intelligente Module, die Sensordaten bereits intern vorverarbeiten, um eine Regelung zunächst lokal zu optimieren. Alle im Fahrzeug vorhandenen Antriebsmodule (in der Regel eines pro Rad) werden dann schließlich durch eine fahrzeugübergreifende Regelung, die beispielsweise auf einer zentralen leistungsfähigen Rechnerplattform ausgeführt wird, aufeinander abgestimmt.

V2XC bildet die Basis für zukünftige Anwendungen und Dienste, für die eine ausschließliche Berücksichtigung von im Sichtbereich des Fahrzeugs verfügbarer Information nicht mehr ausreichend ist. Der Großteil der Anwendungen wird vielmehr auf der Verfügbarkeit von Informationen aus räumlich weit verteilten Quellen basieren, um beispielsweise eine automatisierte globale Verkehrsflussregelung vornehmen zu können. In erster Näherung muss die E/E Architektur dazu mit einer neuen Funkschnittstelle „geöffnet“ werden (z.B. über eine dafür vorgesehene ECU). Diese erlaubt es dann, mit anderen Fahrzeugen oder der externen IKT Infrastruktur in Kontakt zu treten [227]. Die Öffnung des internen Fahrzeugnetzwerks nach außen erzeugt neue zusätzliche Abhängigkeiten und Wechselwirkungen und erhöht nicht zuletzt die Gefahr von Störungen sicherheitskritischer Funktionen oder Angriffen auf das interne Fahrzeugnetz [121].

Um E/E Architekturen für solch neue Technologien tauglich zu machen und gleichzeitig existierende Limitierungen hinsichtlich Erweiterbarkeit für zukünftige Innovationen aufzuheben, ist eine grundlegende Überarbeitung existierender Konzepte notwendig [51]. Dies beinhaltet eine stringentere Anwendung von Entwurfsparadigmen wie Modularität, Kapselung, Standardisierung und Zentralisierung. In [51] werden diese Paradigmen unter dem Oberbegriff der Virtualisierung zusammengefasst. Ein vergleichbarer Prozess wurde schon einmal in den 1980er Jahren durchlaufen, als die direkte Verkabelung zwischen ECUs durch Bussysteme ersetzt wurde [51] (siehe Abb. 5.1).

5.1.2. Auswirkungen auf den Entwicklungsprozess

Anhand den beiden beschriebenen Technologien DbW und V2XC lassen sich wichtige Schlussfolgerungen an zukünftige Entwicklungsprozesse und an darin zu verwendende Werkzeuge ableiten. Wechselwirkungen zwischen verschiedenen Teilsystemen sollten so früh wie möglich bei den Entwicklungsaktivitäten Beachtung finden. Dazu müssen existierende Bottom-up geprägte Entwicklungsprozesse durch neue Meet-in-the-Middle Ansätze ergänzt werden. Dies eröffnet dann die Möglichkeit für eine durchgängige modellbasierte Herangehens-

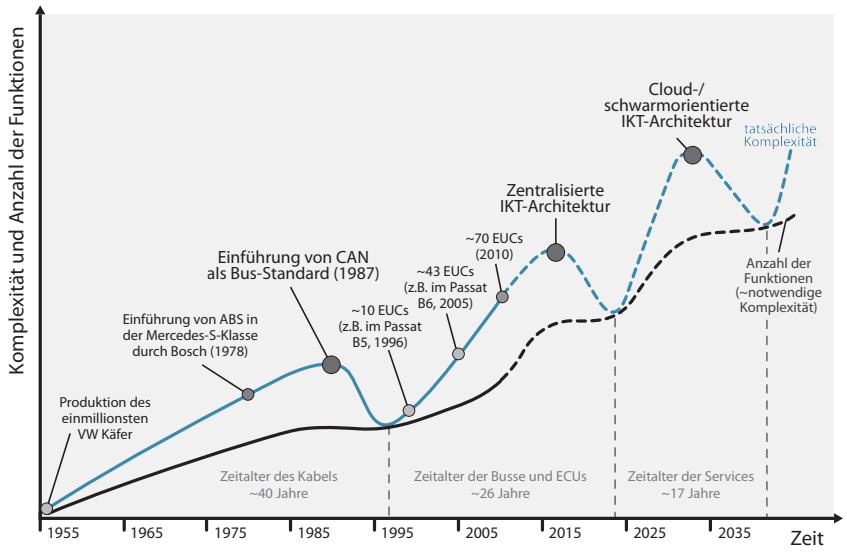


Abbildung 5.1.: Evolution der Komplexität von E/E Architekturen (Quelle: [51])

weise und die Verifikation von Teilsystemen und deren Interaktionen, ohne alle Details der finalen Implementierung kennen zu müssen.

Die plattformbasierte Entwurfsmethodik (vgl. Abschnitt 2.1.1.3) ist ein vielversprechender Lösungsansatz, um die erwähnten Defizite innerhalb automobiler Entwicklungsprozesse zu eliminieren [229]. Die Etablierung von Prinzipien des PBD im automobilen Kontext bedeutet zunächst, Anwendungen vollständig als Software zu *„virtualisieren“*[2] und von ihrer ursprünglich festen Verknüpfung mit einer bestimmten ECU zu lösen. Funktion und Architektur sind damit von Beginn an voneinander separiert. Dies gestattet eine flexible Abbildung von Funktionen auf die E/E Architektur. Zudem können abstrakte modellbasierte Funktionsspezifikationen und deren Zusammenspiel bereits in frühen Entwicklungsstadien z.B. durch Simulation modellbasiert verifiziert werden, ohne die exakte Implementierung der Architektur zu kennen. Dies ist die Grundlage für eine iterative Verfeinerung des Systems und dessen durchgängige Verifikation und resultiert in erhöhter Zuverlässigkeit und reduzierten Kosten [229].

Zusätzlich zu den beschriebenen PBD Prinzipien sind insbesondere die technologische Heterogenität neuartiger Systeme wie DbW oder V2XC und die durch die starke Vernetzung zusätzlich entstehenden Wechselwirkungen bereits in frühen Phasen des Entwurfsprozesses zu berücksichtigen. Diese Notwendigkeit wird

[2] Für den Begriff der Virtualisierung wird an dieser Stelle die Definition aus [51] vorausgesetzt.

dadurch verstärkt, dass es immer schwieriger wird, klare Systemgrenzen zu definieren. Dadurch können bestimmte Wechselwirkungen (z.B. mit dem V2X Netzwerk) nicht einfach ausgeschlossen werden.

Aus technologischer Sicht entstehen diese Wechselwirkungen zwischen verschiedenen Anwendungsdisziplinen. Da sich innerhalb einzelner Disziplinen wie Informationstechnik, Kommunikationstechnik oder Mechatronik bereits spezialisierte Modellierungs- und Simulationswerkzeuge etabliert haben, ist die Möglichkeit zur Wiederverwendung dieser zueinander heterogenen Werkzeuge ein grundlegendes Kriterium für den Erfolg einer Entwicklungsmethodik [20].

Darüber hinaus entstehen aus organisatorischer Sicht Wechselwirkungen zwischen Zulieferern und OEM: Bestimmte Simulationswerkzeuge oder Modelle sind z.B. aus lizenzrechtlichen Gründen nur bei einem Zulieferer verfügbar. Eine verteilte Kopplung kann eine Lösung sein, um trotzdem Untersuchungen von Wechselwirkungen zwischen diesen Modellen zu ermöglichen.

Aus den genannten Gründen ist es notwendig, neue PBD Ansätze zu entwickeln, welche eine heterogene Modellierung gestatteten und existierende Simulationswerkzeuge koppeln können. Komplikationen, die durch Inkompatibilitäten zwischen heterogenen Simulationswerkzeugen entstehen können, sollten soweit wie möglich reduziert werden. Aus organisatorischen Gründen ist eine verteilte Koppelbarkeit vorteilhaft.

5.2. Anforderungen an die entwickelnde Simulationsumgebung

Aus dem beschriebenen Szenario der automobilen E/E Architekturen und den daraus abgeleiteten Auswirkungen auf den Entwicklungsprozess ergeben sich folgende konkrete Anforderungen an die zu entwickelnde Simulationsumgebung:

I) **Interdisziplinäre Modellierung und Simulation**: Es sollte eine Modellierung und Simulation unter gleichzeitiger Verwendung disziplinspezifischer Simulationswerkzeuge möglich sein.

II) **Verbesserte Interoperabilität**: Aus der vorherigen Anforderung ergibt sich unmittelbar die Notwendigkeit für Interoperabilität. Die Voraussetzung dafür sind eindeutig definierte syntaktische und semantische Schnittstellen zur Co-Simulation.

III) **Heterogene Modellierung**: Die Simulationsumgebung sollte eine formale Grundlage besitzen, die es erlaubt, eine Kombination aus heterogenen Teilsystemen und deren Interaktionen so genau wie möglich zu spezifizieren.

IV) **Wiederverwendbarkeit und Erweiterbarkeit**: Die Simulationsumgebung sollte flexibel und strukturiert erweiterbar sein. Durch die Integration disziplinspezifischer Modelle sollte die Spezifikation des Restsystems unangetastet bleiben.

V) **Handhabbarkeit**: Es soll die Möglichkeit bestehen, Schnittstellen zu erzeugen und dadurch Interoperabilität bis zu einem gewissen Grad automatisch herzustellen. Eine solche Automatisierung beschleunigt den Prozess der Integration. Die Möglichkeit zur verteilten Ausführung fördert die Handhabkeit zusätzlich.

VI) **Platformbasierter Entwurfsprozess**: Die Simulationsumgebung sollte für die Anwendung einer Meet-in-the-Middle Methodik wie der PBD Methodik (vgl. Abschnitt 2.1.1.3) geeignet sein. Dies beinhaltet die Möglichkeit zur Spezifikation von Funktion-zu-Architektur Abbildungen sowie deren durchgängige Verifikation innerhalb eines iterativen Verfeinerungsprozesses, unter Berücksichtigung existierender Wechselwirkungen mit anderen Teilsystemen.

5.3. Konzept

Im Allgemeinen sind die Aufgaben einer Schnittstelle zur Co-Simulation vergleichbar mit den Aufgaben der logischen Ebene des Referenzmodells zur parallelen Simulation (vgl. z.B. Abschnitt 4.4). Der prinzipielle Unterschied liegt darin, dass bei einer Co-Simulation unterschiedliche Simulationswerkzeuge genutzt werden, die auf unterschiedlichen Sprachen basieren und unterschiedliche heterogene Berechnungsmodelle implementieren (vgl. Abschnitt 2.2.2). Eine Co-Simulation ist auch dann heterogen, wenn verschiedene Simulatoren einer Klasse (z.B. DE) gekoppelt werden, da sich deren Funktion meist im Detail unterscheidet. Die Heterogenität wird spätestens dann zum Problem, wenn Berechnungsmodelle von kommerziellen Werkzeugen nicht offengelegt werden und somit nicht eindeutig definiert sind. Im schlimmsten Fall liegtim Fall einer Kopplung ein unerwartetes nicht nachvollziehbares Verhalten vor.

In Abschnitt 3.2 wurde die beschriebene Problematik auf die unzureichende semantische Interoperabilität zwischen heterogenen Simulationswerkzeugen zurückgeführt. Semantische Interoperabilität wird in den aktuell verfügbaren Standards zur Co-Simulation wie HLA [22] oder FMI [29] nicht im notwendigen Umfang adressiert. Deren Nutzung ist entweder auf bestimmte Klassen von Berechnungsmodellen limitiert, oder sie erlauben keine hinreichend genaue (d.h. explizite) Spezifikation der erlaubten statischen und dynamischen Semantik des Datenaustauschs (vgl. Abschnitt 3.2.2).

Das Konzept, mit dessen Hilfe der beschriebenen Problematik unter Berücksichtigung der Anforderungen aus Abschnitt 5.2 begegnet werden soll, besteht aus einer Simulatorarchitektur und einer Methode zur Werkzeugkopplung. Für die Umsetzung der Methode wird in einem späteren Abschnitt eine auf der Simulatorarchitektur aufsetzende Werkzeugkette vorgeschlagen und prototypisch implementiert. Simulatorarchitektur und Methode werden im Folgenden erläutert.

5.3.1. Simulatorarchitektur

Das Konzept der Simulatorarchitektur ist in Abb. 5.2 dargestellt. Der gewählte Ansatz basiert auf einem heterogenen Backbone zur Co-Simulation. Heterogene Simulationswerkzeuge werden über diesen Backbone gekoppelt. Der Backbone selbst besteht aus drei Komponenten:

1. Einer **Simulationsmiddleware** wie der HLA [22] (vgl. Abschnitt 3.2.3.2), die es erlaubt, eines oder mehrere Simulationswerkzeuge als Federates innerhalb einer verteilten Federation auszuführen.
2. Einem **SDEM Metamodell** namens *Simulation Data Exchange Metamodel (SDEMM)*[3]: Das SDEMM dient zur flexiblen Spezifikation der Semantik des Datenaustauschs zwischen Simulationswerkzeugen und HLA. Eine Instanz des SDEMM entspricht dem im DSEEP des HLA Standards [26] vage formulierten Konzepts des SDEM (vgl. Abschnitt 3.2.3.2).
3. Einem **heterogenen M&S Werkzeug**, welches eine mathematisch formale Basis besitzt (vgl. Abschnitt 2.3.3 ff.). Das in dieser Arbeit verwendete Werkzeug ist Ptolemy II [102].

5.3.1.1. Hierarchische Kopplung von PtII und HLA

PtII und die HLA unterscheiden sich in einer Hinsicht grundlegend voneinander: Innerhalb einer HLA Federation können u.U. unterschiedliche Kommunikationsmechanismen zur gleichen Zeit verwendet werden. HLA Federations sind bzgl. der dynamischen Semantik nicht strukturiert. In PtII Modellen ist die Bildung von Komponentenhierarchien hingegen fundamental für die Spezifikation von Modellen, die bzgl. des Berechnungsmodells heterogen sind.

Ein Ansatz, diese beiden gegensätzlichen Konzepte dennoch miteinander zu kombinieren, ist die Integration von HLA Federations in jeweils abgeschlossene ho-

[3] Der Begriff des **Metamodells** wird hier entsprechend [154, 245] verwendet. Ein Metamodell ist somit äquivalent zur Definition einer Modellierungssprache und kann anhand von UML Klassendiagrammen dargestellt werden. Dies steht im Gegensatz zur Definition eines Metamodells im MDA [9] Kontext, wo ein UML Klassendiagramm grundsätzlich nur als **Modell** und nicht als Metamodell bezeichnet wird.

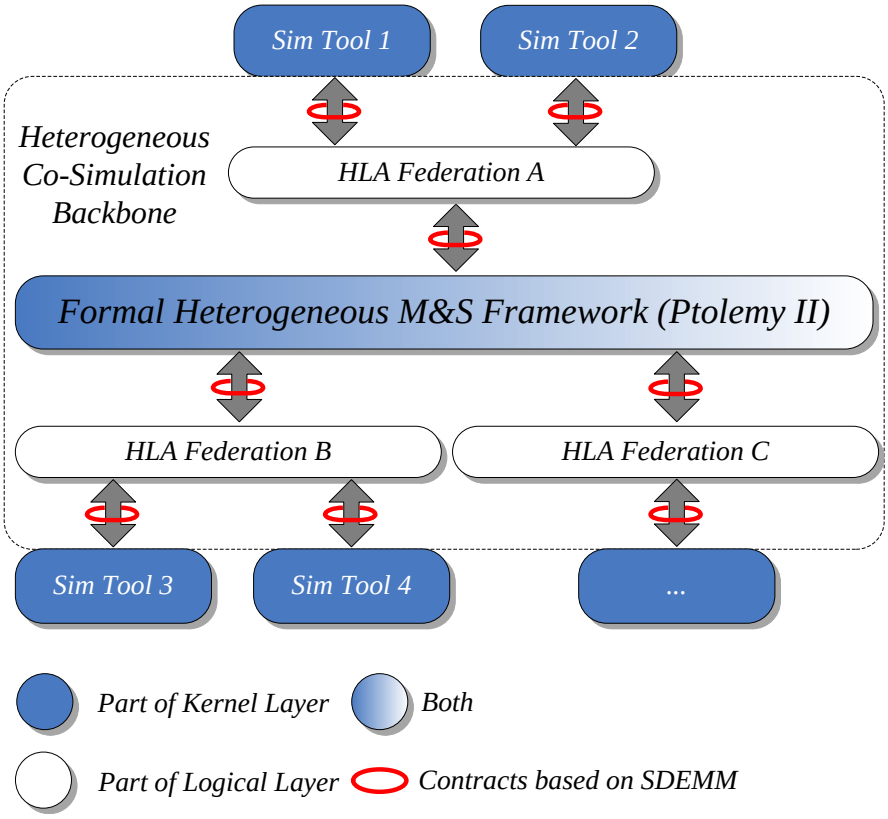


Abbildung 5.2.: Architekturkonzept zur interdisziplinären Co-Simulation

mogene PtII Domänen. Dies kann beispielsweise durch Ableitung von einem bereits existierenden Director erfolgen. Die HLA Erweiterung des Directors ist dann nur in genau dem kompositen Actor sichtbar, in dem der neue Director auch verwendet wird. Die Heterogenität von PtII Modellen bleibt erhalten. Auf diese Weise kann die Teilanforderung der Wiederverwendbarkeit, durch Modifikation eines Teilmodells die Spezifikation des Restsystems nicht anzutasten, auch im Fall einer Co-Simulation erfüllt werden (vgl. Abschnitt 5.2).

5.3.1.2. Aufgaben der Komponenten innerhalb der Architektur

Die HLA ist die technologische Grundlage für die Einbindung verschiedener disziplinspezifischer Simulationswerkzeuge und -modelle in eine verteilte Gesamtsimulation. Sie stellt flexible Schnittstellen für eine auf mehrere Ausführungsplattformen verteilte Simulation zur Verfügung.

Das SDEMM ist die Basis für Spezifikation der statischen und dynamischen Semantik des Datenaustauschs für jeden beteiligten Simulator anhand eines SDEM. Das SDEMM besteht dazu aus einem sog. *Simulation Object Metamodel (SOMM)* und einem *Behavioral Interface Metamodel (BIMM)*. SOMM und BIMM definieren die erlaubte Syntax eines SOM und eines sog. *Behavioral Interface Model (BIM)*. Das SDEMM bildet zugleich die Schnittstelle zu einer C++ Bibliothek namens SDEMMlib. SDEMMlib implementiert eine *semantische Abbildung* [128] von Komponenten des SDEMM auf die semantische Domäne der HLA.

Auf Basis der durch die SDEMMlib Bibliothek festgelegten semantischen Abbildung von SDEMM Komponenten auf die HLA Domäne definiert ein konkretes SDEM wiederum eine semantische Abbildung der HLA in eine bestimmte Anwendungsdomäne. Diese entspricht der Festlegung eines *Vertrags* (engl. *Contract*) zwischen Simulationswerkzeugen und HLA bzgl. der erlaubten *statischen* und *dynamischen* Semantik des Datenaustauschs. Ein SDEM stellt sicher, dass *zwischen* Simulator und HLA nur Interaktionen erfolgen, die laut Vertrag auch gestattet sind. Mehrere SDEMs in Kombination stellen sicher, dass *über* die HLA nur erlaubte Interaktionsmuster erfolgen. Sie limitieren alle möglichen Interaktionsmuster auf eine gültige Teilmenge. SDEMs vereinfachen so die verteilte Integration verschiedener Werkzeuge z.B. zwischen verschiedenen Entwicklungsteams in verschiedenen Firmen (Zulieferer und OEM). Ausführbare SDEMs können während der Implementierung der verteilten Simulation bei der Fehlersuche nützlich sein. Die dazu umgesetzte Bibliothek SDEMMlib (siehe Abschnitt 5.4) kann zusammen mit PtII dazu beitragen, unerwartetes Verhalten zu vermeiden.

PtII hat insgesamt drei Aufgaben innerhalb der Architektur:

1. PtII dient als Anwenderschnittstelle zur Unterstützung der Werkzeugkopplung vor der Simulationslaufzeit. Dies beinhaltet die Konfiguration von

statischer und dynamischer Semantik zwischen einzubindenden Simulationswerkzeugen und HLA in visueller PtII Syntax (vgl. Abb. 2.15) und die anschließende Generierung von HLA Schnittstellen.

2. PtII dient als Koordinator während der heterogenen Co-Simulation. Dies beinhaltet die Steuerung des Datenaustauschs zwischen gekoppelten heterogenen Simulationswerkzeugen während der Simulationslaufzeit. Diese Aufgabe ist Vergleichbar mit der Rolle eines Schedulers in einem gewöhnlichen Simulator.
3. Aufgrund seiner Fähigkeiten zur strukturierten Komposition von heterogenen Modellen dient PtII als zentrales Entwurfswerkzeug für die Umsetzung einer bestimmten Entwurfsmethodik für eingebettete Systeme (vgl. Abschnitt 2.1). Durch die Möglicheit zur Simulation auf abstrakter Ebene kann PtII insbesondere als Ausgangspunkt für eine Meet-in-the-Middle Entwurfsmethodik auf Systemebene genutzt werden.

PtII vereint damit Aufgaben einer Schnittstelle zur Co-Simulation mit Aufgaben eines reinen Simulators. Bzgl. des Aufgabenspektrums kann PtII also nicht eindeutig der logischen Ebene oder der Kernelebene im Referenzmodell für die parallele SystemC Simulation aus Abschnitt 4.1 zugeordnet werden. Dies soll anhand der Farbgebung in Abb. 5.2 deutlich werden.

5.3.2. Methode zur Etablierung einer Simulatorkopplung

Die Vorgehensweise für die Herstellung einer Simulatorkopplung mit Hilfe der beschriebenen Simulatorarchitektur ergibt sich prinzipiell aus der Entwurfsmethodik für eingebettete Systeme, innerhalb derer die Simulatorarchitektur verwendet wird. Im Folgenden wird beispielhaft die Kombination mit einer interdisziplinären Meet-in-the-Middle Entwurfsmethodik skizziert.

5.3.2.1. Interdisziplinäre Entwurfsmethodik für eingebettete Systeme

Eine typische Meet-in-the-Middle Entwurfsmethodik beginnt in PtII mit der Verhaltensspezifikation auf abstrakter Ebene unter Verwendung geeigneter Berechnungsmodelle. Da sich die Systemgrenzen für zukünftige eingebettete Systeme nicht mehr ohne weiteres definieren lassen, müssen diese zu Beginn in geeigneter Weise festgelegt werden. Eine gute Wahl ist die durch die physikalische Ausdehnung des zu entwickelnden Teilsystems definierte Grenze. Alles außerhalb dieser Grenze wird als Systemumwelt betrachtet. Durch die Berücksichtigung der Systemumwelt ist es möglich, Wechselwirkungen des Zielsystems mit der Umgebung zu spezifizieren und zu untersuchen. Die verschiedenen in PtII

verfügbaren Berechnungsmodelle erlauben dabei von Beginn an eine heterogene Modellierung von Teilen des Gesamtsystems in einem geeigneten Formalismus.

In PtII kann das zu entwickelnde Zielsystem z.B. durch einen *Target* Actor und dessen Umgebung durch mehrere *Environment* Actors modelliert werden. Die *Environment* Actors enthalten Modelle der Systemumwelt, welche als Testbenches für *Target* dienen. Im Verlauf der Entwicklung wird *Target* sukzessive dekomponiert und verfeinert. Dies bedeutet in PtII, dass der Actor *Target* durch einen kompositen Actor (gleichen Namens) ersetzt wird. Dieser enthält selbst wiederum beliebige Hierarchien von Actors. Mit Hilfe der Testbenches in den *Environment* Actors können unterschiedliche Versionen von *Target* unter Berücksichtigung von Wechselwirkungen mit der Systemumwelt durchgängig auf Korrektheit verifiziert werden.

Ab einer gewissen Verfeinerungsstufe genügt eine Spezifikation von Teilsystemen im kompositen Actor *Target* durch einen oder mehrere reine PtII Actors nicht mehr. Um Details der Implementierung berücksichtigen zu können, müssen Actors durch bereits existierende (z.B. in einem Bottom-up Ansatz bereits entwickelte) detaillierte Teilmodelle verfeinert werden. Diese liegen typischerweise in disziplinspezifischen Sprachen vor.

5.3.2.2. Entwicklung von HLA Federations

Unabhängig davon, ob es sich bei einem Actor um ein Modell der Systemumwelt oder ein Modell des zu entwickelnden Systems handelt, kann folgende sechsstufige Vorgehensweise zur Etablierung einer Co-Simulation zwischen PtII und externen Simulationswerkzeugen via HLA verwendet werden (siehe Abb. 5.3):

1. **Identifikation von Federates**: In einem ersten Schritt werden sog Federate Actors $F_1 ... F_n$ identifiziert, welche durch Co-Simulation verfeinert werden sollen. Außerdem werden die externen Simulationswerkzeuge $S_1 ... S_n$ ausgewählt, die zur Verfeinerung genutzt werden sollen. Dabei muss für jeden Actor F_i genau ein Werkzeug S_i existieren.
2. **Klassifikation von Berechnungsmodellen**: Im zweiten Schritt werden die Berechnungsmodelle der Simulatoren klassifiziert. Dies beinhaltet
   a) die Identifikation von Regeln, welche die Komponenten, Kommunikations- und Ausführungsmechanismen eines Berechnungsmodells beschreiben (vgl. Abschnitt 2.2.3).
   b) die Klassifikation des Berechnungsmodells durch Vergleich der genannten Regeln mit den Regeln all der Berechnungsmodelle, die in PtII als Domänen implementiert sind. Ein mathematischer Ansatz zur Klassifikation wird in [175] beschrieben. Ein Simulator S_i wird durch

genau die Domäne von PtII am besten charakterisiert, deren Berechnungsmodell dem von S_i am nächsten kommt.

3. **Identifikation von Federations**: Die Actors F_1 ... F_n werden in Federationmodelle $FM_1 ... FM_m$ gruppiert. Dabei ist jeder Actor F_i Teil genau eines Federationmodells FM_k. Im einfachsten Fall ist $n = m$, d.h. jede Federation enthält genau einen externen Simulator. Die Gruppierung erfolgt durch Erzeugung von kompositen Actors, welche eine Teilmenge $\{F_i, ..., F_j\}$ der Actors kapseln. Als ein notwendiges Kriterium, um Actors F_i und F_j in eine Federation FM_k zu gruppieren, kann die „Kompatibilität" der Berechnungsmodelle von F_i und F_j herangezogen werden, d.h. beide müssen durch die gleiche PtII Domäne charakterisiert sein. Damit kann eine Federation einer PtII Domäne zugeordnet werden. Durch die hierarchische Struktur von PtII ist die Gruppierung zusätzlich auf Actors limitiert, die sich auf der gleichen Hierarchieebene befinden. Insgesamt kann zwischen zwei möglichen Modi zur kooperativen Simulation unterschieden werden:

   a) **Single-Federation Modus**: Die zu koppelnden Simulatoren sind Teil ein und derselben Federation. Dieser Modus ist insbesondere dann sinnvoll, wenn alle Simulatoren dem gleichen oder einem ähnlichen Berechnungsmodell folgen. Beispielsweise ist eine Kopplung mehrerer identischer DE Simulatoren relativ unkompliziert, da der HLA Standard bereits einen diskreten ereignisbasierten Time Management Service spezifiziert (vgl. Anhang B).

   b) **Multi-Federation Modus**: In diesem Modus werden mehrere Single-Federations über PtII gekoppelt. Einzelne oder mehrere Simulatoren werden dann mit PtII innerhalb separater kompositer Actors co-simuliert. Die kompositen Actors folgen u.U. verschiedenen Berechnungsmodellen. PtII dient als Gateway zwischen (nahezu) homogenen Federations und koordiniert deren Ausführung. Dieser Modus eignet sich insbesondere zur Unterstützung der RTI bei der Koordination der Heterogenität. Zudem hat eine hierarchische Komposition von Federations eine strukturiertere Implementierung zur Folge.

4. **Konfiguration des Datenaustauschs durch semantische Abbildung**: Um einen kontrollierten Datenaustausch zwischen Simulatoren über die HLA zu ermöglichen, muss die erlaubte Semantik definiert werden. Dieser Vorgang kann entsprechend [128] allgemein als semantische Abbildung bezeichnet werden. In dieser Arbeit beinhaltet die semantische Abbildung die Spezifikation der statischen und der dynamischen Semantik des Datenaustauschs:

   a) **Statische Semantik des Datenaustauschs**: Die Spezifikation der statischen Semantik des Datenaustauschs ergibt sich aus den auszutauschenden Daten in einer HLA Federation. Die explizite Festlegung

geschieht durch Ersetzen von PtII Relationen und Links durch Actors, welche konkrete HLA Object Classes und Interaction Classes, deren Attribute, Parameter und Datentypen repräsentieren. Als Ergebnis erhält man eine Spezifikation der statischen Semantik des Datenaustauschs zwischen Simulatoren, die durch Actors eines Federationmodells *FM* repräsentiert sind. Aus dem *FM* können unmittelbar SOMs und FOM abgeleitet werden, indem die visuelle PtII Syntax auf SOM/FOM Syntax abgebildet wird.

b) **Dynamische Semantik des Datenaustauschs**: Die Spezifikation der dynamischen Semantik des Datenaustauschs ergibt sich aus den erlaubten Interaktionen in einer HLA Federation. Sie beinhaltet eine explizite Beschreibung des erlaubten Verhaltens, die sich von der Initialisierung, über die eigentliche Ausführung der Simulation bis zu deren Terminierung erstreckt. Die dynamische Semantik des Datenaustauschs während der Ausführung resultiert aus den Berechnungsmodellen der Simulatoren, zwischen denen der Datenaustausch stattfinden soll. Wenn alle Simulatoren einer Federation durch die gleiche PtII Domäne charakterisiert sind, reduziert sich der Aufwand für die Anpassung. Die Spezifikation kann durch Annotation von Parametern an die Federate und Object Class Actors erfolgen (insbesondere durch Annotation von BIMs an Federate Actors[4]).

5. **Generierung von Schnittstellen**: Aus der vollständigen Konfiguration eines Federationmodells *FM* werden geeignete HLA Schnittstellen (Interface Wrapper (IFW)) für die einzelnen Simulationswerkzeuge der Federation abgeleitet und automatisch generiert. In den nächsten Abschnitten wird dazu ein Bibliotheksansatz vorgestellt, der im nachfolgenden Integrationsschritt zum Testen und Debuggen von Federations sowie zum Aufdecken von Integrationsfehlern hilfreich ist.
6. **Integration und Test**: Im letzten Schritt werden die generierten Schnittstellen in die Simulationswerkzeuge integriert, so dass eine kooperative Ausführung möglich ist. Abhängig von Werkzeug, Modell und Automatisierungsgrad, kann/muss dieser Schritt automatisch/manuell erfolgen. In PtII beinhaltet dies die Erweiterung von Director und Actor Klassen in geeigneter Weise.

Da ein Federationmodell ein PtII Modell ist, kann es in dem PtII spezifischen XML Dialekt MoML [177] abgespeichert werden. Es ist somit in zukünftigen Co-Simulationen auch innerhalb anderer PtII Modelle direkt wiederverwendbar. Die Schritte drei bis sechs können dann entfallen.

[4]Für das durch PtII selbst gegebene Federate kann die Spezifikation durch Annotation an den das Federationmodell umgebenden kompositen Actor erfolgen.

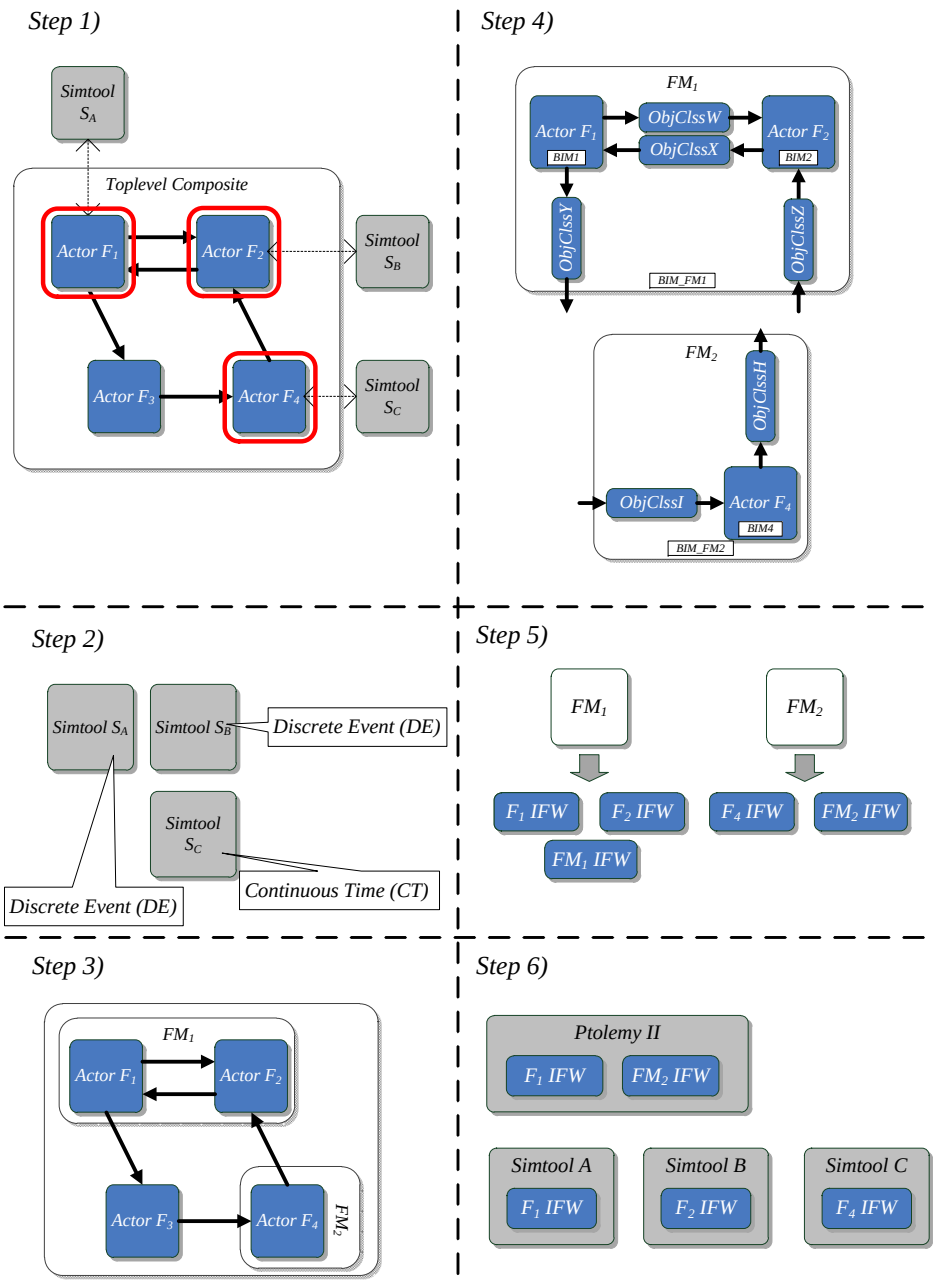


Abbildung 5.3.: Vorgehensweise zur Werkzeugkopplung

Insgesamt existieren mit dem beschriebenen Ansatz zwei miteinander kombinierte Mechanismen zu Verbesserung semantischer Interoperabilität: Innerhalb einer Federation dienen die BIMs dazu, die erlaubte dynamische Semantik des Datenaustauschs so gut wie möglich an die dynamische Semantik der Simulatoren in der Federation anzupassen. Über Domänen hinweg wird die Heterogenität von Berechnungsmodellen vollständig von PtII koordiniert.

An dieser Stelle soll noch einmal explizit angemerkt werden, dass es nicht der Anspruch der vorgeschlagenen Methode ist, die Problematik der Interoperabilität vollständig zu lösen. Die oben geschilderten Probleme, die beim Zusammenschalten heterogener und verteilter Simulationswerkzeuge auftreten können, sind nicht immer vermeidbar. Dies ist aus den genannten Gründen auch gar nicht möglich. Die Methodik soll allerdings eine Hilfestellung bieten, um mögliche Komplikationen auf ein Minimum zu reduzieren.

5.4. Implementierung der Simulatorarchitektur

5.4.1. CERTI HLA

Da die HLA ein Standard ist und keine Implementierung, muss für die Umsetzung der Simulatorarchitektur zunächst eine geeignete RTI Implementierung ausgewählt werden. Im Rahmen einer Recherche wurden diverse existierende freie und kommerzielle Implementierungen einer HLA RTI wie CERTI [209], poRTIco [12], MÄC [16] oder pRTI [11] ausfindig gemacht. Im Zuge der Untersuchungen hat sich CERTI als die am meisten ausgereifte freie Implementierung erwiesen und wurde deswegen für die prototypische Implementierung verwendet. Grundsätzlich ist die RTI Implementierung aber austauschbar.

Die Softwarearchitektur von CERTI [209] ist in Abb. 5.4 dargestellt. Die RTI ist ein verteiltes System bestehend aus zwei Typen von Prozessen, einem lokalen namens *RTI Ambassador (RTIA)* und einem globalen namens *RTI Gateway (RTIG)*. Jeder Federate Prozess interagiert lokal mit einem RTIA Prozess über ein Unix Domain Socket. Er muss dazu die Ambassador Schnittstellen implementieren. Diese werden durch eine Bibliothek namens libRTI zur Verfügung gestellt. Ein RTIA Prozess selbst ist wiederum über TCP und UDP Sockets mit dem RTIG Prozess verbunden. Sämtliche Kommunikation und Synchronisation erfolgt über das RTIG. Eine einzige RTIG Instanz kann mehrere voneinander unabhängige logische Federations gleichzeitig steuern.

CERTI stellt sowohl die HLA Schnittstellen nach dem IEEE 1516 Standard [22] als auch nach dem Originalen DoD HLA 1.3 Standard [257] zur Verfügung. Bis auf einige kleine Unterschiede sind die Standards weitgehend deckungsgleich.

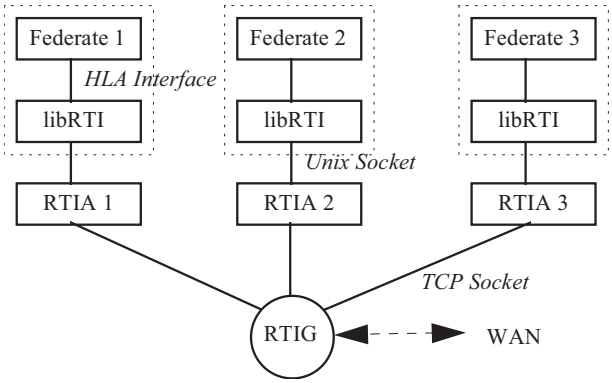


Abbildung 5.4.: Softwarearchitektur von CERTI (Quelle: [209])

Die im Folgenden beschriebene Implementierung basiert auf dem DoD HLA 1.3 Standard. Aufgrund der Modularität der SDEMMlib ist eine einfache Portierung auf den IEEE 1516 Standard möglich.

5.4.2. Simulation Data Exchange Metamodel

Das SDEMM ist die Schnittstelle der SDEMMlib C++ Bibliothek. SDEMMlib ist ein Baukasten, der eine modellbasierte Beschreibung der Semantik des Datenaustauschs zwischen einem Simulationswerkzeug und der HLA erlaubt. Das SDEMM ist dazu in ein SOMM und ein BIMM aufgeteilt, mit deren Hilfe ein SOM und ein BIM erstellt werden können.

Eine konkrete modellbasierte Beschreibung in Form eines SOM und eines BIM in Kombination mit der von SDEMMlib bereitgestellten Klassenbibliothek bildet die Grundlage für eine semi-automatische Erzeugung einer funktionsfähigen HLA Schnittstelle für einen bestimmten Simulator. Die Komponenten der Klassenbibliothek abstrahieren dazu von den standardmäßigen HLA Schnittstellen und lassen sich strukturiert miteinander kombinieren. Abb. 5.5 zeigt das Toplevel Klassendiagramm von SDEMMLib.

Die *Federate* Klasse aggregiert alle anderen Toplevelklassen. Das SOMM wird durch die *SimulationObjectModel* Klasse repräsentiert, das BIMM durch die *BehavioralInterfaceModel* Klasse.

Die *ToolAdaptor* Klasse stellt eine Verbindung von einem Simulationswerkzeug zu einem konkreten SOM (Instanz der *SimulationObjectModel* Klasse) und zu einem BIM (Instanz der *BehavioralInterfaceModel* Klasse) her. Die *ToolAdaptor* Klasse stellt kontrollflussrelevante Methoden zur Verfügung, die vom angebundenen

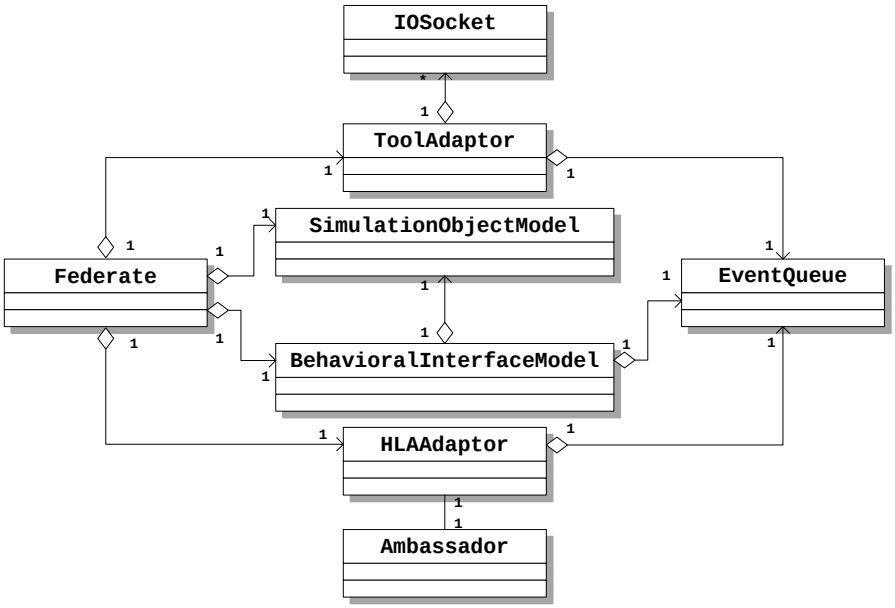


Abbildung 5.5.: Toplevel Klassendiagramm von SDEMMlib

Simulationswerkzeug aufgerufen werden. Daneben kapselt der *ToolAdaptor* Instanzen vom Typ *IOSocket*, welche zum Datenaustausch zwischen HLA und Simulationswerkzeug notwendig sind. *IOSocket*s sind intern mit einem FIFO ausgestattet, in dem von der HLA empfangene Daten zwischengepuffert werden können.

Auf der entgegengesetzten Seite bietet die *Ambassador* Klasse Zugriff auf die RTI Ambassador Schnittstellen und implementiert zugleich die Federate Ambassador Callback Methoden. Die HLA Ambassador Schnittstellen werden schließlich durch die *HLAAdaptor* Klasse gekapselt.

5.4.2.1. Simulation Object Metamodel

Abb. 5.6 zeigt einen Auszug der Klassenstruktur des SOMM. Die Klassen bilden eine objektorientierte Metarepräsentation eines SOM. Ein solches wird durch Instanziierung der *SimulationObjectModel* Klasse erzeugt, welche das SOMM enthält. Die Speicherung von Daten im SOM erfolgt dynamisch.

Die Struktur des SOMM wurde in Anlehnung an die OMT Spezifikation des HLA Standards [24] entwickelt. Im HLA OMT sind Komponenten in Tabellen

strukturiert. Objektklassen im OMT können hierarchisch verschachtelt werden, was äquivalent zur Vererbung in einer objektorientierten Programmiersprache ist. Komponenten einer OMT Tabelle können zudem Komponenten anderer Tabellen referenzieren. Beispielsweise muss ein Attribut in der Attributtabelle die Objektklasse referenzieren, zu der es gehört. Zusätzlich muss jedem Attribut ein Datentyp einer Datentyptabelle zugewiesen werden. Jede Datentyptabelle beinhaltet Typen mit identischen Charakteristika, wie z.B. einfache Datentypen oder Arraytypen.

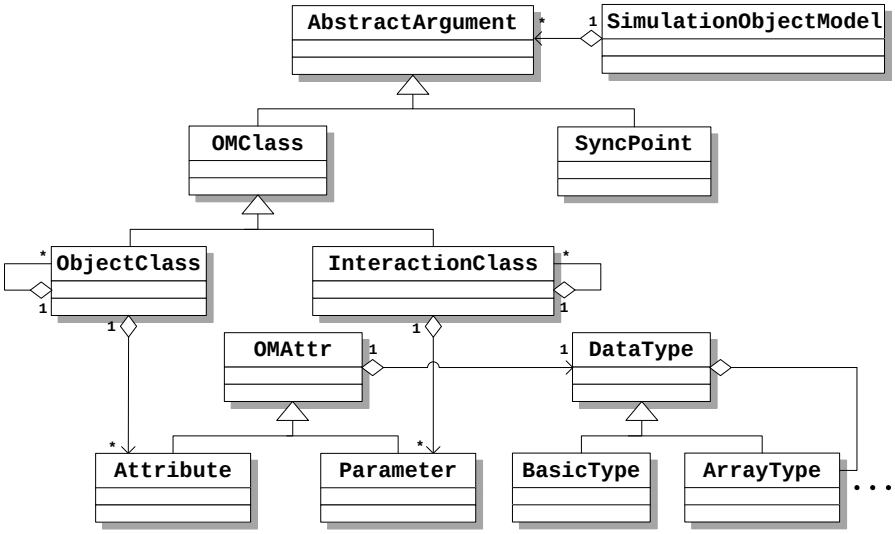


Abbildung 5.6.: Simulation Object Metamodel

Relationen zwischen Objektklassen eines SOM werden mit Hilfe der *ObjectClass* und *InteractionClass* C++ Klassen auf objektorientierte Aggregations- und/oder Vererbungsbeziehungen abgebildet (siehe Abb. 5.6):

Hierarchische Verschachtelung wird durch Selbstaggregation modelliert. Zugehörigkeiten von Attributen und Parametern werden ebenfalls durch Aggregation zwischen *Object/InteractionClass* Klassen und *Attribute/Parameter* Klassen modelliert. Attribute und Parameter besitzen eine Referenz auf exakt einen Datentypen. Für die Erzeugung von Polymorphie bei zusammengesetzten Datentypen wie Arrays wird Vererbung eingesetzt.

In der aktuellen Implementierung der SOMMlib können SOM Tabellen in einer leicht lesbaren Form anhand von verschachtelten C++ Methodenaufrufen spezifiziert werden (siehe Listing 5.1). Zu diesem Zweck muss man von der *SimulationObjectModel* Klasse erben und die *generateObjectModel()* Methode implemen-

tieren. Eine Alternative wäre die Repräsentation eines SOM in XML Syntax. Zur Laufzeit dienen Objekte der Datentypklassen zum Konvertieren zwischen C++ und HLA Datentypen und umgekehrt.

```
virtual void generateObjectModel()
{
 //Synchronization table
 SYNCPOINT("READY_TO_RUN", ...);
 ...
 //Basic data representation table
 BASICTYPE("HLAinteger32LE","32", ...);
 ...
 //Array datatype table
 ARRAYTYPE("HLAInteger32LEArray","HLAAInteger32LE", ...);
 ...
 //Object class structure table
 OBJECTCLASS("ObjectRoot", "N",
  OBJECTCLASSL("wifiDownMsg", "S"),
  OBJECTCLASSL("wifiUpMsg", "P"),
  OBJECTCLASSL("internalSetMsg", "S"),
  OBJECTCLASSL("internalGetMsg", "P")
 );
 //Attribute table
  OBJECT("ObjectRoot.wifiDownMsg",
    ATTRIBUTE("vehicleID", "HLAinteger32LE", ...),
    ATTRIBUTE("size", "HLAinteger32LE", ...),
    ATTRIBUTE("payload", "HLAinteger32LEArray", ...)
  );
 ...
}
```

Listing 5.1: Beispiel C++ Code zur SOM Instanziierung

5.4.2.2. Behavioral Interface Metamodel

Die Klassenstruktur des BIMM ist in Abb. 5.7 illustriert. Äquivalent zu einem SOM wird ein BIM durch Instanziierung der *BehavioralInterfaceModel* Klasse erzeugt, welche das BIMM enthält. Die Speicherung eines BIM erfolgt dynamisch.

Durch Ableitung von der *BehavioralInterfaceModel* Klasse kann ein endlicher Zustandsautomat namens BIM FSM spezifiziert werden. Eine BIM FSM legt das erlaubte Schnittstellenverhalten fest. BIM FSMs definieren eine valide Aufrufreihenfolge von Schnittstellenmethoden der *ToolAdaptor* und *IOSocket* Klassen sowie Methoden der *HLAAdaptor* Klasse. Dadurch können unterschiedliche Interaktions- und Synchronisationsmechanismen modelliert werden. Die semantische Abbildung von BIM Syntax auf die HLA erfolgt durch Spezialisierung der in Abb. 5.7 gezeigten Basisklassen und einen Scheduler. Durch die Vererbung

wird eine statische Abbildung festgelegt, durch den Scheduler wird die statische um eine dynamische Abbildung ergänzt.

Mit Hilfe des Schedulers sind BIM FSM Spezifikationen ausführbar. Sie können damit zur Laufzeit unmittelbar zur Sicherstellung korrekter dynamischer Semantik verwendet werden. Insbesondere ist eine BIM FSM während der dynamischen Integrations- und Testphase einer Federation zum Debugging nützlich.

Vernachlässigt man Besonderheiten wie hierarchische Zustände[5] oder abstrakte Argumente[6], so kann die Implementierung des Verhaltens einer BIM FSM im Scheduler der SDEMMlib mathematisch als ein Tupel $(S, \Sigma, \Omega, \delta, s_0, s_e, S_{int})$ beschrieben werden. Dabei ist S die Menge der Zustände, Σ das Eingangsalphabet, Ω das Ausgangsalphabet, $\delta : S \times \Sigma \rightarrow S \times \Omega$ die Transitionsfunktion, s_0 der Startzustand, s_e der Endzustand und S_{int} die Menge der sog. Interaktionszustände, deren besondere Bedeutung weiter unten erläutert wird.

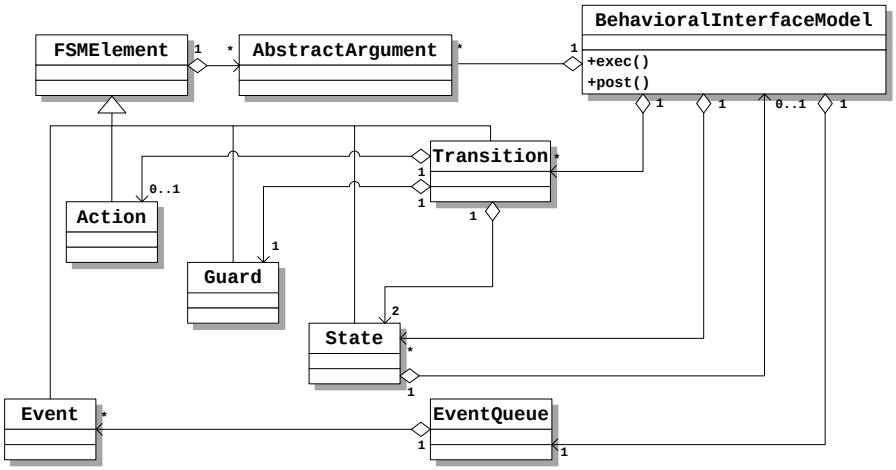


Abbildung 5.7.: Behavioral Interface Metamodel

Die Transitionsfunktion beschreibt den Übergang von Ausgangszuständen in Zielzustände. In der *BehavioralInterfaceModel* Klasse ist die Transitionsfunktion mit Hilfe einer Transitionstabelle realisiert. Diese speichert Einträge vom Typ *Transition*. Eine einzelne *Transition* beschreibt den Übergang von einem Ausgangszustand s_1 in einen Zielzustand s_2. Zustände sind durch die *State* Klasse realisiert. Die Nutzung einer Transition ist durch einen *Guard* geschützt und mit der Ausführung einer Aktion vom Typ *Action* verbunden. Durch die explizite Spezi-

[5] BIM FSMs erlauben die Definition von hierarchischen Zuständen: Ein Zustand kann selbst wieder eine vollständige FSM enthalten. Dadurch wird eine Zustandsexplosion vermieden.

[6] Abstrakte Argumente dienen zur Referenzierung von Elementen des SOM.

fikation erlaubter Ereignisse anhand von Guards wird garantiert, dass nur valide (d.h. spezifizierte) Ereignisse während der gesamten Ausführung auftreten.

Eingangswerte Σ in die Transitionsfunktion entsprechen Ereignissen (Instanzen der *Event* Klasse). Das aktuelle Ereignis entspricht immer dem obersten *Event* in der *EventQueue* (siehe Abb. 5.7). Ereignisse können beim Aufruf von Schnittstellenmethoden der *ToolAdaptor* oder *HLAAdaptor* Klassen oder durch die Ausführung einer *Action* generiert werden. Die Erzeugung eines Ereignisses, der Aufruf einer Schnittstellenmethode oder der Schreibzugriff auf ein *AbstractArgument* durch eine *Action* sind äquivalent zur Generierung eines Ausgangswertes aus Ω.

Der Begriff des Interaktionszustandes wird klar, wenn man den im Scheduler implementierten Ausführungsmechanismus für BIM FSMs genauer betrachtet. Dieser basiert auf einer iterativen Ausführung zweier Methoden *exec()* und *post()*. Diese Methoden sind durch die Action Methods *fire()* und *postfire()* von PtII inspiriert. Eine BIM Iteration besteht aus einem Aufruf von *exec()* und einem darauffolgenden Aufruf von *post()*. Dabei geschieht Folgendes:

- *exec()*: Nimm das vorderste Event von der Event Queue und wähle die Transition, deren Guard den Wert „true“ zurückliefert. Die gewählte Transition wird als aktive Transition bezeichnet. Stelle sicher, dass immer nur eine aktive Transition existiert. Falls aufgrund einer fehlerhaften Spezifikation mehrere aktive Transitionen existieren sollten, so brich die Ausführung ab.
- *post()*: Führe einen Zustandswechsel aus und setze dazu den aktuellen Zustand auf den Zielzustand der aktiven Transition. Wenn der Zielzustand der Toplevel Endzustand oder ein Interaktionszustand ist, dann gib eine 0 zurück um zu signalisieren, dass der Simulator entweder terminieren soll oder die Kontrolle zeitweise an den Simulator übergeben werden soll. Gib in allen anderen Fällen eine 1 zurück um zu signalisieren, dass eine weitere Iteration durchgeführt werden soll.

Insgesamt ist ein Interaktionszustand damit ein Label, an dem die Ausführung der BIM FSM unterbrochen wird, um die Kontrolle temporär dem umgebenden Simulator zu überlassen (z.B. zur Ausführung der lokalen Simulation). Der Zustand, in dem die Ausführung zuletzt unterbrochen wurde, entspricht dem Eintrittspunkt bei der nächsten Iteration.

Aktuell erfolgt die Modellierung einer BIM FSM ähnlich wie die Modellierung eines SOM, nämlich durch Ableitung von *BehavioralInterfaceModel* und Implementierung der *generateBIM()* mit verschachtelten Methodenaufrufen (siehe Codebeispiel 5.2). Auch hier ist XML als konkrete Syntax eine denkbare Alternative.

```
virtual void generateBIM()
{
 //State table
```

```
  STATE("S_START");
  STATE("S_READ");
  STATE("S_UPDATE");
  STATE("S_ADVANCE");
  STATE("S_GRANT");
  ...
  //Special states
  START_STATE("S_START");
  INTERACTION_STATE("S_READ");
  END_STATE("S_GRANT");
  ...
  //Transition table
  TRANSITION(
   STATEREF("S_READ"),
   STATEREF("S_ADVANCE"),
   ACTION("HLA13_A5_12_Action_NextEventReqAvailable"),
   EG("Event_SetNextBarrier")
  ),
  ...
}
```

Listing 5.2: Beispiel C++ Code zur BIM Instanziierung

5.5. Umsetzung der semi-automatischen Werkzeugkopplung

Im Folgenden wird davon ausgegangen, dass die Schritte 1 bis 3 der Vorgehensweise aus Abschnitt 5.3.2 bereits erfolgreich durchgeführt worden sind. Für die Umsetzung der Schritte 4 bis 6 sind einige Erweiterungen zum PtII Framework hinzuzufügen. Diese Erweiterungen basieren auf den nachfolgend aufgeführten neuen Modellierungselementen:

- *HLAComposite*: Ein Actor, der eine HLA Federation repräsentiert.
- *HLAFederate*: Ein Actor, der ein HLA Federate repräsentiert.
- *HLAObjectClass*: Basisklasse eines Actors, der eine HLA Object Class repräsentiert[7].
- *HLADEObjectClass*: DE Actor, der eine HLA Object Class in der DE Domäne repräsentiert.
- *HLAGenDirector*: Ein Director zur Generierung von HLA Schnittstellen.
- *HLADEDirector*: Ein DE Director mit HLA Schnittstelle.

[7] Aktuell existiert auch ein *HLAInteractionClass* Actor, welcher aber im Folgenden nicht weiter betrachtet wird.

Nachfolgend wird die Implementierung der Schritte 4 bis 6 am Beispiel einer verteilten PtII Simulation und Verwendung zweier PtII Instanzen erläutert. Dabei sollen die PtII Instanzen über die DE Domäne gekoppelt sein. Einen Überblick über den Entwurfsfluss gibt Abb. 5.8. Die automatische Generierung von Schnittstellen wurde im Prototypen nur soweit umgesetzt, wie es für eine Demonstration der Machbarkeit notwendig war.

5.5.1. Konfiguration des Datenaustauschs

Die Konfiguration des Datenaustauschs in einem in PtII Syntax beschriebenen Federationmodell *FM* setzt die Spezifikation der erlaubten statischen und dynamischen Semantik entsprechend der Schritte 1 bis 3 der Vorgehensweise aus Abb. 5.3 voraus. Dies beinhaltet, dass das Federationmodell in einem Kompositum des Typs *HLAComposite* beschrieben wurde und die Actors, welche die Federates repräsentieren, vom Typ *HLAFederate* sind. Ist dies nicht der Fall, so müssen die verwendeten Modellierungselemente durch besagte Elemente zunächst ausgetauscht werden.

5.5.1.1. Statische Semantik des Datenaustauschs

Die innerhalb einer Federation auszutauschenden Daten ergeben sich unmittelbar aus den Namen und Typen der Modellierungsartefakte in zugehörigen Federationmodell und deren Verknüpfung. Die Namen der Actors in einem *HLAComposite* Actor entsprechen den Namen der durch die Actors repräsentierten HLA FOM/SOM Artefakte.

Die HLA Publish/Subscribe Beziehungen, wie sie im FOM spezifiziert sein müssen, werden aus der Orientierung der Links zwischen *HLAFederate*, *HLAComposite* und *HLAObjectClass* Actors abgeleitet. Der Aufbau der Object Classes, inkl. deren Attribute und Datentypen, wird aus den Datentypen der Ports der verschiedenen *HLAObjectClass* Actors hergeleitet. Ein Datenaustausch eines PtII Modells mit anderen Federates wird über die Ports zum umgebenden *HLAComposite* Actor bestimmt. In der aktuellen Implementierung wird keine Unterscheidung zwischen HLA Object Class und HLA Object Instance gemacht.

Im Beispiel aus Abb. 5.8 ist das Federationmodell in der ersten PtII Instanz *PtolemyII_First* zu sehen. Diese soll mit einer zweiten PtII Instanz, welche durch einen *HLAFederate* Actor namens *PtolemyII_Second* repräsentiert wird, co-simuliert werden[8]. Die Verbindung zum lokalen Modell von *PtolemyII_First* wird über Ports

[8] In *PtolemyII_Second* muss dafür ein analoges Federationmodell existieren, bei dem die Links eine umgekehrte Flussrichtung aufweisen.

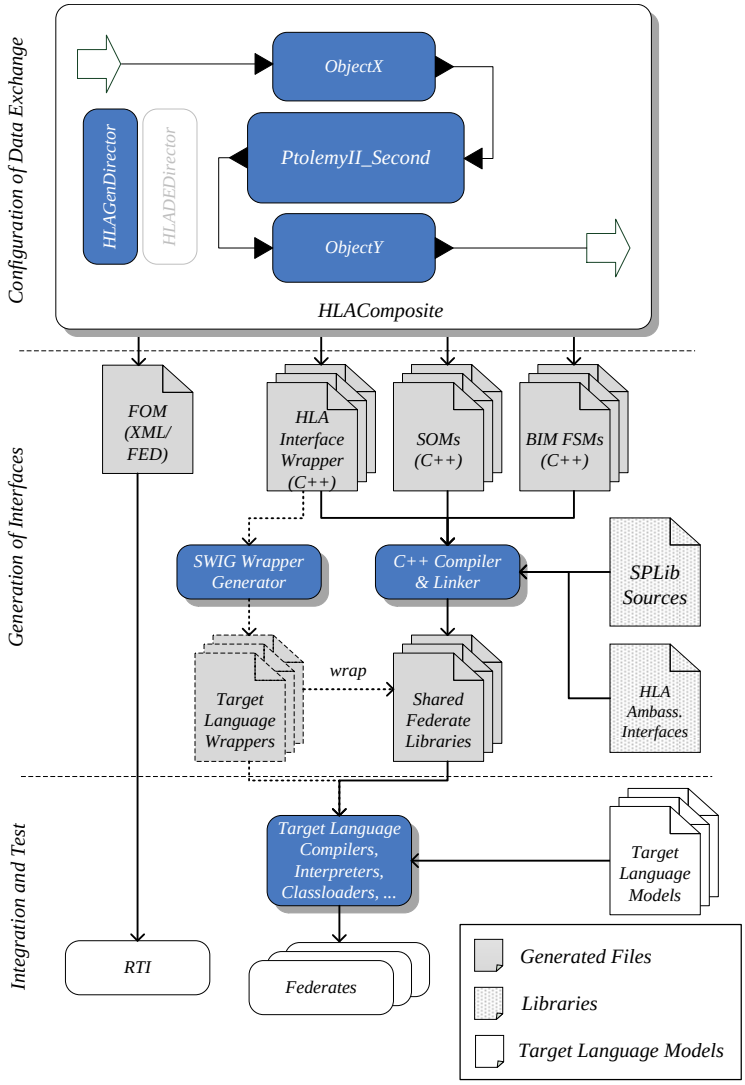


Abbildung 5.8.: Semi-automatische Werkzeugkopplung (*PtolemyII_First* Sicht)

am umgebenden *HLAComposite* hergestellt. Das lokale PtII Modell und der externe Simulator kommunizieren ausschließlich über die beiden *HLAObjectClass* Actors „ObjectX“ und „ObjectY“.

5.5.1.2. Dynamische Semantik des Datenaustauschs

Die erlaubten Interaktionen in einer Federation werden anhand der an die *HLAFederate* Actors annotierten Parameter, insbesondere anhand der durch eine C++ Datei spezifizierten BIM FSM, abgeleitet.

Die abstrakte Semantik von PtII und Spezifika der DE Domäne wurden bereits in Abschnitt 2.3.4 erläutert. Betrachtet man eine komplette PtII Ausführungssequenz entsprechend Abb. 2.17, so beinhaltet diese (unter Vernachlässigung der Teilphasen) Initialization Phase, Execution Phase und Wrapup Phase.

Im betrachteten Beispiel ergibt sich die Spezifikation der dynamischen Semantik des Datenaustauschs aus den Berechnungsmodellen, die jeweils in den beiden PtII Instanzen zur Kopplung verwendet werden. Wie bereits in Abschnitt 5.3.2 angemerkt, reduziert sich die Komplexität einer BIM FSM, wenn die zu integrierenden Simulatoren mit ein und derselben PtII Domäne charakterisiert werden können. Bei der Kopplung von *PtolemyII_First* und *PtolemyII_Second* über die DE Domäne ist die Charakterisierung trivial: Sie ist DE. Aufgrund der Symmetrie, genügt die Entwicklung einer einzigen BIM FSM.

Um die erlaubte dynamische Semantik für die drei Ausführungsphasen separat modellieren zu können, spiegeln sich die Phasen 1:1 als *S_INIT*, *S_EXECUTE* und *S_END* Zustände auf dem Toplevel der BIM FSM wider (vgl. Abb. 5.9). Zur Synchronisation existieren zusätzlich noch die *S_SYNC_READY_TO_RUN* und *S_SYNC_SHUTDOWN* Zustände. Außer dem *S_END* Zustand sind alle Toplevel Zustände hierarchisch, sie besitzen also eine Verfeinerung. Die tatsächliche Funktionalität ist jeweils durch die Verfeinerungen implementiert.

Die beiden Zustände *S_SYNC_READY_TO_RUN* und *S_SYNC_SHUTDOWN* implementieren eine Barriersynchronisation, welche sicherstellt, dass alle Federates gleichzeitig von *S_INIT* nach *S_EXECUTE* und von *S_EXECUTE* nach *S_END* übergehen.

In der Verfeinerung des *S_INIT* Zustands werden HLA Initialisierungsroutinen zur Erzeugung der Federation, der Publish/Subscribe Beziehungen sowie zur Registrierung von HLA Object Instance Attributen ausgeführt. Intern besitzt der Zustand *S_INIT* einen Interaktionszustand (vgl. Abschnitt 5.4.2.2), so dass die Registrierung vom lokalen Simulator getriggert werden kann.

Zur verteilten Kopplung speziell der DE Domäne von PtII muss im *S_EXECUTE* Zustand die lokale Kausalitätsbedingung aus Definition 2.2 erfüllt sein. Möglichkeiten hierzu wurden bereits ausführlich im vorigen Kapitel im Kontext von

SystemC behandelt. Im Unterschied zum vorigen Kapitel kann die lokale Kausalitätsbedingung nun mit Unterstützung der „High-level“ Services der HLA erfüllt werden. Zur Erläuterung ist in Abb. 5.9 die Verfeinerung des *S_EXECUTE* Zustands dargestellt.

S_EXECUTE ist ein hierarchischer Zustand und enthält einen Interaktionszustand namens *S_READ*. Interaktionen mit dem Simulator erfolgen nur dann, wenn kein Ereignis in der Event Queue der BIM FSM vorhanden ist. Dies wird durch das *Event_Absent* Ereignis auf der in *S_READ* eingehenden Transition symbolisiert. Anschließend wird durch die restliche FSM Spezifikation der Zeitfortschritt modelliert.

Die Implementierung basiert auf dem *Next_Event_Request_Available* (NERA) Service, welcher Teil des HLA 1.3 Time Managements ist. Dessen Funktionsweise ist in Anhang B erläutert. Insbesondere erlaubt NERA einen Lookahead von Null, wodurch Deadlocks im Fall eines Microsteps μ vermieden werden können. Um den NERA Service nutzen zu können, müssen alle HLA Object Classes und damit deren Attribute zusätzlich per Parameter als *HLAReliable* deklariert werden.

Der NERA Service garantiert nicht, dass alle Nachrichten empfangen werden, deren Zeitstempel, dem durch den *Time_Advance_Grant* (TAG) Service zurückgelieferten Zeitstempel entsprechen. Dies hat eine nicht-deterministische Anzahl an Microsteps zur Folge. Die hier beschriebene Implementierung ist damit trotz Zero Lookahead nur bis auf die Modeltime τ genau (vgl. Abschnitt 2.3.4.2).

Sobald das *Event_Shutdown* Ereignis auftritt (dieses muss von der lokalen Simulation erzeugt werden), wechselt die FSM in den *S_SYNC_SHUTDOWN*. Anschließend werden die Shutdown Prozeduren der HLA abgearbeitet. Schließlich terminiert die FSM im *S_END* Zustand.

5.5.2. Generierung von Schnittstellen

Mit Hilfe eines Federationmodells und dem durch SDEMMlib bereitgestellten Baukasten können Schnittstellen zur Co-Simulation für verschiedene Simulationswerkzeuge generiert werden. Dazu muss die PtII Syntax eines Federationmodells über eine Modellabfrage in C++ bzw. SDEMMlib Syntax übersetzt werden. Anschließend kann mit Hilfe des *Simplified Wrapper and Interface Generators (SWIG)* [47] die Einbettung von kompiliertem C++ Code in Java oder diverse Skriptsprachen automatisiert werden. Folgende Artefakte müssen generiert werden:

- **FOM**: Das FOM wird als FED Datei (vgl. Abschnitt 3.2.3.2) exportiert. Die FED Datei dient zur Konfiguration der durch CERTI bereitgestellten HLA RTI Implementierung.

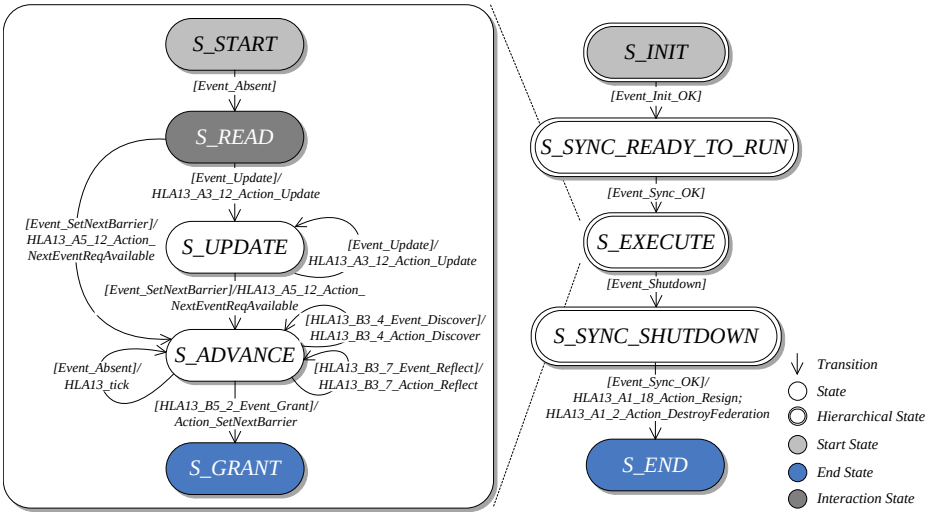


Abbildung 5.9.: Behavioral Interface Model FSM für die PtII DE Domäne

- **SOM**: Die SOMs der Federates werden als C++ Code exportiert (vgl. Abschnitt 5.4.2.1).
- **BIM**: Die BIMs der Federates werden als C++ Code exportiert (vgl. Abschnitt 5.4.2.2). Aktuell werden dazu direkt annotierte BIM C++ Beschreibungen wiederverwendet.
- **HLA Interface Wrapper**: Diese beinhalten abgeleitete Typen der *IOSocket* Klasse sowie einen C++ Wrapper zur vollständigen Kapselung des gesamten C++ Codes. Daneben werden noch SWIG Schnittstellendefinitionen (.i Dateien) erzeugt. Diese sind notwendig, um den C++ Wrapper in einen weiteren C++-fremden Wrapper zu verpacken. Die .i Dateien dienen zur Spezifikation der Abbildung von C++ Datentypen auf Datentypen anderer Sprachen.

Eine Möglichkeit zur Umsetzung der Modellabfrage und der Schnittstellengenerierung ist die Entwicklung eines speziellen PtII Directors für diesen Zweck. Die Machbarkeit wurde anhand eines Directors namens *HLAGenDirector* prototytpisch demonstriert. Dieser muss in den *HLAComposite* Actor des Federationmodells eingefügt werden, für das der Schnittstellencode erzeugt werden soll. Der *HLAGenDirector* hat Zugriff auf verschiedene *ModelInterpreter* Klassen. Diese liefern konkrete (programmiersprachen-spezifische) Interpretationen für das in visueller PtII Syntax spezifizierte Modell, mit dem Ziel der Generierung der oben genannten Artefakte.

Die Artefakte werden dann zusammen mit den Klassen der SDEMMlib in eine C++ Shared Library kompiliert. Falls der Zielsimulator in C++ geschrieben ist (z.B. SystemC), dann kann die Shared Library direkt mit dem Simulationskernel oder dem Modell gelinkt werden. Basiert der Simulator auf einer anderen Sprache wie Java oder C# (PtII basiert z.B auf Java), dann kann auf SWIG zurückgegriffen werden.

5.5.2.1. HLA Interface Wrapper

Der HLA Interface Wrapper tritt für den Zielsimulator als eine einzelne Klasse in Erscheinung, die eine feste Anzahl Kontrollflussmethoden und eine variable Anzahl an Datenflussmethoden besitzt. Erstere werden durch die *ToolAdaptor* Klasse der SDEMMlib bereitgestellt, Letztere durch die zuvor generierten *IOSocket* Klassen. Zu den Kontrollflussmethoden gehören aktuell insbesondere

- *getState()*: Gibt den aktuellen Zustand der BIM FSM zurück.
- *setNextBarrier()*: Setzte die Zeitbarriere, bis zu welcher der lokale Simulator voranschreiten kann.
- *getNextBarrier()*: Gibt den Wert der Zeitbarriere zurück, bis zu welcher der lokale Simulator tatsächlich voranschreiten darf.
- *iterate()*: Iteriere die BIM FSM. Ein Aufruf dieser Methode führt solange Sequenzen von *exec()* und *post()* aus, bis *post()* eine 0 zurück gibt. Diese Methode muss regelmäßig vom Simulator aufgerufen werden, damit die BIM FSM voranschreitet.
- *end()*: Generiere ein *Event_Shutdown* Ereignis zur Terminierung der BIM FSM.

Im Unterschied zu den Kontrollflussmethoden haben Datenflussmethoden keine feste Signatur. Die Methoden zum Lesen und Schreiben beginnen mit *read* oder *write* gefolgt vom Namen der Objektklasse, auf die zugegriffen wird. Anzahl und Typ der Attribute einer Objektklasse definieren Anzahl und Typ weiterer Methodenparameter. Zu den Datenflussmethoden gehören aktuell insbesondere

- *readX()*: Lese die letzte Aktualisierung von *IOSocket* / Objektklasse "X".
- *writeX()*: Schreibe eine Aktualisierung in *IOSocket* /Objektklasse "X".
- *getSocketQueueSize()*: Gib den Füllstand eines spezifizierten *IOSocket*s zurück.
- *update()*: Erzeuge ein *Sim_Event_Update* Ereignis für die BIM FSM. Zusätzlich muss dem Aufruf der Name des Sockets und ein Zeitstempel übergeben werden.

- *popSocket()*: Lösche die nächste empfangene Aktualisierung von einem als Parameter spezifizierten *IOSocket*.

5.5.3. Integration und Test

Die Art und Weise der Integration ist stark von dem zu integrierenden Zielsimulator und dem zugrundeliegenden Berechnungsmodell abhängig. Hier wird exemplarisch die Integration eines HLA Interface Wrappers in die DE Domäne von PtII betrachtet. Für die Integration in PtII dient als Ausgangspunkt die Spezifikation eines Federationmodells. Mit SDEMMlib kann der Prozess der Integration dann auf die Integration der Kontrollflussmethoden und die Integration der Datenflussmethoden heruntergebrochen werden.

5.5.3.1. Integration der Kontrollflussmethoden

Für die Integration der Kontrollflussmethoden müssen die BIM Toplevel Zustände *S_INIT*, *S_EXECUTE* und *S_END* auf die Action Methods *preinitialize()*, *initialize()*, *prefire()*, *fire()*, *postfire()* und *wrapup()* des DE Directors in geeigneter Weise abgebildet werden. Dadurch werden indirekt auch passende HLA Service Calls auf die Ausführungsphasen des Simulators abgebildet.

Zu diesem Zweck müssen Aufrufe der *iterate()* Methode des HLA Interface Wrappers so innerhalb der Action Methods des PtII Directors erfolgen, dass der Zustand der BIM FSM mit den Ausführungsphasen möglichst korrespondiert. Da die BIM Zustände nicht auf der Ebene der PtII Teilphasen definiert wurden, existiert hier ein gewisser Freiheitsgrad. Eine exaktere Spezifikation der BIM FSM könnte daher alle PtII Teilphasen umfassen. Umgekehrt ist die BIM FSM auf diese Weise universeller einsetzbar, was innerhalb der noch folgenden Fallstudien demonstriert wird.

Für die Implementierung wurde ein spezieller *HLADEDirector* entwickelt. Dieser ist vom originalen *DEDirector* abgeleitet. Da PtII in Java geschrieben ist, wird der HLA Interface Wrapper ein weiteres Mal mit Hife von SWIG verpackt und bei der Ausführung dynamisch geladen. Aktuell werden der *S_INIT* und der *S_SYNC_READY_TO_RUN* Zustand der BIM FSM aus Abschnitt 5.5.1.2 in die *initialize()* Methode abgebildet, der *S_EXECUTE* Zustand wird in die *fire()* Methode abgebildet und der *S_SYNC_SHUTDOWN* sowie der *S_END* Zustand werden in die *wrapup()* Methode abgebildet. Der Ablauf ist wir folgt (vgl. Beispiel aus Abb. 5.10):

Sobald die *fire()* Methode des Directors aufgerufen wird, bestimmt dieser, ausgehend vom aktuellen Simulationszeitpunkt τ^{cur}, den aus seiner Sicht frühesten Zeitpunkt τ^{min}, an dem das nächste Mal mit der Federation synchronisiert wer-

den muss. τ^{min} entspricht dem Zeitpunkt, an dem lokal frühestens ein neues Ereignis auftreten kann. τ^{min} kann durch die PtII Methode *getModelNextIterationTime()* abgefragt werden. Wenn τ^{min} größer als die mit dem letzten Aufruf von *getNextBarrier()* gewährte Zeitbarriere ist, macht der Director eine Anfrage für einen Zeitfortschritt, indem er *setNextBarrier()*) aufruft und $\tau^{req} = \tau^{min}$ als Anfragewert übergibt.

Anschließend iteriert der Director die BIM FSM per *iterate()*. Dabei wird τ^{req} dem HLA NERA Service übergeben. Über den TAG Service und Aufruf von *getNextBarrier()*) wird dem Director schließlich der von der HLA berechnete Zeitpunkt τ^{grant} mitgeteilt, bis zu dem der Director tatsächlich voranschreiten darf. Im Allgemeinen ist $\tau^{grant} \leq \tau^{min}$. Durch Einfügen eines Pure Events mit Zeitstempel $\tau^{pure} = \tau^{grant}$ wird die lokale Simulation zur erneuten Synchronisation bei Erreichen von τ^{grant} gezwungen. Dies geschieht durch Setzen des Zeitpunkts, an dem der Director das nächste Mal „gefeuert“ werden soll per *fireContainerAt()* auf τ^{grant}. Danach werden Aktualisierungen von Instanz-Attributen behandelt.

5.5.3.2. Integration der Datenflussmethoden

Für die Integration der Datenflussmethoden wurde durch Ableitung von *HLAObjectClass* ein *HLADEObjectClass* Actor erzeugt. Falls nicht schon geschehen, müssen die Actors der *HLAObjectClass* Basisklasse durch Actors vom Typ *HLADEObjectClass* ausgetauscht werden. Da PtII eine zentrale Rolle innerhalb der Simulationsumgebung einnimmt, hat Flexibilität bei der Implementierung hohe Priorität. Dies wird aktuell durch dynamische Ableitung von Signaturen der datenflussrelevanten Methoden des HLA Interface Wrappers erreicht.

Die Steuerung des Datenflusses liegt in der Hand des *HLADEDirector*s in Kombination mit den *HLADEObjectClass* Actors. Während einer Simulationsausführung haben *HLADEObjectClass* Actors nur dann eine aktive Rolle, wenn sie über Relationen mit den Ports des umgebenden *HLAComposite* Actors verbunden sind. Sie repräsentieren dann die HLA Object Classes, über die das restliche lokale PtII Modell in die Federation eingekoppelt wird. Alle anderen *HLAObjectClass* Actors sowie die *HLAFederate* Actors sind passiv und werden niemals gefeuert. Der Ablauf beim Empfangen und Senden über aktive *HLADEObjectClass* Actors ist wie folgt:

Zum Lesen aus dem HLA Interface Wrapper ist der *HLADEDirector* mit einer Methode namens *reflectDir()* ausgestattet. Diese Methode wird immer dann vom *HLADEDirector* aufgerufen, wenn die zuletzt genehmigte Zeitbarriere erreicht wurde (vgl. Abb. 5.10). Die Methode prüft dann, ob einer der *IOSocket* Puffer neue empfangene Aktualisierungen (sog. Reflections) enthält (Aufruf von *getSocketQueueSize()*). Wenn dies der Fall ist, dann wird die vollständige Signatur der zugehörigen *read()* Methode mit Hilfe des SOMs und der *ModelInterpreter*

Klassen zur Laufzeit hergeleitet. Die Reflection wird dann in ein PtII Token umgewandelt und zum entsprechenden *HLAObjectClass* Actor weitergeleitet. Zu diesem Zweck ist der *HLADEObjectClass* Actor mit einer *reflectAct(token,time)* Methode ausgestattet. Diese Methode speichert das übergebene Token (ähnlich wie in Abschnitt 4.5.3) in einer internen Queue und ruft dann die *fireAt(time)* Methode auf. Dies stellt sicher, dass der Actor exakt zu dem Zeitpunkt wieder gefeuert wird, an dem das empfangene Token zum PtII Modell über einen entsprechenden Ausgangsport weitergeleitet werden soll. Der Zeitparameter definiert somit eine zukünftige Zeitbarriere.

Im Sendefall überprüft ein aktiver *HLADEObjectClass* Actor seine PtII Eingangsports immer dann hinsichtlich neu verfügbarer Tokens, wenn er gefeuert wird. Wenn Tokens vorhanden sind, so werden diese zum *HLADEDirector* durch Aufruf von *updateDir(token)* weitergeleitet. Diese Methode konvertiert das übergebene Token mit Hilfe der *ModelInterpreter* Klassen in einen *write()* und einen *update()* Methodenaufruf auf dem HLA Interface Wrapper. Der Zeitstempel der generierten Aktualisierung entspricht der aktuellen lokalen Zeit.

5.5.3.3. Einhaltung der Kausalität

Single-Federation Modus

Im Fall des Single-Federation Modus ist mit dem durch die BIM FSM aus Abschnitt 5.5.1.2 definierten Ablauf eine Verletzung der zeitlichen Kausalität ausgeschlossen[9]: Der *HLADEDirector* kennt vor und während des Synchronisationsvorgangs alle zukünftigen Ereignisse innerhalb der lokalen PtII Simulation, da die restliche lokale PtII Simulation während des Synchronisationsvorgangs pausiert. Außer durch die vom *HLADEDirector* kontrollierte Federation kann zum aktuellen Zeitpunkt τ^{cur} innerhalb der PtII Simulation daher kein Ereignis mehr generiert werden, dessen Zeitstempel kleiner als das per *setNextBarrier()* an den HLA Interface Wrapper übergebene τ^{min} ist. Falls nach der Iteration des HLA Interface Wrappers $\tau^{grant} < \tau^{min}$ ist, so wird dies unmittelbar vom *HLADEDirector* durch Einfügen des entsprechenden Pure Events mit $\tau^{pure} = \tau^{grant}$ berücksichtigt.

Multi-Federation Modus

Im Multi-Federation Modus ist die Anwendung des beschriebenen Schemas in den vorhandenen *HLADEDirector*s zwar notwendig, aber nicht in allen Fällen hinreichend für den Erhalt der Kausalität. Dies soll anhand der Einbettung mehrerer *HLAComposite* Actors der HLA DE Domäne in eine Toplevel DE Domäne verdeutlicht werden.

[9] Diese Aussage bezieht sich hier auf eine Kausalitätsverletzung bzgl. τ. Microsteps μ werden an dieser Stelle vernachlässigt.

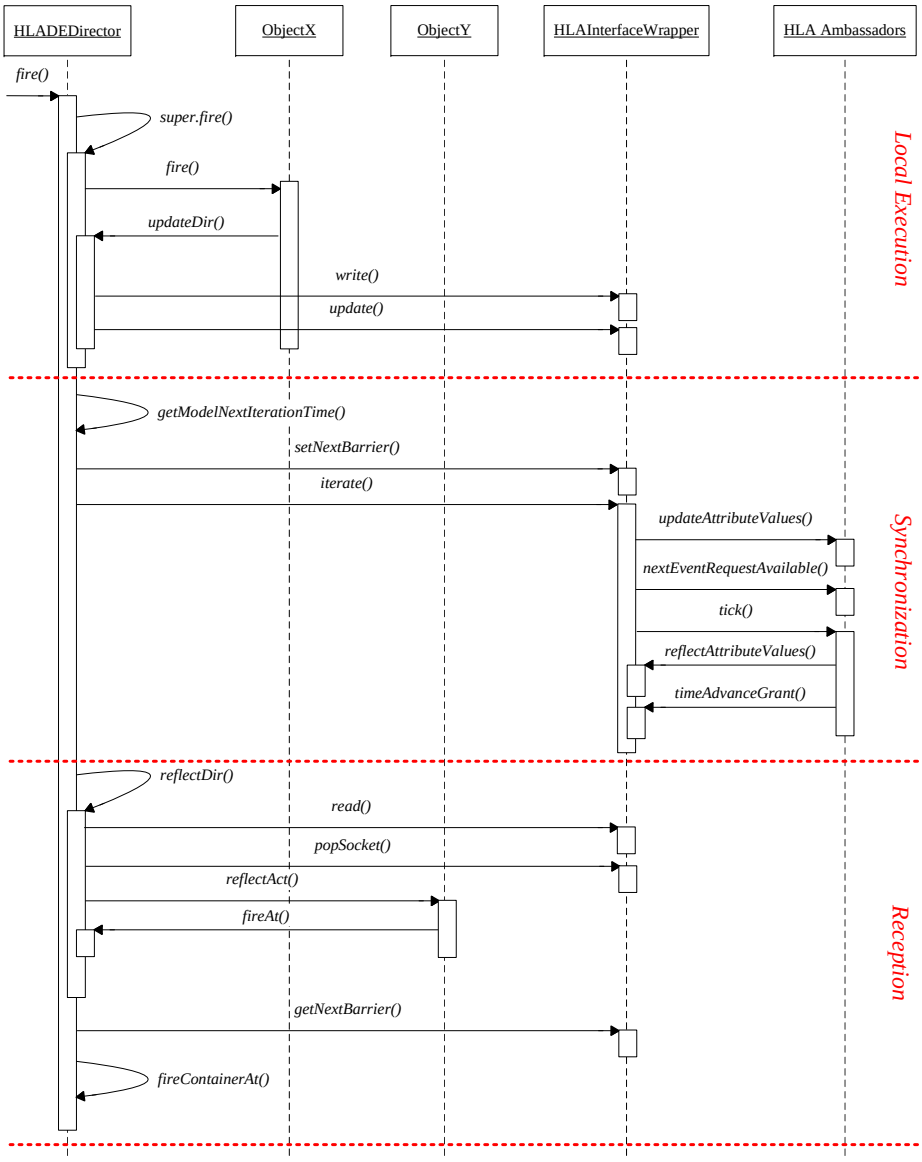


Abbildung 5.10.: Beispielhafte Aufrufsequenz von Methoden im *HLADEDirector* (Sicht von *PtolemyII_First*)

Im Folgenden wird angenommen, es existieren *HLAComposite* Actors c_i mit $1 \leq i \leq N$, $N \in \mathbb{N}$ und $N \geq 2$. Jeder Actor c_i beinhaltet einen Director d_i vom Typ *HLADEDirector* sowie ein Federationmodell, das im Single-Federation Modus korrekt und ohne Kausalitätsverletzungen ausführt. Die Federationmodelle in Kombination mit den $d_1 ... d_N$ koppeln die Federations $f_1 ... f_N$ in das restliche PtII Modell ein. Dies ist in Abb. 5.11 beispielhaft anhand zweier Federationmodelle c_n und c_m dargestellt.

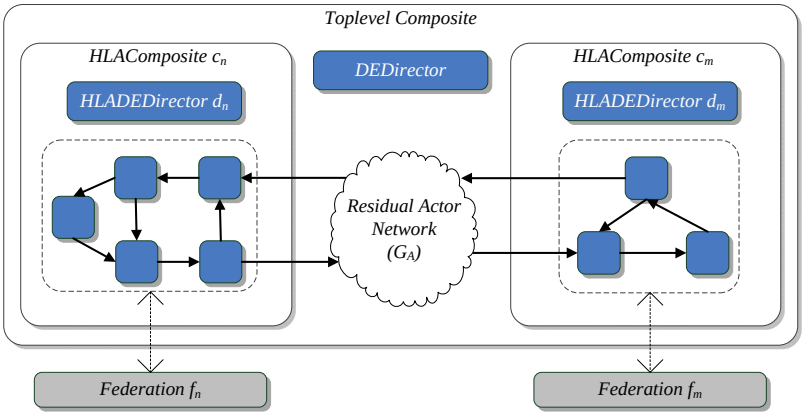


Abbildung 5.11.: Beispiel für eine Multi-Federation

Auf dieser Basis können folgende Fälle für den DAG G_A der DE Domäne (vgl. Abschnitt 2.3.4.4) und Paare c_n und c_m von *HLAComposite* Actors auf dem Toplevel unterschieden werden:

1. G_A **ist schleifenfrei**: Es wird angenommen, dass nur Pfade von c_n in Richtung c_m existieren. c_n kann dann entweder über a) *verzögerungsfreie*[10] oder b) *verzögerungsbehaftete* Pfade Tokens an c_m senden.

   a) **Nur verzögerungsfreie Pfade von c_n nach c_m**: Aufgrund der Schleifen- und Verzögerungsfreiheit zwischen c_n und c_m liegt c_n in G_A garantiert näher an einer Wurzel als c_m. Daher gilt $\lambda_n < \lambda_m$. Wenn nun ein Ereignis e_n auftritt, das c_n bei (τ_n, μ_n) aktiviert und ein Ereignis e_m auftritt, das c_m bei (τ_m, μ_m) aktiviert und $\tau_n = \tau_m$ sowie $\mu_n = \mu_m$ ist, dann wird c_n wegen $\lambda_n < \lambda_m$ auch garantiert vor c_n ausgeführt. Damit erfolgt die Synchronisation von d_n mit f_n über die HLA Schnittstelle ebenfalls vor der Ausführung von c_m. Im Ergebnis befindet sich in der Eventqueue von PtII ein durch d_n erstelltes Pure Event e_n^{pure} mit $\tau_n^{pure} = \tau_n^{grant}$. Wenn d_m nun mit f_m synchronisiert, so wird das

[10] Dies beinhaltet Verzögerungsfreiheit bzgl. τ und bzgl. μ.

Limit τ_n^{grant} bei der Synchronisation bzw. beim Aufruf von *getModelNextIterationTime()* berücksichtigt, da sich dieses in Form von e_n^{pure} bereits in der Eventqueue befindet. d_m wird deswegen dem *setNextBarrier()* Aufruf und damit dem NERA Service niemals einen Zeitstempel τ_m^{req} übergeben, der größer ist als τ_n^{grant}. Es ist somit garantiert, dass $\tau_m^{grant} \leq \tau_n^{grant}$ gilt und die Kausalität bleibt erhalten.

b) **Verzögerungsbehaftete Pfade von c_n nach c_m**: Beliebige Verzögerungen auf Pfaden von c_n nach c_m zerschneiden den DAG im Vergleich zum vorherigen Fall in Teilgraphen. Es ist dann in Abhängigkeit der Position von Quell- und Verzögerungsactors in der DAG Topologie nicht mehr garantiert, dass c_m im Fall von $\tau_n = \tau_m$ und $\mu_n = \mu_m$ nach c_n ausgeführt wird: Sollten c_n und c_m als Folge der Verzögerung zufällig den gleichen Level λ haben, so ist die Ausführungsreihenfolge nicht mehr definiert. U.U. könnte dann $\tau_m^{grant} > \tau_n^{grant}$ werden. Die Kausalität könnte damit potentiell verletzt werden[11].

Eine Lösung um sicherzustellen, dass weiterhin $\tau_m^{grant} \leq \tau_n^{grant}$ garantiert ist, ist die manuelle Vergabe von Prioritäten, so dass im Fall von gleichem τ, gleichem μ und gleichem λ der komposite Actor c_n in jedem Fall vor c_m ausgeführt wird. Solche Prioritäten können in PtII für jeden DE Actor separat gewählt werden (vgl. [217]).

2. G_A **ist nicht schleifenfrei**: Für bestimmte n und m können sich c_n und c_m u.U. gegenseitig Tokens senden. Um die durch die PtII DE Domäne festgelegten Regeln der erlaubten dynamischen Semantik nicht zu verletzen, muss innerhalb einer solchen Schleife in Summe eine minimale Verzögerung von einem Microstep vorhanden sein (vgl. Abschnitt 2.3.4.4). Im Fall einer Multi-Federation genügt dies nicht. Selbst Verzögerungen bzgl. τ reichen nicht aus. Vielmehr müssen weitere Randbedingungen erfüllt sein, um Kausalitätsfehler zu vermeiden.

   Angenommen, innerhalb der einzigen Schleife zwischen c_m und c_n existieren beliebige Verzögerungen bzgl. τ auf jedem Teilpfad und es treten Ereignisse e_n und e_m bei (τ_n, μ_n) und (τ_m, μ_m) auf mit $\tau_n = \tau_m$ und $\mu_n = \mu_m$. Aufgrund der DAG Topologie wird c_n vor c_m ausgeführt. Die Ursache dafür, dass trotz der Verzögerungen Kausalitätsverletzungen auftreten können ist, dass c_m nach der Synchronisation von d_n mit f_n und dem Einfügen eines Pure Events e_n^{pure} mit $\tau_n^{pure} = \tau_n^{grant}$ auf dem entgegengesetzten Pfad weiterhin Tokens verschicken kann, die vor τ_n^{grant} bei c_n eintreffen, obwohl

[11] Diese Kausalitätsverletzung würde allerdings von PtII erkannt werden. Die Ausführung würde dann in jedem Fall mit einer Exception abbrechen.

f_n schon bis τ_n^{grant} vorangeschritten ist. Bei umgekehrter Priorisierung gilt dasselbe reziprok für c_n.

Eine mögliche Lösung für dieses Dilemma basiert auf der regelmäßigen Generierung eines speziellen Pure Events e_{nm}^{sync} mit einem zeitlichen Abstand von $\Delta\tau_{nm}^{sync} > 0$, beispielsweise durch einen separaten *DiscreteClock* Actor, der höchste Priorität besitzt. e_{nm}^{sync} erzeugt dann innerhalb des PtII Modells regelmäßig eine Synchronisationsbarriere, die weder von d_n noch d_m bei der Synchronisation mit f_n bzw. f_m überschritten werden kann.

$\Delta\tau_{nm}^{sync}$ muss größer als Null sein, damit die Simulation nicht in einem Deadlock verharrt und eine unendliche Anzahl an Microsteps ausführt. Um Kausalitätsfehler zu vermeiden, muss $\Delta\tau_{nm}^{sync}$ außerdem gezielt nach oben hin beschränkt werden. Dies ist möglich für alle Schleifen, die in Summe mindestens eine zeitliche Verzögerung von $\Delta\tau > 0$ enthalten. Dabei sind folgende Fälle für die Pfadverzögerungen zwischen einem c_n und einem c_m zu unterscheiden:

a) $\Delta\tau_{nm} > 0$ und $\Delta\tau_{mn} > 0$: Die Kausalität zwischen c_n und c_m bleibt gewahrt, wenn $\Delta\tau^{sync} = \min(\Delta\tau_{nm}, \Delta\tau_{mn})$.

b) $\Delta\tau_{nm} > 0$, $\Delta\tau_{mn} = 0$ und $\Delta\mu_{mn} \geq 0$: Die Kausalität zwischen c_n und c_m bleibt gewahrt, wenn $\Delta\tau^{sync} = \Delta\tau_{nm}$ und $prio(c_m) > prio(c_n)$[12].

c) $\Delta\tau_{nm} = 0$, $\Delta\tau_{mn} = 0$: Die Kausalität kann mit der beschriebenen Methode nicht hergestellt werden.

Unter der Voraussetzung, dass Fall c) nicht existiert, müssen die Bedingungen aus den Fällen a) und b) für alle Pfade zwischen allen Paaren c_n und c_m mit $n, m \in N$ und $n \neq m$ erfüllt sein, damit die Multi-Federation insgesamt kausal korrekt ausführbar ist. Dazu kann beispielsweise ein globales Pure Event mit dem kleinsten im Gesamtmodell vorhandenen $\Delta\tau^{sync}$ generiert werden. Eine andere Variante ist es, für jede existierende Schleife ein separates Pure Event e^{sync} zu generieren. In einem HLA Director müssen dann nur die Pure Events berücksichtigt werden, die durch Schleifen mit dem eigenen kompositen Actor ausgelöst werden.

Die Ableitung eines $\Delta\tau^{sync} > 0$ ist vergleichbar mit der Ableitung eines Lookaheads größer Null und der Vermeidung von Deadlocks durch Vermeidung von kritischen Zyklen beim Null Message Algorithmus (vgl. Abschnitt 4.4.3.3). Die beschriebene Methode ist als Alternative zur Methode der Priorisierung auch im schleifenfreien Fall anwendbar, führt aber evtl. zu konservativeren Synchronisationsintervallen.

[12] Falls $\Delta\mu_{mn} = 0$, so ist die korrekte Priorisierung wegen der topologischen Sortierung des DAG bereits automatisch hergestellt. Andernfalls muss die Priorisierung manuell vergeben werden.

5.6. Fallstudie I: System/Netzwerk Co-Simulation

Werkzeuge zur Modellierung und Simulation von Kommunikationsnetzwerken erlauben die Spezifikation von Protokollen entsprechend dem ISO/OSI Referenzmodell [147]. Typischerweise wird entlang des zu modellierenden Kommunikationsprotokolls an unterschiedlichen Stellen abstrahiert, um einen geeigneten Trade-off zwischen Performanz und notwendiger Genauigkeit zu erzielen.

Bei den weit verbreiteten diskreten ereignisbasierten Simulatoren wie z.B. OMNeT++ [258], ns-2 [66], dessen Nachfolger ns-3 [132] oder OPNET [80] können unterschiedlich komplexe Modelle von Schichten des Protokollstacks nach Bedarf zusammengesetzt werden. Entsprechend [263] sind solche Netzwerksimulatoren allerdings eher für die Untersuchung von Algorithmen und Kommunikationsprotokollen geeignet und weniger für Funktionstests und Performanzanalysen von HW/SW Systemen (siehe auch Abb. 5.12). Die Granularität der zwischen Kommunikationsendpunkten ausgetauschten Dateneinheiten bewegt sich typischerweise auf der Ebene von Nachrichten und Paketen. Daraus ergibt sich eine zeitliche Granularität auf der Ebene von Protokolltransaktionen. Wegen ihrer Effizienz eignen sie sich somit insbesondere auch als Generator von Testpatterns für detailliertere Teilmodelle von HW/SW Systemen.

	HW model complexity	SW model complexity	HW performance metrics	Speed	Note
Cycle accurate	●●●●●	●●●●●	●●●●●	○○○○○	High realism
Instruction accurate	●●●●○	●●●●●	●●●●○	●●○○○	Testing functionality
Virtual Processing Units	●●●○○	●○○○○	●●●○○	●●●○○	Design space exploration
Simulation Instrumentation	●○○○○	●●●●○	●●○○○	●●●●○	Timing
Network Simulation	○○○○○	●○○○○	○○○○○	●●●●●	Algorithms, Protocols

Abbildung 5.12.: Trade-offs zwischen der Simulation von Netzwerken und Hardware/Software (Quelle: [263])

5.6.1. Beispiel: OMNeT++

Die Grundlage des frei verfügbaren Netzwerksimulators OMNeT++ [258] bildet ein diskreter ereignisbasierter Kernel, dessen Berechnungsmodell vergleichbar mit der des SystemC Kernels oder der DE Domäne von PtII ist. Der markanteste Unterschied ist die Tatsache, dass der OMNeT++ Kernel keine Deltacycles bzw. Microsteps kennt. Mit Hilfe sog. Frameworks wie z.B. INET oder MiXiM [165] kann der Kernel für verschiedene Anwendungsgebiete wie die Simulation von Internet Protokoll Stacks oder Wireless Sensor Networks (WSNs) spezialisiert werden. Diese Frameworks sind in der Hauptsache Bibliotheken, die bereits validierte Simulationsmodelle zur Verfügung stellen.

Ein Modell in OMNeT++ besteht ähnlich wie ein PtII Modell aus hierarchisch geschachtelten Modulen. Die Art und Weise der Schachtelung repräsentiert die logische Struktur eines Netzwerks bestehend aus Knoten und virtuellen drahtgebundenen oder drahtlosen Netzwerkkanälen. Jeder Knoten implementiert ein funktionales Modell der simulierten Netzwerkprotokolle in Form von Protokoll-Zustandsmaschinen. Diese beschreiben das Verhalten beim Senden und Empfangen von Paketen sowie die Art und Weise der Manipulation von Datenstrukturen. Einzelne Protokollschichten lassen sich modular kombinieren. Darüber hinaus ist es im Allgemeinen möglich, äußere Umgebungseinflüsse wie z.B. Mobilität oder Hindernisse bis zu einem gewissen Grad zu modellieren. Typische beobachtbare Parameter sind Datendurchsatz, Übertragungslatenzen, Paketverluste oder Übertragungsfehler.

5.6.2. Konzept

Um die grundlegende Funktionsfähigkeit einer HLA basierten Co-Simulation zu demonstrieren, wurde eine Single-Federation zur Simulation vernetzter Multiprozessorsysteme entwickelt. Das den implementierten Szenarien zugrundeliegende Konzept ist in Abb. 5.13 dargestellt.

Die Gesamtsimulation besteht aus einem Netzwerkmodell und mehreren detaillierten MPSoC Modellen. Das Netzwerkmodell besteht wiederum aus mehreren abstrakten Knotenmodellen, die ein bestimmtes Kommunikationsprotokoll simulieren. In Abhängigkeit von einem konkreten Anwendungsfall können ausgewählte abstrakte Netzwerknoten mit detaillierten MPSoC Modellen verfeinert werden. Dies resultiert in einer System/Netzwerk Co-Simulation zwischen dem Netzwerkmodell und den MPSoC Modellen. Ein Anwendungsfall für die Verfeinerung eines Knotens ist beispielsweise die Verifikation unterschiedlicher Knotenkonfigurationen hinsichtlich der Erfüllung von Performanzanforderungen, die zur Ausführung einer Anwendung notwendig sind.

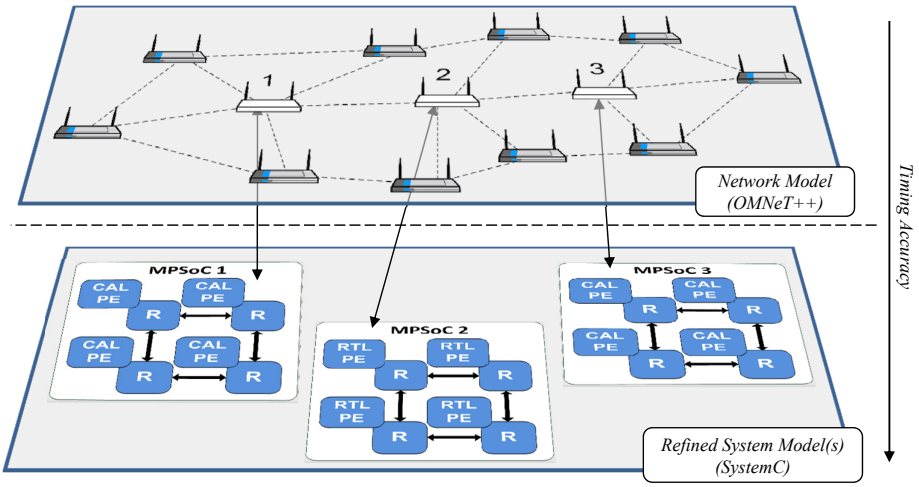


Abbildung 5.13.: Konzept der System/Netzwerk Co-Simulation

5.6.3. Single-Federation

Die entwickelte *Networked MPSoC Federation* besteht aus einem *Network Federate* und ein oder mehreren *MPSoC Federate*s. Abb. 5.14 zeigt ein entsprechendes Federationmodell in PtII Syntax, welches die existierenden HLA Objektklassen und Datenflüsse zwischen den Federates über die Objektklassen illustriert. Das *Network Federate* implementiert das Netzwerkmodell, die *MPSoC Federate*s verfeinerte Systemmodelle. Die Federates kommunizieren über zwei Object Classes namens *upMsg* und *downMsg*. Das *Network Federate* publiziert *upMsg* und abonniert *downMsg* Object Classes. Die *MPSoC Federate*s publizieren *downMsg* und abonnieren *upMsg*Object Classes. Die Attribute der Object Classes sind in den Sprechblasen dargestellt. Da PtII selbst nicht Teil der Federation ist, existieren im Federationmodell keine externen Ports zum umgebenden PtII Modell.

5.6.3.1. Network Federate

Das Network Federate wurde mit Hilfe von OMNeT++/MiXiM implementiert. Die Methode der Integration des HLA Interface Wrappers ist in weiten Teilen äquivalent zur Integrationsmethode in die DE Domäne von PtII (vgl. Abschnitt 5.5.3):

Die BIM FSM des *HLADEDirectors* von PtII kann vollständig wiederverwendet werden. Die Kontrollflussmethoden können, vergleichbar zu Integration in den

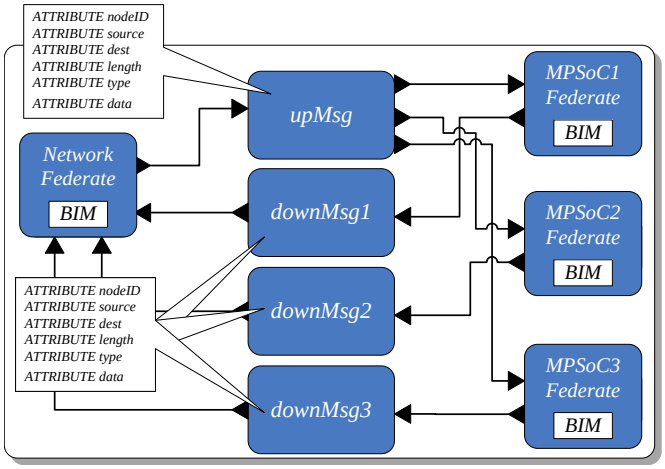


Abbildung 5.14.: Modell der *Networked MPSoC Federation*

HLADEDirector, mit Hilfe eines vom OMNeT++ Standardscheduler abgeleiteten Schedulers in den Kernel integriert werden.

Vergleichbar mit der Integration der Datenflussmethoden in den *HLADEDirector* und die *HLADEObjectClass* Actors, können die Datenflussmethoden im Fall vom OMNeT++ in den neuen Scheduler und die Knoten des Netzwerkmodells integriert werden. Die Knoten eines Netzwerkmodells werden dazu in lokale Knoten und verteilte Knoten eingeteilt (siehe Abb. 5.15).

Lokale Knoten besitzen ausschließlich eine Repräsentation innerhalb des Netzwerkmodells. Sie implementieren den kompletten Protokollstack im Netzwerkmodell. Verteilte Knoten besitzen zusätzlich eine Repräsentation in Form eines Systemmodells. sie implementieren nur untere Protokollschichten im Netzwerkmodell. Protokolle höherer Schichten werden in einem verfeinerten Systemmodell implementiert. Die Schnittkante zwischen oberen und unteren Protokollschichten ist prinzipiell frei wählbar. Im konkreten Fall befindet sie sich zwischen dem Data Link Layer und dem Network Layer. Welche Knoten lokale und welche verteilte Knoten sind, kann durch einen Parameter in OMNeT++ eingestellt werden.

Vor dem beschriebenen Hintergrund erklären sich unmittelbar die Namen der Object Classes im FM aus Abb. 5.14: Eine *upMsg* / *downMsg* ist eine Nachricht von einer niedrigeren / höheren zu einer höheren /niedrigeren Protokollschicht.

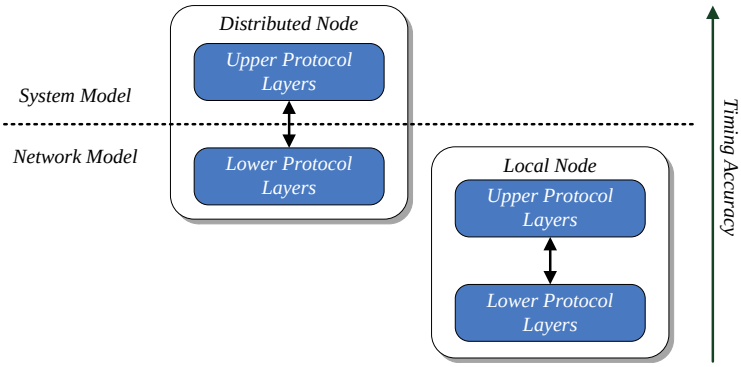


Abbildung 5.15.: Verteilte und lokale Netzwerkknoten

5.6.3.2. MPSoC Federate

Die MPSoC Federates wurden mit Hilfe des SystemC Kernels und einer Erweiterung des bereits bekannten HeMPS Modell (Wireless HeMPS) implementiert. Die Beschreibung einer Methode zur HLA Schnittstellenintegration ähnlich zur PtII und OMNeT++ Integration findet sich in [RMR+12]. Die hier beschriebene Methode unterscheidet sich insofern vom Ansatz aus [RMR+12], als dass auf eine Integration von Kontroll- und Datenflussmethoden des HLA Interface Wrappers in den SystemC Kernel verzichtet wurde. Vielmehr integriert die verfolgte Variante die Methoden vollständig auf Modellebene. Diese sog. Non-Intrusive Lösung reduziert die Komplexität der Implementierung. Der Ansatz ist vergleichbar mit der Methode zur Synchronisation auf Modellebene, die bereits im Kontext der parallelen SystemC/TLM Simulation in Abschnitt 4.6 verwendet wurde. Er ist insbesondere dann anwendbar, wenn nur eine einzige Modellinstanz (ein HeMPS Modell) innerhalb des MPSoC Federates existiert.

Ähnlich zu [RMR+12] existiert in Abb. 5.16) in einem der PEs des HeMPS Modells eine virtuelle drahtlose Netzwerkschnittstelle genannt *Virtual Wireless Interface (VWI)*. Das VWI ist über ein Registerinterface an das PE angebunden und in den Adressbereich des Plasmakerns integriert. Es interagiert sowohl über die Kontroll- als auch die Datenflussmethoden mit dem HLA Interface Wrapper. Zum Nachrichtenaustausch existieren Eingangs- und Ausgangspuffer. Der Zugriff auf die Puffer wird vom Plasma Prozessor über einen Kontrollblock gesteuert.

Da das VWI das einzige Modul ist, das mit dem HLA Interface Wrapper interagiert, kann die zeitliche Synchronisation zwischen SystemC Kernel und HLA Interface Wrapper indirekt über *wait(time)* Aufrufe in einem *SC_THREAD* Prozess

des Kontrollblocks erfolgen. Der HLA Interface Wrapper selbst basiert weiterhin auf der bereits aus PtII bekannten BIM FSM des *HLADEDirectors*.

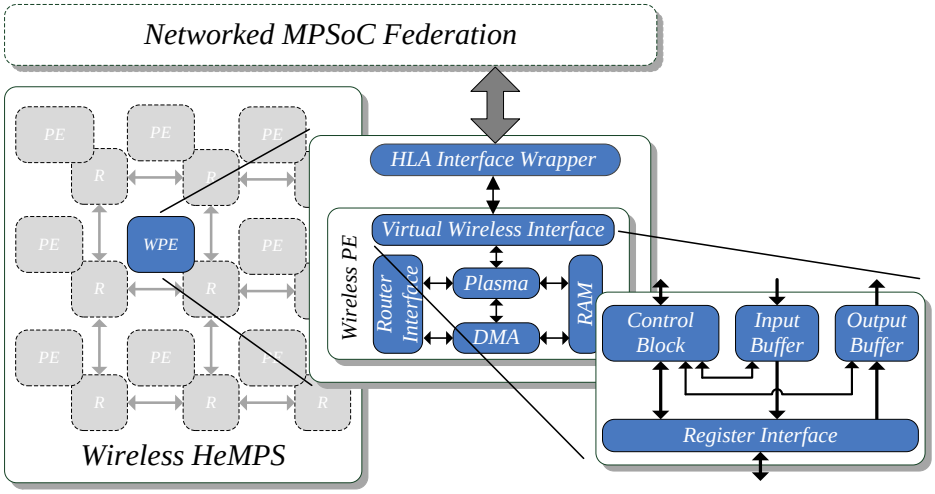


Abbildung 5.16.: MPSoC Federate

5.6.4. Szenario I: Performanzanalyse für venetzte MPSoCs

Das erste Testszenario besteht aus einer Anzahl an Knoten, die in einem Kreis mit einem Radius von 10 Metern angeordnet sind. In der Mitte des Kreises befindet sich ein zusätzlicher Knoten, der alle anderen Knoten im Kreis mit Datenpaketen unter Verwendung unterschiedlicher Übertragungsraten stimuliert. Das Übertragungsmedium ist ein IEEE 802.11b Kanal mit 11 Mbps mit einem simulierten Paketverlust von 1%. Ein Knoten aus dem Kreis ist als verteilter Knoten konfiguriert, alle anderen als lokale Knoten.

Die im MPSoC Federate des verteilten Knotens verwendeten Knotenkonfigurationen des HeMPS Modells führen allesamt bei 100 MHz aus. Sie sind durch die Anzahl an PEs und die Art der Abbildung von Softwaretasks auf die PEs charakterisiert. Die Kombination aus beidem resultiert in unterschiedlichen Verarbeitungszeiten und Beschränkungen für den Durchsatz. Da das IEEE 802.11b MAC Protokoll keine Sicherungsmechanismen zur Verfügung stellt, verursachen unterschiedliche Durchsatzbeschränkungen unterschiedliche Paketverlustraten im VWI, insbesondere dann, wenn vorhandene Daten langsamer verarbeitet werden als neue Daten nachkommen.

Die Struktur der auf den MPSoCs ausgeführten Software ist in Abb. 5.17 dargestellt. Sie besteht grundsätzlich aus sechs Tasks. Der *NET* Task bildet zusammen mit dem VWI das Network Layer. Alle restlichen Tasks gehören zum Application Layer. Die Tasks bilden eine Pipelinestruktur. Als konkrete Applikationen werden eine Dummyapplikation und eine Erweiterung der bereits bekannten MPEG Applikation verwendet.

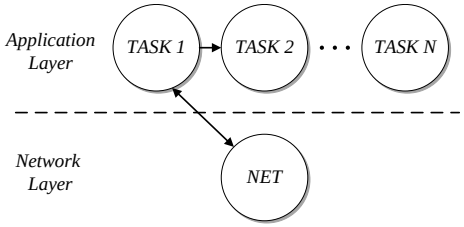


Abbildung 5.17.: Struktur der Beispielanwendungen

Die Application Layer Tasks empfangen Datenpakete vom *NET* Task und verarbeiten diese. Im Fall der Dummyapplikation wird jedem Netzwerkpaket eine Rechenkomplexität w in Form von ganzahligen Multiplikationen zugewiesen. Jeder Task der Applikation verarbeitet $\frac{1}{5} \times w$ Operationen. Im Fall der MPEG Applikation übertragen Nachrichten eine konfigurierbare Anzahl an 8x8 Blöcken eines MPEG Streams. Die MPEG Applikation selbst ist prinzipiell identisch zur MPEG Pipeline aus Abschnitt 4.4.9.

Abb. 5.18 illustriert die verwendeten Konfigurationen von HeMPS und zugehörige Abbildungen von Tasks. Entsprechend ihrer Größe und der Taskabbildung sind sie mit *1x2*, *2x2a* *2x2b* und *3x3* bezeichnet. Das VWI befindet sich immer an genau dem PE, auf dem der *NET* Task ausgeführt wird. Der mit *M* markierte Knoten ist der Master, auf dem keine Tasks ausgeführt werden können.

Die Abb. 5.19 und 5.20 illustrieren die durch die verschiedenen Konfigurationen entstandenen und gemessenen prozentualen Paketverluste im VWI für die Dummy und die MPEG Applikation (nicht zu verwechseln mit den 1% Paketverlust, die immer durch den drahtlosen Kanal entstehen).

Wie zu erkennen ist, hat die Konfiguration von Hardware und Software starken Einfluss auf die Höhe des Paketverlusts. Im Fall einer Senderate von 500 Pkt/s über den drahtlosen Kanal ist das 1x2 System in der Lage, Pakete mit einer Komplexität von 500 Ganzzahlmultiplikationen ohne Verlust zu verarbeiten. Im Gegensatz dazu kann das 3x3 System Pakete mit einer Komplexität von bis zu 4000 Operationen verlustfrei verarbeiten. Eine Verdopplung der Paketrate resultiert erwartungsgemäß in einer Linksverschiebung des Diagramms.

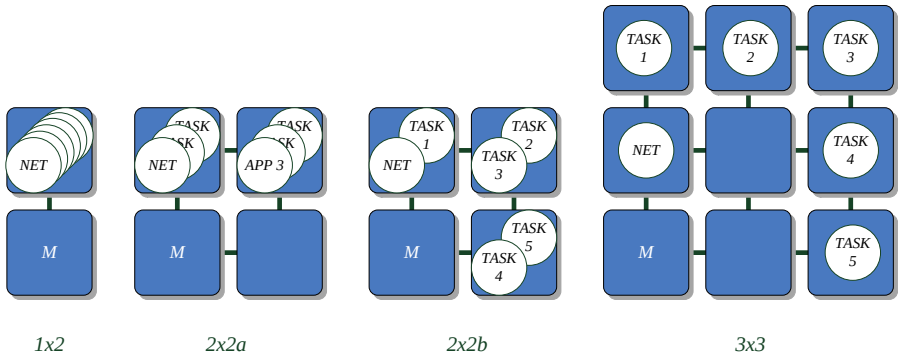
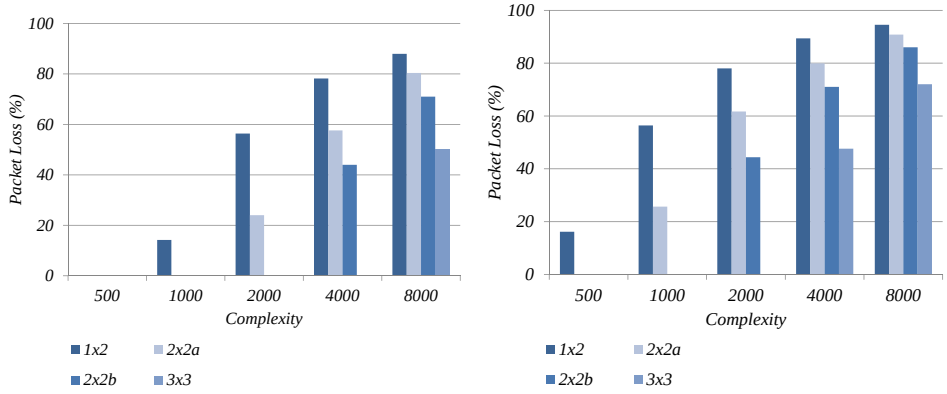


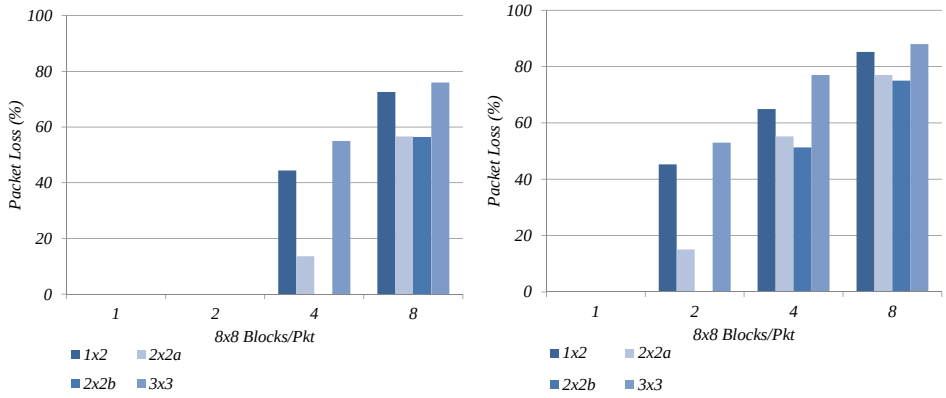
Abbildung 5.18.: Verwendete Konfigurationen von Wireless HeMPS

Im Fall der MPEG Applikation ist offensichtlich Variante 2x2b die am besten geeignete, da sie bis zu 4 MPEG Blöcke pro Paket bei 500 Pkt/s und bis zu 2 Blöcke pro Paket bei 1000 Pkt/s verarbeiten kann. Der beim 3x3 System zusätzlich entstehende interne Kommunikationsoverhead über das NoC ist offensichtlich zu hoch, um die MPEG Verarbeitung weiter zu beschleunigen. Dies hat einen Anstieg des Verlusts im Vergleich zum 2x2b System von 19.6% bei 500 Pkt/s und 13% bei 1000 Pkt/s zur Folge.



(a) 500 Pkt/s (b) 1000 Pkt/s

Abbildung 5.19.: Dummy Applikation



(a) 500 Pkt/s (b) 1000 Pkt/s

Abbildung 5.20.: MPEG Applikation

5.6.5. Szenario II: Verteilte Ausführung

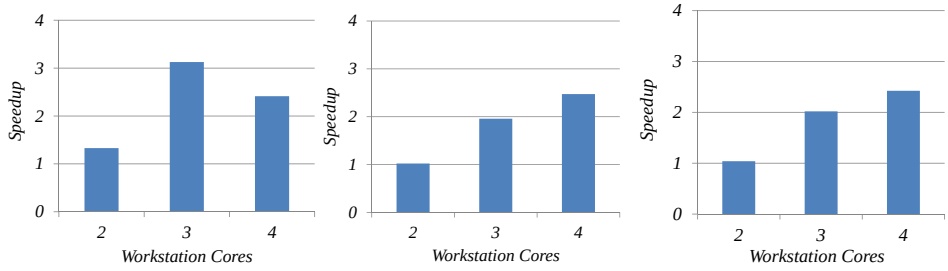
Die folgende Untersuchung soll zeigen, dass die verteilte Ausführung von Federates auch zur Performanzsteigerung genutzt werden kann. Durch die Verteilung der Federates wird die grobgranulare Parallelität zwischen mehreren disziplinspezifischen Teilmodellen genutzt: Weder das Netzwerkmodell noch ein MPSoC Modell selbst werden parallel ausgeführt. Die Kombination mit den Ansätzen aus Kapitel 4 ist ein möglicher Ansatzpunkt für zukünftige Arbeiten.

Die Untersuchung basiert auf einem Szenario bestehend aus 64 Knoten, die in einer Meshstruktur angeordnet sind und über einen IEEE 802.11b Kanal kommunizieren. Jeder Knoten sendet Datenpakete per Broadcast mit einer Frequenz von 100 Hz. Die Knoten in der Diagonalen des Meshs sind verteilte Knoten, alle anderen sind lokale Knoten. Lokale Knoten implementieren die Broadcast Applikation mit Hilfe eines simplen Traffic Generators auf dem Application Layer. Im HeMPS System wurde wieder die Task Pipeline aus Abb. 5.17 verwendet: *TASK 5* generiert Pakete. Alle anderen schieben diese weiter bis zum *NET* Task.

Die Messungen wurden auf einer SHM Workstation mit einer 2.0 GHz Quadcore CPU, 8 GB RAM und unter Verwendung von SystemC 2.2.0, OMNeT 4.1 und CERTI 3.2 durchgeführt. Bei zwei, drei und vier Workstation Kernen wurde das Network Federate immer separat auf einem Kern ausgeführt. Die erzielten parallelen Beschleunigungen im Vergleich zu sequentieller Ausführung aller Federates auf einem einzigen Kern sind in Abb. 5.21 dargestellt.

Die Beschleunigung auf zwei Kernen ist wegen der ungleichen Lastverteilung zwischen Netzwerksimulation und MPSoC Simulationen im Allgemeinen stark

beschränkt (teilweise $< 1x$): Der Hauptanteil des Rechenaufwandes wird, trotz der 56 lokalen Knoten, durch die MPSoC Modelle generiert. Zudem werden beim Übergang auf zwei Kerne die acht MPSoC Modelle noch nicht parallel ausgeführt, da ein Kern vollständig für die Netzwerksimulation reserviert ist. Die Beschleunigungen der 2x2 und 3x3 Simulationen fallen auf zwei Kernen generell niedriger aus als die der 1x2 Simulationen. Die vergleichsweise bessere Lastverteilung zwischen Netzwerksimulation und MPSoC Simulationen basierend auf dem 1x2 Modell ist eine Erklärung für diesen Effekt.



(a) 1x2 Modelle (b) 2x2 Modelle (c) 3x3 Modelle

Abbildung 5.21.: Gemessene Beschleunigung bei verteilter Ausführung

Trotz der ungleichen Lastverteilung zwischen Netzwerk und MPSoC Federates erreicht die Beschleunigung bei Verwendung des 1x2 Modells auf drei Kernen (d.h. nur zwei Kerne sind für die MPSoC Simulationen verfügbar) Werte $> 3x$. Ein Grund können Cacheeffekte innerhalb der Speicherhierarchie der verwendeten Workstation sein: Bei sequentieller Ausführung der MPSoC Simulationen übersteigt die Menge der zu speichernden Daten die Größe des verfügbaren schnellen Cachespeichers, was zu vielen langsamen Hauptspeicherzugriffen führt. Werden mehr Kerne für die MPSoC Simulationen verwendet, so ist insgesamt mehr schneller Cachespeicher zur Datenhaltung verfügbar. Unter Umständen können die Modelle dann vollständig im Cache gehalten werden. Die Anzahl langsamer Hauptspeicherzugriffe wird dadurch reduziert, was in einer zusätzlichen Beschleunigung resultiert. Bei den 2x2 und 3x3 Modellen ist dies nicht der Fall, da die Modelle offensichtlich so groß sind, dass ihre Größe die Kapazität des Caches übersteigt, auch bei drei Kernen.

Im Fall von insgesamt vier Kernen geht die Beschleunigung beim 1x2 Modell wieder auf $2.4x$ zurück. Die Beschleunigungen der 2x2 und 3x3 Modelle steigt an. Dabei wurden ähnliche Werte gemessen. Vernachlässigt man die Netzwerksimulation, so beträgt die theoretisch maximal mögliche Beschleunigung der MPSoC Simulationen $3x$. Eine Ursache weswegen diese bei allen drei MPSoC Modellen nicht erreicht wird, ist der durch den höheren Parallelisierungsgrad ent-

stehende höhere Synchronisationsaufwand in Verbindung mit der nun auch zwischen den MPSoC Simulationen vorhandenen ungleichen Lastverteilung (zwei Kerne führen drei MPSoC Modelle aus und ein Kern nur zwei). Auch mögliche positive Cacheeffekte reichen offensichtlich nicht aus, um eine ähnlich hohe Beschleunigung zu erzielen, wie zuvor beim 1x2 Modell auf drei Kernen.

5.7. Fallstudie II: Simulation von V2X basierten E/E Architekturen

Ausgehend von den in Abschnitt 5.1 beschriebenen Zusammenhängen ist die Abdeckung folgender drei Aspekte grundlegend für eine simulative Verifikation zukünftiger V2X-basierter E/E Architekturen: E/E Architektur, V2X Kommunikation und (restliche) physikalische Umwelt. Generell existieren Wechselwirkungen zwischen jedem der drei genannten Aspekte (vgl. Abb. 5.22). Mögliche und notwendige Detailgrade zur Modellierung der Aspekte und deren Wechselwirkungen werden im Folgenden diskutiert.

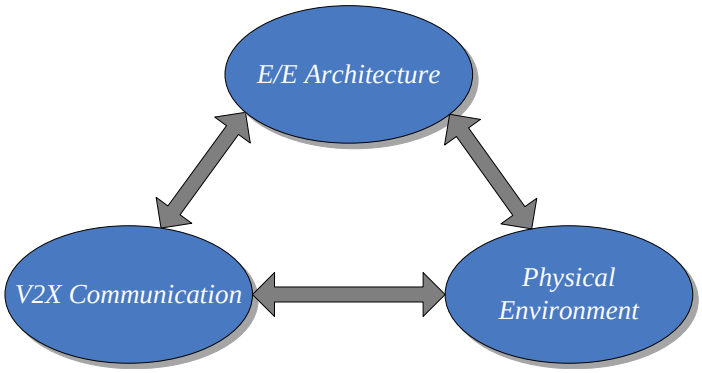


Abbildung 5.22.: Aspekte und Wechselwirkungen in einer Simulation für V2X-basierte E/E Architekturen

5.7.1. E/E Architektur

Entsprechend den Ausführungen in Abschnitt 5.1 erfolgt die Entwicklung zukünftiger E/E Architekturen optimalerweise unter Verwendung einer Meet-in-the-Middle Entwurfsmethodik wie dem PBD. Eine Voraussetzung für die Übertragung von PBD Prinzipien ist die Definition von geeigneten Abstraktionsebe-

nen für die Spezifikation der Funktion und die Spezifikation der Architektur. Dabei muss einerseits von den Details einer bestimmten Architekturimplementierung abstrahiert werden aber andererseits genügend Information vorhanden sein, um möglichst genaue Vorhersagen über Eigenschaften von Funktionen und Architektur wie z.B. notwendige/verfügbare Berechnungs- oder Kommunikationsperformanz zu erlauben [229].

Die Selektion der E/E Architektur ist die Hauptaufgabe eines OEM. Vincentelli und Di Natale schlagen für die Integration dieser Aufgabe mit einer PBD Entwurfsmethodik in [229] eine Granularität der Funktionsspezifikation auf der Ebene einzelner Teilfunktionen und der E/E Architekturspezifikation auf der Ebene von ECUs und Bussen vor. Einzelne funktionale und architektonische Teilkomponenten müssen bis zu einem gewissen Grad parametrierbar sein. Für eine Exploration relevante Parameter von Teilkomponenten müssen von dem jeweiligen Verantwortlichen (z.B. dem OEM oder einem Zulieferer) bereitgestellt werden, der diese Teilkomponenten entwickelt. Typische Analysen auf der Ebene der Gesamtarchitektur beinhalten z.B. (vgl. [229]) Evaluationen von Ende-zu-Ende Latenzen und Schedulability oder System-Level Simulationen von Funktion und Timing.

Eine detailliertere Exploration einzelner Teilkomponenten sowie deren funktionale Verifikation setzt eine Modellierung der Teilkomponenten mit feinerer Granularität voraus. Im Verlauf des Entwicklungsprozesses müssen von den Verantwortlichen daher akkuratere Modelle von Teilsystemen wie Funktionen, Betriebssystemkomponenten, ECUs oder Bussen bereitgestellt werden, welche für eine durchgängige Verifikation bis hin zur Implementierung notwendig sind. Die Gesamtarchitektur kann dann mit diesen Modellen schrittweise verfeinert werden.

Beispielsweise muss die Granularität von ECUs u.U. auf die Ebene einzelner Kerne aufgebrochen werden, um die Abbildung von Funktionen mit der notwendigen Genauigkeit untersuchen zu können. Dies beinhaltet z.B. die Verifikation des Schedulings und der Ausführungsdauer auf einzelnen Kernen oder der entstehenden Kommunikationslatenzen zwischen den Kernen unter Verwendung exakterer zyklenapproximativer Modelle.

5.7.2. V2X Kommunikation und physikalische Umwelt

Durch V2X Kommunikation wird die E/E Architektur über eine Funkschnittstelle „geöffnet“ (vgl. Abschnitt 5.1.1). Funktionen, die auf der E/E Architektur ausgeführt werden sind u.U. in hohem Maße von Informationen abhängig, die über den drahtlosen Kanal eintreffen. Umgekehrt sendet das eigene Fahrzeug über den Funkkanal Informationen an andere Fahrzeuge.

V2X Kommunikation erzeugt neue Wechselwirkungen des Fahrzeugs, welche in einem hohen Grad nicht deterministisch sind. Dies betrifft sowohl die Menge an zu verarbeitenden Nachrichten, die in einem bestimmten Zeitraum eintreffen, als auch deren Inhalt. Die beiden Haupteinflussparameter sind der drahtlose Kommunikationskanal in Kombination mit dem Verhalten der umgebenden Verkehrs.

Zur Simulation des drahtlosen Kommunikationskanals eignen sich paketbasierte Netzwerksimulatoren. Dies wurde bereits in Fallstudie I demonstriert. Im Fall von V2X Kommunikation muss in erster Linie zusätzlich das Bewegungsverhalten des Straßenverkehrs modelliert und simuliert werden.

5.7.2.1. Modellierung und Simulation von Straßenverkehr

Unter Verkehrssimulation versteht man die Simulation von Straßenverkehr auf Basis mathematischer Modelle (vgl. [254]). Dies beinhaltet typischerweise die Modellierung von Straßen, Kreuzungen, Routen etc. sowie die Modellierung des dynamischen Verhaltens des Verkehrs oder einzelner Fahrzeuge. Die typischerweise in Verkehrssimulatoren implementierten Bewegungs- oder Mobilitätsmodelle lassen sich inhaltlich in drei Klassen einteilen:

- **Submikroskopische Modelle** haben einen sehr hohen Detailgrad. Sie beschreiben die Längs- und Querdynamik eines einzelnen Fahrzeugs mit Hilfe komplexer Differentialgleichungen. Die Gleichungen werden für jeden Simulationsschritt neu berechnet. Der Rechenaufwand ist daher extrem hoch. Ein sub-mikroskopisches Bewegungsmodell findet sich beispielsweise in der Software CarMaker [6] der Firma IPG.
- **Mikroskopische Modelle** beschreiben das Verhalten mehrerer Fahrzeuge. Im Unterschied zu sub-mikroskopischen Modellen, werden Bewegungsparameter eines einzelnen Fahrzeugs meist in Abhängigkeit benachbarter Fahrzeuge bestimmt (sog. Fahrzeugfolgemodelle). Der Detailgrad eines simulierten Fahrzeugs ist um einiges geringer als bei submikroskopischen Modellen. Die Verhaltensmodellierung beschränkt sich oft auf die Längsdynamik. Mikroskopische Modelle sind meistens diskret. Beispiele sind die Verkehrssimulatoren SUMO [166] oder PTV Vissim [13].

  In SUMO werden z.B. Fahrzeugfolgemodelle verwendet, welche das Verhalten bzw. den Zustand für jedes einzelne Fahrzeug separat mit Hilfe von kontinuierlichen Differentialgleichungen berechnen. Zur Ausführung dieser Modelle implementiert der Kernel von SUMO ein Berechnungsmodell (vgl. Abschnitt 2.2.3.3), bei dem sich das Zeitinkrement im Mikrosekundenbis Sekundenbereich variieren lässt. Insgesamt ist die Simulation räumlich kontinuierlich aber zeitdiskret.

- **Makroskopische Modelle** basieren auf der Verwendung von Statistik und Anleihen aus der Strömungslehre zur Verkehrsflussmodellierung. Typische Parameter sind Dichte und Geschwindigkeit. Da keine einzelnen Fahrzeuginstanzen mehr vorhanden sind, reduziert sich die Anzahl zu berechnender Ereignisse um ein Vielfaches im Vergleich zu mikroskopischen Modellen. Makroskopische Modelle werden typischerweise für große innerstädtische Szenarien verwendet, bei denen eine Analyse oder Prognose der Verkehrsbedingungen im Vordergrund steht. PTV Visum [14] ist ein Beispiel eines makroskopischen Simulators.

Neben den genannten Klassen existieren auch noch gemichte Ansätze (sog. mesoskopische Modelle), welche mikroskopische und makroskopische Modellierungsmethoden verknüpfen [254]. Im hier betrachteten Kontext sind Verkehrsmodelle der ersten beiden Kategorien relevant. Aufgrund einer ähnlichen zeitlichen Auflösung eignen sich insbesondere mikroskopische Modelle für die Kopplung mit einer paketbasierten Netzwerksimulation. Je stärker der Schwerpunkt auf einer Betrachtung der durch V2X beeinflussten Fahrdynamik liegt, desto mehr ist die Verwendung submikroskopischer Bewegungsmodelle notwendig.

5.7.2.2. Bidirektionale Kopplung von Netzwerk- und Verkehrssimulation

Die Notwendigkeit für eine bidirektional gekoppelte Simulation von Netzwerkkommunikation und Straßenverkehr wurde von Sommer et al. in [239] aufgezeigt. Einer der in [239] genannten Hauptgründe dafür ist die Tatsache, dass V2X Kommunikationsprotokolle ohne ein entsprechendes Feedback eines Modells des Straßenverkehrs nicht ausreichend getestet werden können. Umgekehrt erlaubt eine bidirektionale Kopplung auch die Untersuchung des Einflusses der Netzwerkkommunikation auf den Straßenverkehr.

Eine solche Kopplung ist im Simulationswerkzeug Veins [239] anhand von OMNeT++ und SUMO umgesetzt (siehe Abb. 5.23). Das Ziel dieser Kopplung ist die Untersuchung von V2X Kommunikationsprotokollen zur dynamischen Routenplanung oder von sicherheitsrelevanten Applikationen. Die Kopplung zwischen OMNeT++ und SUMO basiert auf TCP/IP.

Veins stellt verschiedene Modelle der einschlägigen V2X Protokolle wie IEEE 1609.x und 802.11p [30][21] basierend auf dem MiXiM Framework zur Verfügung. SUMO ist als zusätzliches Mobilitätsmodell in MiXiM integriert. Dadurch ist eine weitaus akkuratere Modellierung und Simulation von Straßenverkehr möglich, als mit den in MiXiM standardmäßig vorhandenen Mobilitätsmodellen. Zudem bietet SUMO die Möglichkeit, Straßenkarten diverser Formate zu importieren.

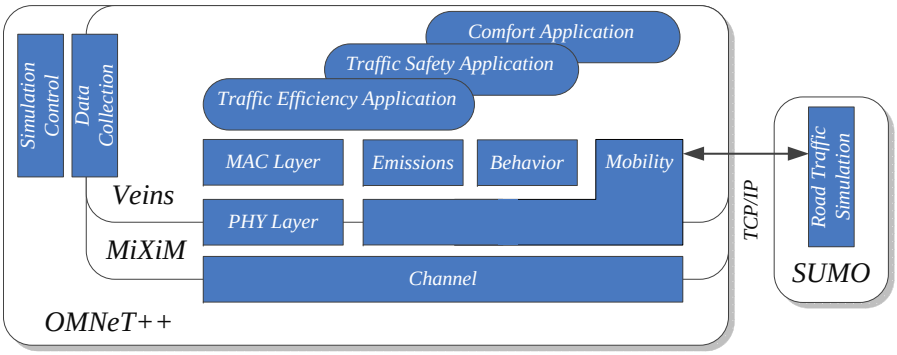


Abbildung 5.23.: Architektur des Veins Frameworks (Quelle: [238])

Die Argumentation aus [239] ist unmittelbar auf den Anwendungsfall der E/E Architektursimulation übertragbar: Aus Sicht von [239] werden der Kommunikations- und der Verkehrsaspekt ein weiteres Mal erweitert und zwar um den E/E Architekturaspekt, mit dem Ziel, zusätzlich Aussagen über die Ausführungsperformanz von V2X Applikationen und Funktionen und deren Wechselwirkungen mit dem drahtlosen Kommunikationskanal machen zu können. Die durch Veins zur Verfügung gestellte Granularität von Daten und der Zeit reicht dabei aus, um Stimuli und Testbenches für die Verifikation von Funktionen zu generieren, die auf der E/E Architektur ausgeführt werden.

5.7.2.3. Modellierung und Simulation der restlichen physikalischen Umwelt

Unter diesen Aspekt fallen alle Komponenten, welche nicht durch die Simulation der V2X Kommunikation und die Simulation des Straßenverkehrs mit Veins erfasst werden. Dazu gehören vor allem auch Interaktionen mit der restlichen umgebenden physikalischen Umwelt über Sensor/Aktuator Schnittstellen der E/E Architektur. Soll beispielsweise zusätzlich die Wechselwirkung der Räder eines DbW Systems, wie es in Abschnitt 5.1.1 beschrieben ist, mit der Straße berücksichtigt werden, so müssen Sensoren, Aktuatoren und die mechanische Interaktion der Räder mit der Straße anhand von Differentialgleichungen modelliert werden. Für die Modellierung anhand von Differentialgleichungen eignen sich Werkzeuge wie Matlab/Simulink [8] oder das bereits im Kontext der submikroskopischen Verkehrssimulation erwähnte Werkzeug CarMaker [6]. Auch die CT Domäne von PtII ist dafür geeignet.

5.7.3. Multi-Federation

Ausgehend von den zuvor beschriebenen Anforderungen wurde eine Multi-Federation entwickelt, welche es ermöglicht, Wechselwirkungen innerhalb eines Systems von vernetzten V2XC-fähigen Fahrzeugen in einer verteilten Simulation zu untersuchen. Als Beispielanwendung zur Ausführung auf der E/E Architektur und zur Demonstration der Funktionsfähigkeit des Konzepts dient ein V2X basierter Abstandsregeltempomat (Adaptive Cruise Control (ACC)).

Die Multi-Federation besteht aus einem PtII Federate, einem Veins Federate und einem HeMPS Federate. Die beiden disziplinspezifischen Simulatoren Veins und HeMPS werden in jeweils separaten Federations mit PtII co-simuliert. Die Komponenten der Multi-Federation sind in Abb. 5.24 dargestellt und werden im Folgenden erläutert.

5.7.3.1. Gesamtsystemmodell

PtII ist der Ausgangspunkt für die Modellierung des Gesamtsystems. Es wird angenommen, dass das zu entwickelnde System ein einzelnes Fahrzeug ist. Damit bildet die Fahrzeugkarosserie die Grenze zwischen dem zu entwickelnden System und der Systemumwelt.

Aus einer Top-down Perspektive besteht das Gesamtsystemmodell in PtII daher aus einem internen Fahrzeugmodell und einem Umgebungsmodell. Ersteres wird innerhalb eines kompositen Actors namens *Intra Vehicle Composite* modelliert, Letzteres in einem kompositen Actor namens *Inter Vehicle Composite*. Beide Actors kommunizieren über Ports und Relations. Interaktionen zwischen beiden Actors unterliegen dem DE Berechnungsmodell. Sofern Actors niedrigerer Hierarchieebenen keinen eigenen Director besitzen, folgen sie automatisch auch dem DE Berechnungsmodell.

Da ein spezieller Fokus auf der E/E Architektur des Fahrzeugs liegt, repräsentiert das interne Fahrzeugmodell aktuell alleine die E/E Architektur. Das komplette interne Fahrzeugmodell oder einzelne Komponenten (Actors) können beliebig verfeinert werden. Mit dem neuen Actor vom Typ *HLAComposite* können für die Verfeinerung auch sprachfremde Modelle verwendet werden. Auch das Umgebungsmodell kann durch beliebige (komposite) Actors bis auf die notwendige Modellierungstiefe weiter verfeinert werden (siehe Abb. 5.24).

5.7.3.2. Internes Fahrzeugmodell

Als Grundlage für die Entwicklung des internen Fahrzeugmodells dienten die Konzepte für zukünftige E/E Architekturen, die bereits in Abschnitt 5.1.1 disku-

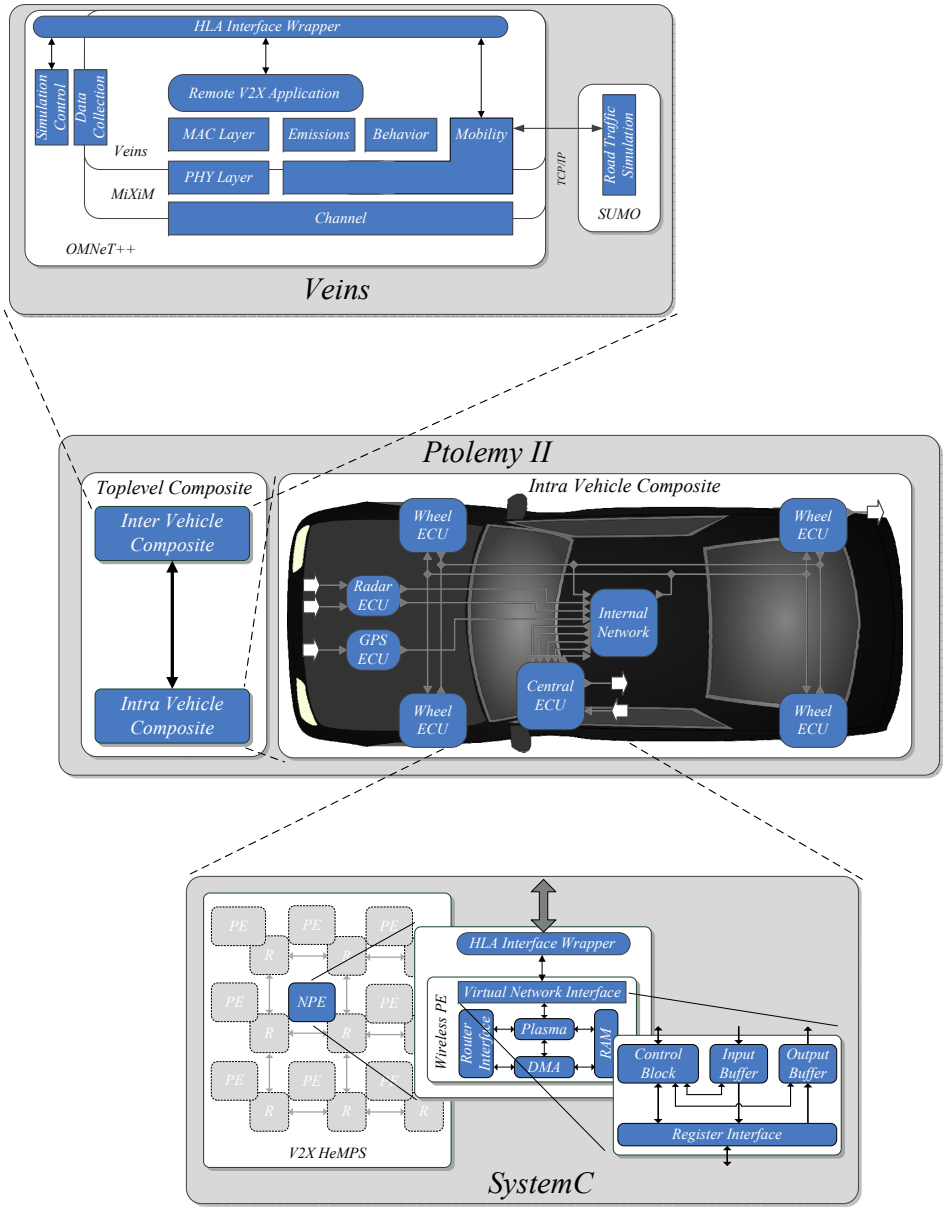


Abbildung 5.24.: Multi-Federation zur Simulation von E/E Architekturen

tiert wurden. Das daraus abgeleitete abstrakte Grundgerüst einer zukünftigen E/E Architektur ist in Abb. 5.24 oben dargestellt. Es wird angenommen, dass die Struktur der Architektur im Großen und Ganzen festgelegt ist, einzelne Komponenten bzw. Actors aber ausgetauscht oder bis zu einem gewissen Grad variiert und verfeinert werden können.

In Abb. 5.24 ist jede ECU und das interne Netzwerk durch einen separaten Actor modelliert. Es existiert eine zentrale Multicore ECU mit Namen *Central ECU*. Für jedes Rad existiert ein Actor namens *Wheel ECU*[13]. Daneben gibt es noch eine *Radar ECU* und eine *GPS ECU*. Es wird angenommen, dass alle ECUs über einen standardisierten fahrzeuginternen Kommunikationsbackbone (modelliert durch einen Actor namens *Internal Network*) Nachrichten austauschen können.

Central ECU

In Anlehnung an das in [227] beschriebene Konzept für V2XC-fähige Automotive Gateways, existiert mit der *Central ECU* ein Steuergerät, welches die Schnittstelle zwischen dem internen Bordnetz und dem externen drahtlosen Kommmunikationsnetzwerk bildet. Der entsprechende Actor in Abb. 5.24 verfügt deswegen über separate Ports für die V2X Kommunikation mit dem Umgebungsmodell.

Für die *Central ECU* existiert neben einem reinen kompositen PtII Actor auch ein Actor vom Typ *HLAComposite*, über den das interne Fahrzeugmodell in PtII mit einer Erweiterung des bereits bekannten *Wireless HeMPS* SystemC Modells aus Abschnitt 5.6 verfeinert werden kann. Das diesem Actor zugrundeliegende Federationmodell ist in Abb. 5.25 dargestellt. Die Actors mit Namen *wifiUpMsg* und *wifiDownMsg* sind vom Typ *HLADEObjectClass* und erfüllen den gleichen Zweck wie die *upMsg* und *downMsg* Actors aus der Fallstudie in Abschnitt 5.6. Zusätzlich dazu existieren zwei Actors / Object Classes mit Namen *internalGetMsg* und *internalSetMsg*, mit deren Hilfe das Verhalten der *Wheel ECU* gesteuert werden kann.

Auf HeMPS Seite wurde das *Virtual Wireless Interface* Modul zu einem allgemeinen *Virtual Network Interface* (VNI) erweitert, so dass sowohl V2X Datenpakete über das externe V2X Netzwerk (via *wifiUpMsg* und *wifiDownMsg*), als auch Datenpakete über das interne Netzwerk (via *internalGetMsg* und *internalSetMsg*) empfangen und versendet werden können.

Wheel ECU

Entsprechend dem in Abschnitt 5.1.1 beschriebenen DbW Ansatz, ist eine *Wheel ECU* für Vorverarbeitung von Sensordaten und die Radansteuerung verantwortlich. Um die Fallstudie zu vereinfachen, wurde o.B.d.A. nur eine abstrakte *Wheel*

[13] In der hier beschriebenen Fallstudie war nur eine der Wheel ECUs aktiv.

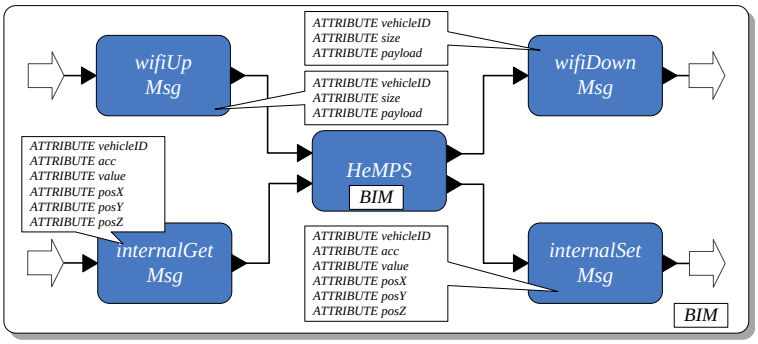


Abbildung 5.25.: Modell der HeMPS Federation

ECU umgesetzt. Die Implementierung basiert auf einem CT Director. In Kombination mit dem DE Director wird damit die Möglichkeit zur heterogenen Co-Simulation demonstriert.

Als Vorgabe erhält der *Wheel ECU* Actor einen Beschleunigungswert aus dem internen Bordnetz (*Internal Network* Actor) und erzeugt daraus durch Integration eine Geschwindigkeit für das interne Fahrzeugmodell. Über einen externen Port wird die erzeugte Geschwindigkeit der Repräsentation des internen Fahrzeugmodells im Umgebungsmodell mitgeteilt. Über die *Wheel ECU* könnten auch weitaus komplexere Verhaltenseigenschaften des Fahrzeugs oder Einflussgrößen der Fahrbahn modelliert werden. Auch die Verfeinerung der *Wheel ECU*s mit einem vollständigen submikroskopischen Dynamikmodell eines Fahrzeugs (beispielsweise anhand von CarMaker [RSG+10]) wäre denkbar.

Radar und GPS ECUs

Neben dem mit dem *Wheel ECU* Actor implizit modellierten Geschwindigkeitssensor sind der *Radar ECU* und der *GPS ECU* Actor die einzigen in der Fallstudie explizit modellierten Sensorkomponenten. Die *Radar ECU* liefert die Position und die Geschwindigkeit des vorausfahrenden Fahrzeugs. Die *GPS ECU* liefert die Position des eigenen Fahrzeugs. Aktuell werden die genannten Werte durch das Umgebungsmodell berechnet. Aus diesem Grund dienen die beiden Steuergerätemodelle ausschließlich als reine Stellvertreterobjekte und reichen die durch das Umgebungsmodell berechneten Werte an den *Internal Network* Actor einfach durch.

5.7.3.3. Umgebungsmodell

Das Umgebungsmodell im *Inter Vehicle Composite* Actor wird vollständig durch Veins repräsentiert. Veins simuliert den umgebenden Straßenverkehr und die drahtlose Netzwerkkommunikation zwischen Fahrzeugen. Der *Inter Vehicle Composite* Actor wurde dazu mit Hilfe eines Actors vom Typ *HLAComposite* umgesetzt. Das Federationmodell im *Inter Vehicle Composite* Actor ist in Abb. 5.26 dargestellt. Im Prinzip ist das Federationmodell der Veins Federation reziprok zum Federationmodell der HeMPS Federation definiert: Im Unterschied zum HeMPS Federate empfängt das Veins Federate von PtII Aktualisierungen für *wifiUpMsg* und *internalGetMsg* Object Classes und sendet Aktualisierungen für die *wifiDownMsg* und *internalSetMsg* Object Classes.

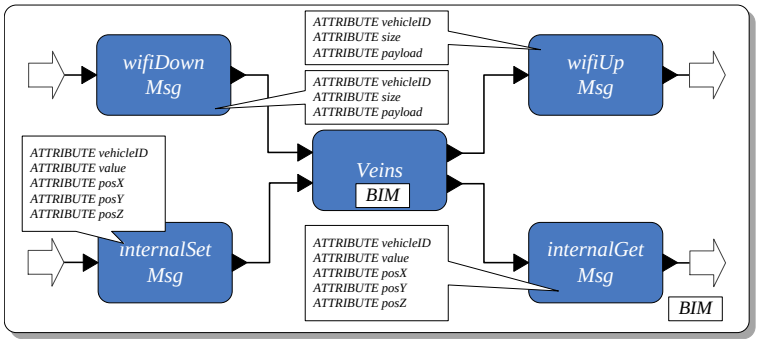


Abbildung 5.26.: Modell der Veins Federation

Die Integration in Veins entspricht weitgehend der Implementierung des OMNeT++ basierten Network Federates aus der ersten Fallstudie. Ein in PtII modelliertes internes Fahrzeugmodell hat eine Repräsentation in Form eines verteilten Knotens innerhalb von Veins. Im Unterschied zu Abschnitt 5.6 werden für diesen verteilten Knoten nicht nur Netzwerkpakete des Network Layer und des MAC Layer zwischen PtII und Veins übertragen, sondern auch Bewegungsinformationen zum verteilten Knoten, die über die *internalGetMsg* Actors / Object Class Instanzen in die Sensormodelle des internen Fahrzeugmodells eingespeist werden. Umgekehrt können Bewegungsparameter wie z.B. die Geschwindigkeit des verteilten Knotens durch Aktualisierung einer *internalSetMsg* Object Class Instanz überschrieben werden. Wie bei der OMNeT++ Integration ist einstellbar, ob und welche Fahrzeuge unter der Kontrolle von PtII oder Veins sind. Grundsätzlich ist es auch möglich, die abstrakte Verhaltenssimulation mehrerer Fahrzeuge in Veins durch eine detailliertere Simulation in PtII zu ersetzen.

5.7.4. Szenario I: Test einer ACC Funktion

Um die Funktionsfähigkeit der heterogenen Co-Simulation zwischen PtII und Veins zu demonstrieren, wurde der *Central ECU* Actor zunächst nicht mit dem HeMPS Modell verfeinert. Stattdessen wurde die ACC Funktion im kompositen *Central ECU* Actor auf algorithmischer Ebene (ohne Berücksichtigung von Architekturcharakteristika) und unter Verwendung des DE und CT Berechnungsmodells umgesetzt.

5.7.4.1. Intelligent Driver Model

Die auf der *Central ECU* implementierte Funktion beschränkt sich auf eine ACC Implementierung basierend auf dem sog. *Intelligent Driver Model (IDM)* [253]. Das IDM ist ein kollisionsfreies mikroskopisches Fahrzeugfolgemodell, das auch als Grundlage für eine echte ACC Implementierung dient [162, 159, 167]. Das IDM ist eine kontinuierliche Funktion der tatsächlichen Geschwindigkeit v, des Abstands zum vorausfahrenden Fahrzeug s und der Geschwindigkeitsdifferenz Δv:

$$a_{IDM}(v, s, \Delta v) = \frac{dv}{dt} = a \left[1 - \left(\frac{v}{v_0} \right)^{\delta} - \left(\frac{s^*(v, \Delta v)}{s} \right)^2 \right], \tag{5.1}$$

$$s^*(v, \Delta v) = s_0 + vT + \frac{v\Delta v}{2\sqrt{ab}}. \tag{5.2}$$

Dabei ist s^* der gewünschte minimale Abstand. Die Bedeutung der übrigen Modellparameter kann Tab. 5.1 entnommen werden. Hier sind auch typische Parametrierungen illustriert. In [254] werden die Modellparameter anhand der folgenden drei Standardsituationen veranschaulicht:

- **Beschleunigen auf freier Strecke**: Dies geschieht zunächst mit der maximalen Beschleunigung bei normalem Verkehr a. Die Beschleinigung geht bei Annäherung an die Wunschgeschwindigkeit v_0 in einer durch den Parameter δ beschriebenen Weise gegen Null. Je größer δ gewählt wird, desto später reduziert sich die Beschleunigung.
- **Folgefahren**: Das Folgefahren geschieht mit dem durch die Folgezeit T charakterisierten Abstand zuzüglich dem Minimalabstand s_0 bei stehendem Verkehr.
- **Annähern**: Bei der Annäherung an langsamere Fahrzeuge wird in Normalsituationen die komfortable Verzögerung b nicht überschritten.

Parameter	Autobahn	Stadtverkehr	Sinnvoll
Wunschgeschwindigkeit v_0	120 km/h	54 km/h	50 - 200 km/h
Folgezeit T	1.0 s	1.0 s	0.9 - 3 s
Minimalabstand s_0	2 m	2 m	1 - 5 m
Beschleunigungsexponent δ	4	4	1 - ∞
Beschleunigung a	1.0 m/s^2	1.0 m/s^2	0.3 - 3 m/s^2
Komfortable Verzögerung b	1.5 m/s^2	1.5 m/s^2	0.5 - 3 m/s^2

Tabelle 5.1.: Modellparameter des IDM (Quellen: [254, 160])

5.7.4.2. Funktionsblöcke

Zur Einbettung des IDM in den *Central ECU* Actor wurden mehrere Funktionsblöcke in Form von kompositen Actors implementiert. Das zu verwendende Übertragungsmedium, um Informationen über das vorausfahrende Fahrzeug zu erhalten (*Radar ECU* oder *V2XC Kanal*), kann innerhalb der *Central ECU* durch Vertauschung der Portanbindung gewechselt werden. Folgende Funktionsblöcke existieren (wenn nicht anders angegeben, wurde das DE Berechnungsmodell verwendet):

- **Message Decoder (nur V2X Kanal)**: Ein über den V2XC Kanal empfangenes Paket wird vom Message Decoder Block in seine Bestandteile (Header und Payload) zerlegt. Die einzelnen Datenfelder werden als PtII Tokens an den *Message Filter* weitergeleitet.
- **Message Filter (nur V2X Kanal)**: Dieser Funktionsblock filtert relevante V2X Pakete aus dem eingehenden Strom von Paketen aus. Ein Paket wird als relevant klassifiziert, wenn es vom vorausfahrenden Fahrzeug stammt. Dieses wird anhand einer ID identifiziert. Alle anderen Pakete werden verworfen.
- **Environment Picture**: Dieser Funktionsblock dient als lokaler Zwischenspeicher eines aktuellen „Abbildes der Umgebung“ in Form von zuletzt empfangenen Umgebungsinformationen. Es speichert die über die *GPS ECU* und die *Wheel ECU* eingelesenen Werte für Position und Geschwindigkeit des eigenen Fahrzeugs sowie die entweder über die *Radar ECU* oder den V2X Funkkanal empfangenen Werte für Position und Geschwindigkeit des vorausfahrenden Fahrzeugs.
- **IDM Controller**: Der IDM Controller Block ist die eigentliche Implementierung des oben beschriebenen IDM. Auf Basis der Daten des *Environment Picture* Blocks gibt das IDM die Beschleunigung a für die *Wheel ECU* (Regelstrecke) vor. Dieser komposite Actor folgt dem CT Berechnungsmodell.

Mit der durch das IDM berechneten Beschleunigung a_{IDM} generiert die *Wheel ECU* die Geschwindigkeit v. Diese wiederum wird an die Repräsentation des Fahrzeugs in Veins (den verteilten Knoten) zurück übermittelt.

5.7.4.3. Dynamische Abstandsregelung via Radar und V2X

In Veins wird ein Verkehrsszenario mit zwei Fahrzeugen ausgeführt. Das vorausfahrende Fahrzeug wird von Veins gesteuert. Auf einer geraden Strecke in x-Richtung passiert es periodisch Straßenkreuzungen mit einem Abstand von 200 m (vgl. Abb. 5.27). Das nachfolgende Fahrzeug wird durch das interne Fahrzeugmodell bzw. die IDM Implementierung in PtII gesteuert. SUMO wurde mit einem Aktualisierungsintervall (Update Time in [161]) von 0.01 s ausgeführt. Die verwendeten IDM Parameter sind in Tab. 5.2 zusammengestellt. Zu Beginn der Simulation wurde Veins ein Vorlauf von 10 s Simulationszeit gegeben.

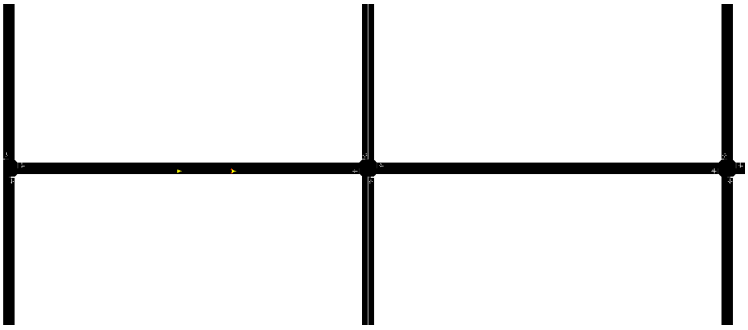


Abbildung 5.27.: Straßenkreuzungen mit einem Abstand von 200 m in Veins

Parameter	Gewählter Wert
Wunschgeschwindigkeit v_0	54 km/h
Folgezeit T	1.0 s
Minimalabstand s_0	2.0 m
Beschleunigungsexponent δ	4
Beschleunigung a	2.6 m/s^2
Komfortable Verzögerung b	3.0 m/s^2

Tabelle 5.2.: Gewählte IDM Modellparameter

Radar ACC

Abb. 5.28 zeigt die mit PtII aufgezeichneten Simulationsergebnisse für Geschwindigkeit, Beschleunigung, Abstand und Position bei Verwendung einer ACC Funktion basierend auf der *Radar ECU*. Innerhalb des nachfolgenden Fahrzeugs (im PtII E/E Architekturmodell sowie zur Synchronisation mit Veins) wurde, wie in SUMO, generell eine Aktualisierungsrate von 0.01 s verwendet. Die roten Kurven illustrieren das Verhalten des vorausfahrenden und vollständig in Veins simulierten Fahrzeugs, die blauen Kurven das Verhalten des nachfolgenden und teilweise in PtII simulierten Fahrzeugs. Am Plot der Geschwindigkeit ist sehr gut das regelmäßige Auftreten der Straßenkreuzungen zu erkennen: Das in Veins bzw. SUMO verwendete Mobilitätsmodell (eine Erweiterung des IDM) berücksichtigt u.a. die Rechts-vor-Links Regel. Dies resultiert in regelmäßigen Abbrems- und Beschleunigungszyklen des vorausfahrenden Fahrzeugs, sobald dieses eine Kreuzung passiert. Die Geschwindigkeit des vorausfahrenden Fahrzeugs sinkt periodisch von 13.9 m/s auf ca. 2.2 m/s ab und steigt dann wieder bis auf 13.9 m/s an. Dazwischen befinden sich Abschnitte von ca. 8 s Dauer, bei denen freie Fahrt mit konstanter Geschwindigkeit von 13.9 m/s möglich ist.

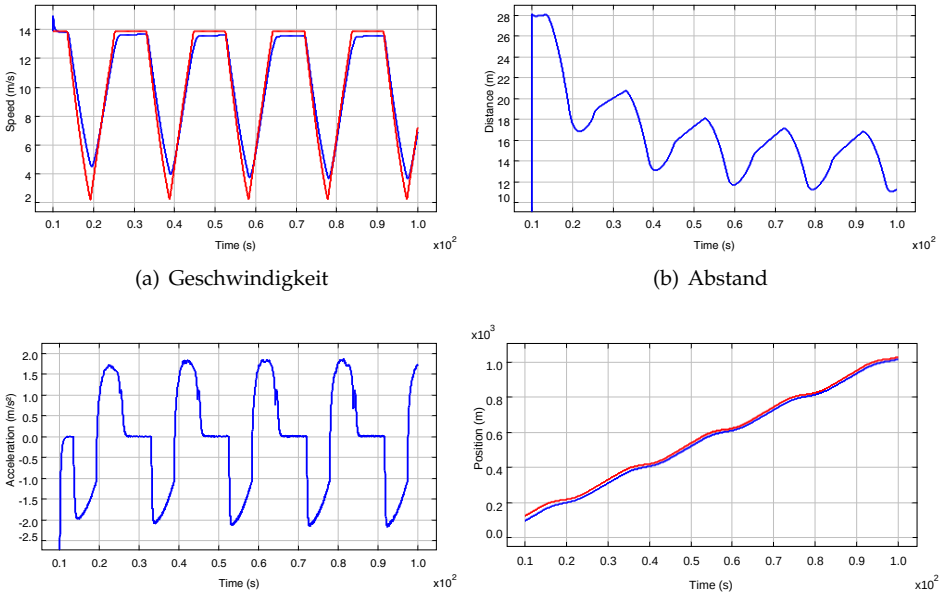
Aufgrund der in PtII implementierten Basisvariante des IDM, beachtet das nachfolgende Fahrzeug die Rechts-vor-Links Regel nicht. Es folgt ausschließlich dem Beschleunigungsverhalten des vorausfahrenden Fahrzeugs und hält dabei eine in einem gewissen Rahmen variable räumliche Distanz ein. Im gemessenen Zeitintervall von 90 s unterschreitet diese niemals den Wert von 11 m, so dass das vorausfahrende und das nachfolgende Fahrzeug nahezu parallele Trajektorien in x-Richtung besitzen (siehe Abb. 5.28 unten rechts. Über das gesamte Zeitintervall von 90 s ist im Schnitt dennoch ein leichtes Sinken der Distanz zu beobachten. Diese Verhalten ist auf einen Einschwingvorgang zu Beginn des Szenarios zurückzuführen, da das nachfolgende Fahrzeug mit einer größeren Distanz hinter dem vorausfahrenden Fahrzeug startet, als durch das durch die Summe aus Folgezeit T und dem Minimalabstand s_0 definierte Minimum vorgegeben ist.

V2X ACC

Um die Funktionsfähigkeit der Netzwerksimulation zu testen und zugleich den Einfluss verschiedener Sendeintervalle von V2X Nachrichten auf das ACC Verhalten zu untersuchen, wurde nicht mehr die Radar ECU, sondern der V2X Kommunikationskanal als Übertragungmedium von Umgebungsinformation und zur Stimulation der ACC Funktion verwendet. Durch Variation des Beaconintervalls der periodisch verschickten V2X Broadcastnachrichten (sog. *Cooperative Awareness Messages (CAM)* [30]) zwischen 0.1 *s*, 1.0 *s* und 10.0 *s* wurden unterschiedliche Sendeintervalle erzeugt. Gründe für höhere Beaconintervalle könnten beispielsweise ein überlasteter Funkkanal sein. Die Aktualisierungsraten innerhalb von SUMO, PtII und dazwischen betrugen weiterhin 0.01 s. Bei Beaconintervallen von 0.01 *s* und 0.1 *s* waren die Ergebnisse noch weitgehend identisch zu den Plots

des Radar ACC. Die Verzögerung durch den drahtlosen Kanal resultierte in einer leichten hochfrequenten Oszillation der Distanz. Die Plots von Geschwindigkeit und Position für 1.0 s und 10.0 s sind in Abb. 5.29 zu sehen.

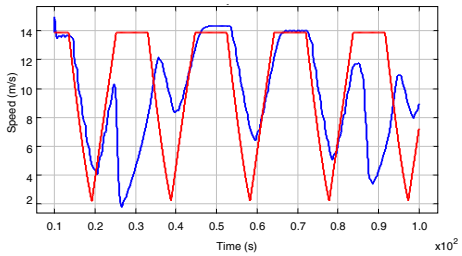
Es ist deutlich zu erkennen, dass der Verlauf der Geschwindigkeit des nachfolgenden Fahrzeugs im Vergleich zu Abb. 5.28 nicht mehr exakt dem des vorausfahrenden Fahrzeugs folgt. Die resultierenden Fahrmaneuver lassen sich insbesondere zu Beginn des Szenarios durch größere Ausschläge in der minimalen und maximalen Geschwindigkeit charakterisieren. Größere Sendeintervalle resultieren zudem in einer zunehmend veränderten Trajektorie des nachfolgenden Fahrzeugs sowie in einem größeren Abstand der Trajektorien von vorausfahrendem und nachfolgendem Fahrzeug. Da bei den betrachteten Simulationen der Betrag der maximal möglichen Bremsbeschleunigung nicht auf weniger als $9m/s^2$ limitiert war und außer der Latenz der drahtlosen Kanals keine weiteren Latenzen modelliert wurden, sind keine Kollisionen zwischen den Fahrzeugen aufgetreten. Kollisionsfreiheit ist eine grundlegende Eigenschaft des IDM [253].



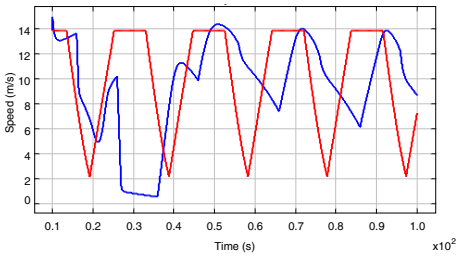
(a) Geschwindigkeit (b) Abstand

(c) Beschleunigung (d) Position

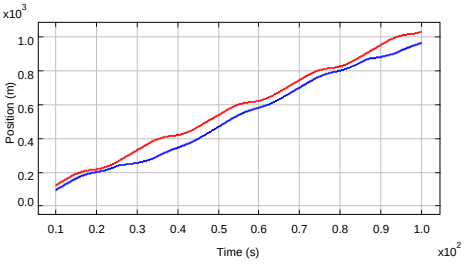
Abbildung 5.28.: Messwerte mit Radar ACC, rot = vorausfahrendes Fahrzeug, blau = nachfolgendes Fahrzeug



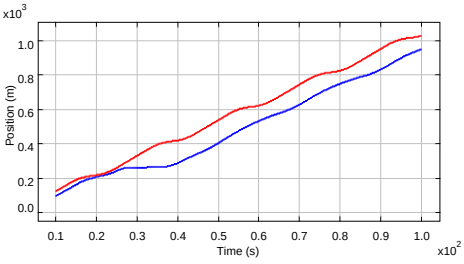
(a) Geschwindigkeit (1.0 s Beacons)



(b) Geschwindigkeit (10.0 s Beacons)



(c) Position (1.0 s Beacons)



(d) Position (10.0 s Beacons)

Abbildung 5.29.: Messwerte mit V2X ACC, rot = vorausfahrendes Fahrzeug, blau = nachfolgendes Fahrzeug

5.7.5. Szenario II: Verifikation einer ACC Implementierung

Mit dem zweiten Szenario soll die Funktionsfähigkeit einer Co-Simulation basierend auf mehreren Federations (Multi-Federation Modus) demonstriert werden. Dazu wurde die abstrakte Implementierung der V2X ACC Funktion im *Central ECU* Actor aus Szenario I durch eine Implementierung auf Basis von HeMPS ersetzt und mit Hilfe der HeMPS Federation co-simuliert. Die ACC Funktion wurde dazu als Taskgraph auf das HeMPS System portiert. Das Resultat ist eine verfeinerte Spezifikation der Implementierung der ACC Funktion, welche Charakteristika einer möglichen MPSoC Architektur der *Central ECU* berücksichtigt. Im Folgenden werden die einzelnen umgesetzten Teilschritte zunächst erläutert. Anschließend werden Messergebnisse vorgestellt und diskutiert.

5.7.5.1. Verfeinerung der Central ECU mit HeMPS

Zur Portierung der ACC Funktion auf HeMPS wurden die in Abschnitt 5.7.4.2 beschriebenen PtII Funktionsblöcke zunächst manuell in C-Code übersetzt. Jeder Funktionsblock wurde dazu als ein eigenständiger Softwaretask für das HeMPS OS implementiert. Zusätzlich wurden folgende Softwaretasks speziell zur Kommunikation mit dem VNI entwickelt:

- **Sensor Read**: Liest in regelmäßigen Abständen alle über das VNI eintreffenden Sensordaten ein und leitet sie an den *Environment Picture* Task weiter.
- **Actuator Write**: Wenn eine NoC-Nachricht vom *IDM Controller* Task empfangen wird, so wird der mit der Nachricht übermittelte Beschleunigungswert über das VNI an die *Wheel ECU* übetragen.
- **V2X Receive**: Sobald ein V2X Paket über das VNI empfangen wurde, so wird dies ausgelesen und an den *Message Decoder* Task weitergeleitet.
- **V2X Send**: Sendet den aktuellen Fahrzeugzustand per V2X an andere Fahrzeuge.

In Summe existieren damit acht Tasks. Diese wurden auf einem 2x2 HeMPS Modell verteilt. Da das VNI nur an ein einziges PE angebunden ist (vgl. Abb. 5.24 unten), mussten alle vier zusätzlich implementierten Tasks auf ein und denselben Kern abgebildet werden.

5.7.5.2. Integration der Multi-Federation

Zur Integration der Multi-Federation wurde zunächst der bisher verwendete *Central ECU* Actor durch einen Actor vom Typ *HLAComposite* ersetzt. Dieser ent-

hält das HeMPS Federationmodell. Um den Erhalt der Kausalität zu garantieren, wurde die in Abschnitt 5.5.3.3 beschriebene Methode zur Ableitung eines statischen Lookaheads im PtII Modell angewendet. Eine Analyse der Modellstruktur ergab folgende Modellcharakteristika:

Zwischen *Inter Vehicle Model* und *Central ECU* existiert eine Schleife. Alle vom *Inter Vehicle Model* zur *Central ECU* verlaufenden Pfade sind vollständig verzögerungsfrei. Von der *Central ECU* in Richtung *Inter Vehicle Model* existieren zwei Pfade, einer verläuft über die *Wheel ECU* und einer koppelt *Central ECU* und *Inter Vehicle Model* direkt (V2XC Kanal). Die *Wheel ECU* aktualisiert die Geschwindigkeit mit einem Intervall von $0.01s$. Auf dem Pfad zum V2X Kanal existiert eine Verzögerung von nur einem Microstep.

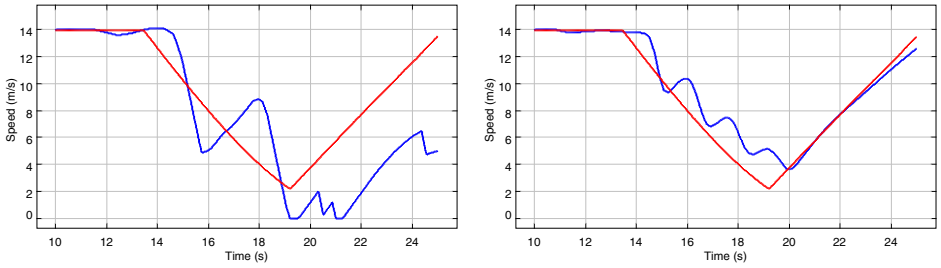
Würde von *Central ECU* zu *Inter Vehicle Model* nur der Pfad über die *Wheel ECU* existieren, könnte mit Bedingung 2b) aus Abschnitt 5.5.3.3 ein für das ganze PtII Modell gültiges $\Delta\tau^{sync}$ von $0.01s$ gewählt werden. Da die *Wheel ECU* dieses selbstständig generiert, wäre zudem keine zusätzliche Generierung von Pure Events z.B. anhand eines *DiscreteClock* Actors notwendig.

Wegen des zweiten Pfades für den V2X Kanal ist eine Multi-Federation allerdings nicht unmittelbar möglich. Besagter Pfad verursacht eine Schleife mit einer Verzögerung von $\Delta\tau = 0$. Entsprechend Bedingung 2c) aus Abschnitt 5.5.3.3 kann also keine Kausalität hergestellt werden. Die gewählte Lösung besteht darin, die Schleife durch gezieltes Einfügen einer zeitlichen Verzögerung in den zweiten Pfad aufzubrechen. Die Wahl der Verzögerung war dabei bis zu einem gewissen Grad beliebig, da die ACC Funktion im betrachteten Szenario durch Nachrichten, die vom eignen Fahrzeug verschickt werden, aktuell nicht beeinflusst wird. Um Bedingung 2a) für beide Pfade zu erfüllen und gleichzeitig den Synchronisationsoverhead mit dem co-simulierten *Inter Vehicle Model* nicht unnötig zu vergrößern, sind alle Verzögerungen mit Werten $\geq 0.01s$ geeignet.

5.7.5.3. Ergebnisse

Abb. 5.30 illustriert die Messergebnisse für ein simuliertes Zeitintervall von 15 s und 10 s Vorlauf für den Fall, dass die radarbasierte ACC Applikation auf einem zyklenapproximativen HeMPS Modell (CAL PEs und zeitliche Entkopplung der NoC Router um acht Takte) ausgeführt wird und das Fahrzeug seinen Zustand regelmäßig über den V2X Kanal ausgibt. Das Modell wurde mit Taktfrequenzen von 50 kHz und 100 kHz simuliert. Wie man der Abbildung entnehmen kann, reagiert die 100 kHz Version etwas schneller als die 50 kHz Variante. Dies resultiert darin, dass die Geschwindigkeit des eigenen Fahrzeugs bereits bei 20 s wieder ansteigt, während die 50 kHz Variante mit viel größerer Amplitude oszilliert, vollständig auf 0 $\frac{m}{s}$ zurückgeht und erst bei ca. 21 s wieder richtig mit dem Anstieg beginnt. Aufgrund des verhältnismäßig hohen Berechnungsaufwandes

für das HeMPS Modell, dauerte die detaillierte Simulation von 15 s um einige Größenordnungen länger als ohne HeMPS. Auf einem Core i5 Dual-Core Rechner mit 2.5 GHz wurden für die 50 kHz Variante ca. 11 min und für die 100 kHz Variante ca. 22 min Ausführungszeit gemessen. Ohne HeMPS lag die Ausführungszeit bei ca. 18 s.



(a) 50 kHz (b) 100 kHz

Abbildung 5.30.: V2X ACC als Applikation auf HeMPS, rot = vorausfahrendes Fahrzeug, blau = nachfolgendes Fahrzeug

5.7.6. Spezifikation von Funktionsabbildungen durch Aspekte

In den vorangegangenen Fallstudien wurde gezeigt, dass der vorgeschlagene Ansatz eine heterogene Modellierung und Co-Simulation sowie eine strukturierte Komposition von Teilmodellen erlaubt. Teilmodelle können Teilsysteme im Sinne von funktionalen oder architektonischen Artefakten des zu entwickelnden Systems oder äußere Umwelteinflüsse repräsentieren und simulieren. Es wurde demonstriert, dass die hierarchische Struktur von PtII Modellen direkt zur Verfeinerung von Teilmodellen mit detaillierteren (co-simulierten) Modellen bis hin zur finalen Implementierung genutzt werden kann. Die genannten Punkte sind notwendige Voraussetzung für eine umfassende interdisziplinäre Analyse und Verifikation des zu entwickelnden Systems.

Entsprechend der in Abschnitt 5.2 identifizierten Anforderungen, ist eine Simulationsumgebung für die Anwendung in einem Entwicklungsprozess für zukünftige E/E Architekturen insbesondere dann geeignet, wenn sie neben den bereits genannten Punkten auch weitergehende Konzepte eines plattformbasierten Entwurfsprozesses unterstützt. Ein solcher zeichnet sich insbesondere durch hohe Flexibilität bei den Möglichkeiten zur Exploration des Entwurfsraumes aus. Neben der reinen Verfeinerung von abstrakten Funktionen ist ein weiteres wichtiges Kriterium daher die Möglichkeit zur flexiblen Abbildung von Funktions-

komponenten auf Architekturkomponenten und deren anschließende abstrakte Simulation (vgl. Abschnitt 2.1.1.3).

Um den Einfluss einer Funktion-zu-Architektur Abbildungen bereits auf hohen Abstraktionsgraden flexibel modellieren zu können, existieren in PtII spezielle Actors vom Typ *Aspect* [217], welche einen kompositen Actor um die notwendige Funktionalität erweitern. Das einem Aspekt zugrundeliegende Prinzip entspricht dem eines Quantity Managers aus dem METRO II Framework [95]. Ein Aspekt ermöglicht die Spezifikation einer Abbildung mehrerer Funktionen auf ein und dieselbe Architekturkomponente und die Untersuchung des Einflusses eines daraus resultierenden sequentiellen Schedulings der Funktionen. Die Änderung einer Abbildung kann dabei ohne die Änderung der Verdrahtung von Actors erfolgen. Eine kleine Fallstudie, in der PtII Aspects zur Entwurfsraumexploration im Kontext des V2X ACC genutzt werden, ist in [RBB+14] zu finden.

5.8. Einordnung in verwandte Arbeiten und Fazit

Ein allgemeiner Überblick über den Stand der Technik zur heterogenen Co-Simulation wurden bereits in Kapitel 3.2 gegeben. Zu den Arbeiten, welche in diesem Kontext am engsten mit dem hier beschriebenen Konzept verwandt sind, gehört die Arbeit von Liu et al. in [184] sowie die Arbeiten von Niaki und Sander in [39] und [40].

In [184] wird die Idee, heterogene Simulationswerkzeuge über ein zentrales heterogenes M&S Framework (in diesem Fall ebenfalls Ptolemy II) zu integrieren, zum ersten Mal erwähnt. Als Methode wird die Verwendung von atomaren Actors als domänenspezifische „Tool Actor“ vorgeschlagen. Ein solcher „Tool Actor“ dient als Wrapper für ein externes CAE Tool, welches über eine beliebige proprietäre Schnittstelle eingebunden werden kann. Der Ansatz wird von Wetter in [265] für die Co-Simulation im Kontext sog. Building Control Virtual Testbeds (BCVTB) aufgegriffen. Die Implementierung von BCVTB fokussiert ausschließlich auf die Kopplung externer Simulatoren über das *Synchronous Data Flow (SDF)* Berechnungsmodell von PtII.

Das Konzept in [39, 40] ist prinzipiell mit den PtII Ansätzen vergleichbar. Als Middleware dient allerdings nicht PtII, sondern ForSyDe [226]. Externe Werkzeuge werden mit Hilfe von „Wrapper Processes“ in ForSyDe integriert. In [39] wird vorgeschlagen, die erlaubte dynamische Semantik des Datenaustauschs mit Hilfe einer Sequenz von „Simulation Functions“ formal zu beschreiben. Die Integration wird anhand eines synchronen Berechnungsmodells demonstriert. Eine automatische Generierung von Schnittstellen wird als mögliche Erweiterung erwähnt.

Im Unterschied zu den zuvor genannten Ansätzen kombiniert der in dieser Arbeit beschriebene Ansatz ein heterogenes M&S Framework mit einer standardisierten Middleware zur verteilten Co-Simulation. Der Fokus liegt dabei auf der verteilten Simulation V2X-basierter E/E Architekturen. Weder in [184] noch in [265] existieren Möglichkeiten zur expliziten Spezifikation der erlaubten statischen und dynamischen Semantik des Datenaustauschs zwischen Simulatoren. In [39, 40] ist dies zwar möglich, allerdings können mit Sequenzen von „Simulation Functions“ nur sehr einfache Interaktionsmuster modelliert werden. Eine Spezifikation der statischen Semantik ist in [39, 40] nicht möglich.

Die HLA Toolbox [4] und das HLA Blockset [3] sind zwei kommerzielle Lösungen, welche die HLA mit Matlab/Simulink integrieren. Allerdings besteht weder die Möglichkeit zur Erstellung von Federationmodellen, noch zur expliziten Beschreibung der erlaubten dynamischen Semantik des Datenaustauschs. In [242] wird ein Ansatz zur hybriden (gemischt diskret/kontinuierlichen) Simulation zwischen Matlab/Simulink und DEVS [269] unter Verwendung der HLA vorgeschlagen. Die Adaption der beiden Berechnungsmodelle erfolgt vollständig mit Hilfe propritärer HLA Adapter und ohne Unterstützung eines heterogenen M&S Frameworks. Die Erstellung von Federationmodellen ist nicht möglich. Zur gleichen Zeit wie diese Arbeit wurde mit [171] eine weitere Kopplung von PtII und HLA vorgestellt. Die Möglichkeit zur Modellierung von Federations ist in [171] nicht gegeben. Auch ist der geplante Anwendungsbereich nicht die Simulation von E/E Architekturen, sondern von industriellen Anlagen.

Im Kontext des *Model Integrated Computing (MIC)* [245] bzw. der *Model-driven Architecture (MDA)* [9] sind insbesondere die Arbeiten von Karsai et. al [154], Hemmingway et al. [131], Topcu et al. [252] und Adak et al. [33] relevant. In [154] werden verschiedene Entwurfsmuster für die Integration von Entwicklungswerkzeugen vorgestellt. Eine Lösung zur Spezifikation der erlaubten dynamischen Semantik des Datenaustauschs wird nicht beschrieben. In [131] wird ein HLA Metamodell basierend auf dem *Generic Modeling Environment (GME)* [245] beschrieben. Dieses dient als Basis für die Entwicklung von HLA Federations und zur Generierung von Schnittstellencode. Eine Metamodellierung von Schnittstellenverhalten ist ebenfalls nicht möglich. Dies ist explizit die Motivation der Arbeiten in [252, 33], welche auch GME nutzen. Weder in [131] noch in [252, 33] wird ein formales M&S Framework zum Management von Heterogenität und strukturierten Komposition von Modellen verwendet.

Neben Veins [239] existieren noch eine Reihe weiterer Simulatoren für die V2X Kommunikation wie z.B. TraNS [216], MoVES [62], iTETRIS [168, 221] oder V-SimRTI [218]. Hervorzuheben ist dabei insbesondere das VSimRTI Framework, welches eine modulare Kopplung von Simulatoren über eine „Runtime Infrastructure“ erlaubt, welche durch die HLA inspiriert ist. Der Fokus aller genannten V2X Simulatoren liegt auf einer vergleichsweise grobgranularen Betrach-

tung von V2X Kommunikation im Kontext von Verkehrsmanagement oder Verkehrseffizienz. Dabei werden einzelne Fahrzeuge grundsätzlich als abstrakter Punkt betrachtet. Für eine simulationsbasierte Verifikation von Komponenten einer E/E Architektur ist dieser Ansatz daher nicht ausreichend.

Die in diesem Kapitel vorgestellte Methode kombiniert ein heterogenes M&S Werkzeug (PtII) und eine HLA Implementierung namens CERTI zu einem neuen Werkzeug, das sich wie folgt in den DSEEP (vgl. Abschnitt 3.2.3.2) einordnet: Dadurch, dass PtII sowohl als Werkzeug zur Modellierung von Schnittstellen zur Co-Simulation fungiert als auch die Grundlage für die Ausführung einer heterogenen verteilten Co-Simulation ist, erstreckt sich der Anwendungsbereich des vorgestellten Ansatzes vom Entwurf (DSEEP Schritt 3) über Entwicklung (DSEEP Schritt 4), Integration und Test (DSEEP Schritt 5) und die Ausführung (DSEEP Schritt 6) bis zur Nutzung von PtII als zentrales Analysewerkzeug (DSEEP Schritt 7). Die sechs Schritte der Vorgehensweise aus Abschnitt 5.3.2 können als Spezialisierung der Schritte 3 bis 5 des DSEEP betrachtet werden. SDEMMlib wirkt insbesondere in den DSEEP Schritten 4 und 5 unterstützend.

Die beschriebene Methode ist als erster prinzipieller Ansatz zu verstehen. Obgleich eine heterogene verteilte Co-Simulation verschiedener PtII Domänen bereits jetzt möglich ist, ist das zurzeit per BIM FSM modellierte Synchronisationsverfahren auf die verteilte Kopplung diskreter ereignisbasierter Simulatoren limitiert. Für die Integration weiterer disziplin- und domänenspezifischer Simulatoren ist es daher notwendig, neue geeignete BIM Spezifikationen zu entwickeln. In diesem Zusammenhang sind detailliertere Untersuchungen hinsichtlich Charakteristika von Berechnungsmodellen notwendig, wie diese die dynamische Semantik des Datenaustauschs beeinflussen und in welchen Fällen eine Kopplung von Werkzeugen via Ptolemy II unabdingbar bzw. verzichtbar ist. Das Ziel könnte ein weiterer Grad der Automatisierung sein, bei dem BIM FSMs automatisch generiert werden können. Schlussendlich wurden bisher keine detaillierten Performanzuntersuchungen im Kontext der Co-Simulation durchgeführt. Auch eine Kombination aus paralleler und kooperativer Simulationsmethoden ist denkbar.

6. Schlussfolgerung und Ausblick

6.1. Zusammenfassung und Schlussfolgerung

Im Rahmen dieser Dissertation wurden neue Methoden und Werkzeuge vorgestellt, welche das Ziel haben, den Herausforderungen an Entwicklungsprozesse für zukünftige eingebettete Systeme zu begegnen. Dazu wurden zu Beginn dieser Arbeit zunächst Methoden

1. zur Beschleunigung von Simulationswerkzeugen sowie
2. zur Verbesserung der Interoperabilität zwischen Simulationswerkzeugen

als zentrale Teilaspekte von zukünftigen Entwicklungsprozessen für eingebettete Systeme identifiziert. Anschließend wurden neue Methoden und Werkzeuge vorgestellt, um diesen Herausforderungen zu begegnen.

Hinsichtlich Teilaspekt I wurde ein allgemeiner Ansatz für die parallele Simulation von eingebetteten MPSoCs auf Manycore Architekturen abgeleitet, welcher sich aus einer schichtenorientierten SystemC-basierten Laufzeitumgebung und dem Konzept der Simulationssynthese zusammensetzt. Ausgehend davon wurden verschiedene alternative Ansätze und Strategien für die parallele SystemC-basierte Simulation von zyklenakkuraten und zyklenapproximativen Modellen entwickelt.

Die verschiedenen Strategien wurden implementiert, experimentell untersucht und anhand verschiedener MPSoC Modelle bewertet. Zyklenakkurate und zyklenapproximative Modelle standen dabei im Vordergrund der Betrachtung, da sie sich typischerweise durch extrem lange Laufzeiten auszeichnen. Begleitend wurden Möglichkeiten zur Automatisierung des Parallelisierungsprozesses anhand einer teil- und einer vollautomatisierten Werkzeugkette entwickelt und exemplarisch in einen existierenden MPSoC Entwurfsfluss integriert. Insgesamt sind folgende Beiträge in Bezug auf Teilaspekt I hervorzuheben, da sie über den aktuellen Forschungsstand hinausgehen:

- Eine in Schichten strukturierte Laufzeitumgebung für die parallele Simulation von SystemC-basierten Simulationsmodellen. Die Schichtenarchitektur ist die Basis für Adaptivität, Konfigurierbarkeit und Optimierung der parallelen Simulation in Abhängigkeit von Modell und Ausführungsplatt-

form. Auf Basis des Konzepts der Schichtenarchitektur wurden diverse Parallelisierungsstrategien zu Vergleichszwecken umgesetzt.

- Methoden zur semi-automatischen und vollautomatischen Synthese von parallelen SystemC Simulationen für zukünftige Manycore Architekturen. Die vollautomatische Synthesemethode basiert auf einer Kombination aus statischer Compileranalyse und dynamischer Laufzeitanalyse für die Extraktion von Modellinformationen, mit dem Ziel der feingranularen Modellpartitionierung und der Konfiguration der zugrundeliegenden Laufzeitumgebung.
- Die Klassifikation von logischen Links in zyklenakkuraten und zyklenapproximativen parallelen SystemC Simulationen in deadlock-kritisch und deadlock-unkritisch anhand spezieller Modelleigenschaften. Es wurde gezeigt, dass es möglich ist, zusätzliche Kausalitätsbedingungen für das Kernelscheduling anhand von Eigenschaften logischer Links abzuleiten, deren Einhaltung eine kausal korrekte und deadlockfreie parallele Ausführung garantiert[1]. Der Ansatz dient zudem als Basis für eine Optimierung der Performanz, insbesondere im Hinblick auf die Vermeidung globaler Synchronisation.
- Eine Modellierungsmethodik für die zyklenapproximative MPSoC Simulation auf Transaktionsebene, bestehend aus einem parallelisierbaren deterministischen TLM Kommunikationsprotokoll und einem hierarchischen Scheduling. Die Kombination aus beidem kann zur adaptiven temporären Entkopplung auf Basis einer dynamischen Berechnung sog. lokaler Quanta genutzt werden. Die dynamische Berechnung lokaler Quanta gestattet eine kontrollierte Beschränkung des zeitlichen Fehlers unter Verwendung von Informationen über den Modellzustand zur Laufzeit.
- Die Anwendung von Laufzeitumgebung, Synthese- und Modellierungsmethoden auf dem Single-chip Cloud Computer (SCC) und einem gewöhnlichen Shared Memory Multiprozessor und die Durchführung umfassender Messreihen. Die Architektur des SCC dient als Blaupause für zukünftige Manycore Architekturen.

In Bezug auf Teilaspekt II, die Verbesserung der Interoperabilität zwischen existierenden Simulationswerkzeugen, wurde die Problemstellung zunächst anhand des Entwicklungsprozesses für zukünftige V2X basierte automobile E/E Architekturen konkretisiert. Die sich ergebende Haupteinsicht war, dass prinzipiell eine interdisziplinäre und nicht nur eine multidisziplinäre Herangehensweise notwendig ist, um Wechselwirkungen zwischen Teilsystemen möglichst frühzeitig

[1] Zyklenapproximative Modelle müssen der in Abschnitt 4.6 beschriebenen TL Modellierungsmethode folgen. Bei diesen Modellen ist eine Ableitung zusätzlicher Kausalitätsbedingungen für das Kernelscheduling grundsätzlich nicht notwendig, da sie sich auf der Ebene von Timedcycles selbst synchronisieren.

berücksichtigen zu können. Für ausführbare Modelle kann eine solche interdisziplinäre Sichtweise durch kooperative Simulation erreicht werden.

Aus diesem Kontext heraus wurde ein methodischer Ansatz abgeleitet, der in einem Entwicklungsprozess für eingebettete Systeme bei der schnellen Herstellung von Interoperabilität zwischen heterogenen Simulationswerkzeugen hilfreich ist. Die Machbarkeit des Ansatzes wurde anhand einer prototypisch implementierten Werkzeugkette demonstriert und anhand verschiedener Fallstudien untersucht und bewertet. Geleistete wissenschaftliche Beiträge bzgl. Teilaspekt II, die über den aktuellen Stand der Forschung hinausgehen, sind:

- Die Verknüpfung eines heterogenen Modellierungs- und Simulationswerkzeugs namens Ptolemy II und einer Simulationsmiddleware namens High Level Architecture zu einer Simulatorarchitektur. Diese dient als Basis für eine Methode, welche es erlaubt, heterogene Simulationswerkzeuge strukturiert zu kombinieren und verteilt auszuführen. Die Methode erlaubt zusätzlich eine explizite modellbasierte Spezifikation eines *„Kommunikationsvertrags“* zwischen Simulationswerkzeugen und HLA, wodurch die erlaubte statische und dynamische Semantik des Datenaustauschs festgelegt wird. Sie lässt sich mit einer simulationsbasierten Entwurfsmethodik für eingebettete Systeme kombinieren.
- Die Umsetzung der Methode anhand einer Werkzeugkette, welche Entwurf, Integration und Test einer heterogenen verteilten Co-Simulation teilautomatisiert unterstützt. Neben der Rolle als Koordinator einer heterogenen Co-Simulation zur Laufzeit dient Ptolemy II insbesondere auch als Anwenderschnittstelle bei der Realisierung neuer Simulatorkopplungen vor der Laufzeit. Mit Hilfe eines Simulatoradapters in Form einer eigens implementierten erweiterbaren C++ Bibliothek ist es möglich, Schnittstellenspezifikationen direkt auszuführen, was für das Debugging nützlich ist.
- Die Demonstration der Anwendbarkeit des Gesamtkonzepts in verschiedenen Fallstudien. Diese basieren auf verschiedenen Simulatorkopplungen, welche wiederum aus unterschiedlichen Kombinationen von PtII, dem Netzwerksimulator OMNeT++, dem V2X Simulator Veins und einem in SystemC implementierten MPSoC Modell bestehen.

6.2. Ausblick

Die im dieser Dissertation erarbeiteten Konzepte und Methoden sowie die entstandenen Werkzeuge können als Ausgangspunkt für weitere Forschungsarbeiten im Bereich des rechnergestützten Entwurfs, der Modellierung und der Simulation von zukünftigen eingebetteten elektronischen Systemen angesehen wer-

den. Dies schließt sowohl die Entwicklung von Methoden mit ein, welche die in dieser Dissertation erarbeiteten Methoden weiterentwickeln, als auch die Anwendung der entstandenen Werkzeuge für die Untersuchung neuer Applikationen und Funktionen.

Die umgesetzten Strategien zur parallelen SystemC Simulation sind für die Nutzung in Verbindung mit zyklenakkuraten und ausgewählten zyklenapproximativen TL Modellen sowie homogenen Ausführungsplattformen geeignet. Ziel zukünftiger kann es sein, die Methoden auf eines größeres Spektrum von Modellierungsstilen sowie eine Kombination aus heterogenen Ausführungsplattformen zu erweitern.

Auch die entwickelten Ansätze zur Verbesserung der Interoperabilität bieten Potential für zukünftige Arbeiten. Für die Integration weiterer disziplinspezifischer Simulatoren sind detailliertere Untersuchungen hinsichtlich der Fragestellung notwendig, welche Charakteristika von Berechnungsmodellen zu welcher dynamischen Semantik des Datenaustauschs führen und wie sich dies auf die Kopplung auswirkt. Aus einer anwendungsorientierten Perspektive ist eine Übertragung der Konzepte auf andere einschlägige Standards wie FMI denkbar. Auch eine Zusammenführung paralleler und kooperativer Simulationsmethoden ist erstrebenswert.

Durch die prinzipielle Erweiterbarkeit aller umgesetzten Komponenten sind zukünftige Arbeiten im Kontext der Anwendung von paralleler oder kooperativer Simulation mit Hilfe der entwickelten Werkzeuge nicht auf eine bestimmte Anwendungsdomäne für eingebettete Systeme beschränkt. Am ITIV werden beide Werkzeuge in erster Linie für die Forschung im Bereich eingebetteter Multiprozessorsysteme sowie V2X basierter automobiler E/E Architekturen dienen.

A. On-Chip Kommunikation auf dem SCC

Da der MPB Speicher des SCC auf 48×16 KB limitiert ist, eignet er sich weniger für die Anwendung in einem traditionellen SHM Kontext, als vielmehr zum Zwischenpuffern von Nachrichten beim *Message-Passing (MP)*. Tatsächlich besitzen die Kerne des SCC mit dem MPB eine native Hardwareschnittstelle für einseitige Kommunikation [110] (vgl. Abschnitt 2.4.2.2). Mechanismen zur

1. Speicherorganisation des MPB,
2. Zugriffssteuerung und Synchronisation sowie
3. Integration mit dem privaten Speicherbereich eines jeden SCC Kerns

stehen jedoch nicht unmittelbar zur Verfügung, sondern müssen softwareseitig in Form eines Protokolls realisiert werden. Der Entwurfsraum für ein solches Protokoll präsentiert sich als multidimensional. In [222] sind mögliche Entwurfsalternativen für den Spezialfall nicht-blockierender einseitiger Übertragung von kleinen Nachrichten (< 200 Byte) zusammenfassend dargestellt. Alleine dafür werden insgesamt sechs Dimensionen aufgeführt.

A.1. Existierende leichtgewichtige Lösungen

Die standardmäßig von Intel für den SCC zur Verfügung gestellte RCCE Bibliothek [198][15] umfasst bereits eine einfache MP API zur On-Chip Kommunikation. Dazu teilt RCCE die 16 KB MPB Speicher, die pro Tile vorhanden sind, in jeweils zwei 8 KB große Blöcke auf. Ein kleiner Teil dieser Blöcke wird für Statusflags benutzt, der Rest dient als Puffer für das MP. Für den Datenaustausch existiert die Gory API und die Basic API. Erstere ist einseitig und nicht-blockierend, Letztere zweiseitig und blockierend. Verlustfreie Kommunikation ist nur mit der Basic API möglich. Die enge Kopplung von Sende- und Empfangsvorgängen, die mit der zweiseitigen blockierenden Basic API einhergeht, kann aus folgenden Gründen zum Nachteil werden:

1. Durch die blockierende Schnittstelle erfolgen Sende-, Empfangs- und Berechnungsvorgänge *innerhalb* eines Prozesses zwangsläufig sequentiell. Damit muss die Reihenfolge von Sende- und Empfangsvorgängen auch *zwischen* Prozessen einer Topologie bekannt sein, um Deadlocks zu vermeiden (vgl. Ringshift-Beispiel in [198]).
2. Durch die implizite Synchronisation während einer einzelnen Übertragung können mehrere Sende- und Empfangsvorgänge *zwischen* Prozessen nicht asynchron zueinander erfolgen.
3. Zu jedem Zeitpunkt kann innerhalb eines 8 KB MPB Blocks immer nur eine einzige Nachricht gespeichert werden. Bei Übertragung relativ kleiner Nachrichten kann die Bandbreite einbrechen (vgl. [198]).

In der sog. iRCCE Bibliothek [89] existiert zwar eine nicht-blockierende Schnittstelle. Allerdings erfolgt eine einzelne Datenübertragung mit dieser Schnittstelle weiterhin zweiseitig und damit synchron. Die hat zur Folge, dass Prozesse während eines Sende- oder Empfangsvorgangs solange aktiv pollen müssen, bis eine Synchronisation mit dem Kommunikationspartner stattgefunden hat. Auch der Pipelinemodus von iRCCE basiert auf einer blockierenden Schnittstelle und arbeitet prinzipiell synchron. Zudem ist der Pipelinemodus nur für Datenpakete sinnvoll anwendbar, die größer als 8 KB sind [89].

A.2. Implementierung

Aus den genannten Gründen wurde die RCCE Bibliothek um die Möglichkeit zur verlustlosen einseitigen Kommunikation erweitert. Diese ist bei unregelmäßigen Kommunikationmustern vorteilhaft [110, 48]. Insbesondere wird die Kommunikation von der Synchronisation separiert, was die Grundlage für die Implementierung einer von der Basisdienstebene unabhängigen logischen Ebene ist.

Im Unterschied zu RCCE und iRCCE basiert das entwickelte Verfahren auf dynamischen Message Queues für jeden SCC Kern. Diese werden im MPB Speicherbereich des jeweiligen Tiles instanziiert. Zur Übertragung einer Nachricht muss ein Sender die Nachricht in die Message Queue des Empfängers schreiben. In einer Message Queue können gleichzeitig mehrere Nachrichten vorgehalten werden. Sender und Empfänger müssen einen Übertragungsvorgang nicht mehr aktiv koordinieren. Das Prinzip ist in Abb. A.1 am Beispiel der Kommunikation zwischen zwei SCC Kernen *A* und *B* illustriert.

Betrachtet man beispielsweise die Übertragung einer *Message* von Kern *A* nach Kern *B*, so existiert eine Kette von zwei Puffern, bestehend aus einem *Send Buffer* Objekt im privaten Speicherbereich von Kern *A* und einem als *Stream Buffer* bezeichneten Speicherbereich im gemeinsam genutzten MPB Bereich auf der Seite

des empfangenden Kerns *B*. Der Datentransfer über diese Kette von Puffern wird auf Sender- und Empfängerseite jeweils durch ein *Message Buffer* Objekt gesteuert. Dieses organisiert den *Stream Buffer* als statischen oder dynamischen Ringpuffer. Als Grundlage dafür greift das *Message Buffer* Objekt auf einen *Stream Proxy* zurück, welcher den *Stream Buffer* zunächst als bytebasierten statischen Ringpuffer organisiert.

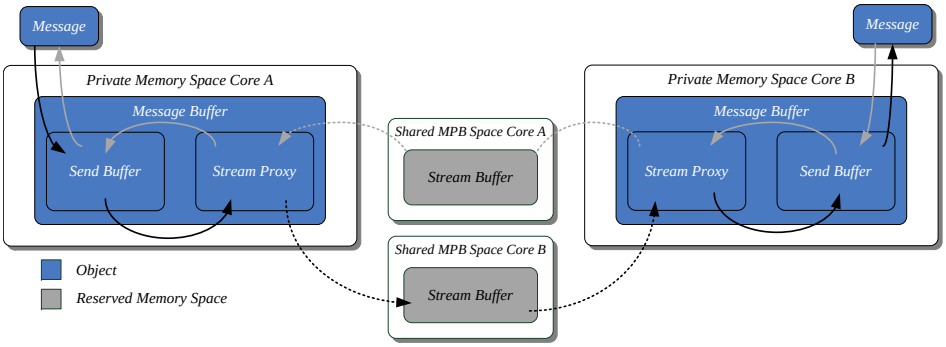


Abbildung A.1.: On-Chip Message-Passing am Beispiel zweier SCC Kerne

A.2.1. Send Buffer

Der *Send Buffer* befindet sich im privaten Speicherbereich. Er dient der Zwischenspeicherung von Nachrichten, bevor eine Übertragung über das NoC zum einem beliebigen *Stream Buffer* erfolgt. Ein *Send Buffer* setzt sich aus mehreren verketteten Listen zusammen, einer *Receiver List* und mehreren *Message List*s. In der *Receiver List* werden IDs aller erreichbaren SCC Kerne mit Zeigern auf die *Message List*s abgespeichert. Eine ID identifiziert damit sowohl eine *Message List* im privaten Speicher als auch den zugehörigen *Stream Buffer* im MPB Bereich des jeweiligen Ziel SCC Kerns. Eine *Message List* besteht aus Zeigern auf *Message*s. *Message*s werden dabei immer in der Reihenfolge abgespeichert, in der sie eingefügt wurden. Die Größe der *Message List*s ist allein durch die Größe des verfügbaren privaten Speichers eines SCC Kerns beschränkt.

A.2.2. Stream Proxy

Ein *Stream Proxy* verfügt über einen *bufStart* Zeiger, mit dem beliebige *Stream Buffer* Speicherbereiche referenziert werden können. Im Lesefall referenziert der *bufStart* Zeiger den eigenen *Stream Buffer*. Im Schreibfall referenziert der *bufStart* Zeiger einen *Stream Buffer* eines anderen SCC Kerns.

Stream Buffer werden von einem *Stream Proxy* als statische Ringpuffer zur Übertragung von Bytestreams organisiert. Die Implementierung beinhaltet die Cachekohärenzsicherung durch Anwendung des SCC-spezifischen Invalidierungsmechanismus: Damit L1 Caching beim MPB Zugriff möglich ist (MPMT + L1CM, vgl. Abschnitt 2.4.3.2), muss sowohl vor einer Lese- als auch vor einer Schreiboperation zunächst die `CL1INVMB` Instruktion ausgeführt werden (vgl. Abschnitt 2.4.3.2). Bei einem Schreibvorgang werden dabei nur Daten mit einer Granularität von 32 Byte (Größe einer Cachline) vom vorgeschalteten Write-Combine Buffer (WCB) unmittelbar in den MPB übertragen (WCB Flush). Um einen WCB Flush auch bei Daten von geringerer Größe als einer Cacheline auszulösen, genügt es, Junk-Daten in irgendeine beliebige Cacheline zu schreiben.

In Abb. A.3 ist die Speicherorganisation durch den *Stream Proxy* anhand der hellgrau unterlegten Felder dargestellt. Der *bufStart* Zeiger speichert die Startadresse eines *Stream Buffer*s. *bufSize* entspricht dessen Gesamtgröße in Bytes. *bufOffset* spezifiziert die Distanz zu *bufStart* in Bytes, ab der ein Schreib- bzw. Lesezugriff erfolgen soll.

- **Lesezugriff**: Falls $readSize \leq bufSize$, so werden die MPMT Zeilen im L1 Cache mit der `CL1INVMB` Instruktion invalidiert und unter Berücksichtigung der Modulo-Arithmetik *readSize* Bytes vom *Stream Buffer* eingelesen. Falls $readSize > bufSize$, liegt ein Fehlerzustand vor.
- **Schreibzugriff**: Falls $writeSize \leq bufSize$, so werden die MPMT Zeilen im L1 Cache mit der `CL1INVMB` Instruktion invalidiert und unter Berücksichtigung der Modulo-Arithmetik *writeSize* Bytes in den zugehörigen *Stream Buffer* geschrieben. Falls $writeSize > bufSize$, so liegt ein Fehlerzustand vor.

A.2.3. Message Buffer

Der *Message Buffer* hat drei Aufgaben: I) Verwaltung der *Stream Buffer* als statische oder dynamische Ringpuffer mit einer Granularität von sog. *Transmission*s II) Garantie eines wechselseitigen Ausschlusses bei *Stream Buffer* Zugriffen III) Vermeidung von *Stream Buffer* Überläufen.

A.2.3.1. Verwaltung statischer oder dynamischer Ringpuffer

Zur Abbildung von *Messages* auf einen durch den *Stream Proxy* bereitgestellten Bytestream verwaltet der *Message Buffer* einen statischen oder einen dynamischen Ringpuffer auf Basis von *Transmission*s. *Transmission*s sind immer Vielfache von einem Byte. Im Fall eines dynamischen Ringpuffers können sie in der Größe variieren. Da statische Ringpuffer ein vereinfachter Sonderfall dynamischer Ringpuffer sind, werden im Folgenden nur dynamische Ringpuffer beschrieben.

Eine *Transmission* besteht aus einem drei Byte *Header* und einem *Payload* von variabler Größe (siehe Abb. A.2). Der Header besteht wiederum aus einer *processID* und einem *size* Feld. Die *processID* identifiziert den sendenden SCC Kern, die *size* spezifiziert die Größe des Payloads.

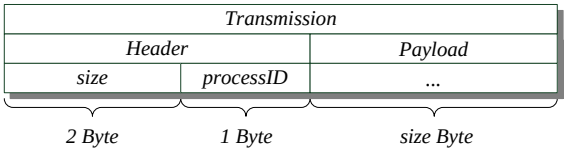


Abbildung A.2.: Struktur einer *Transmission*

In Abb. A.3 ist dargestellt, wie *Transmissions* im Fall eines dynamischen Ringpuffers auf einzelne Bytes abgebildet werden. Im Beispiel befindet sich die erste (am längsten im Puffer befindliche) *Transmission* rechts und die zuletzt gespeicherte *Transmission* links. Um beide identifizieren zu können, werden zusätzlich zwei Adresszähler namens *first* und *last* im MPB Speicherbereich allokiert. Ersterer zeigt auf das erste Byte der ersten *Transmission*, Letzterer auf das erste Byte des noch unbelegten Speichers im *Stream Buffer*. Aufgrund der Tatsache, dass *Transmission*s linear in der Reihenfolge der Schreibzugriffe abgelegt werden, ist es möglich, den *bufOffset* der nächsten zu lesenden *Transmission* durch Addition des *bufOffset* der letzten gelesenen *Transmission* mit deren Header- und Payloadlänge wie folgt zu berechnen: $bufOffset_{new} = bufOffset + 3 + size$. Das Gleiche gilt für die Berechnung des nächsten *bufOffset* zum Schreiben einer neuen *Transmission*.

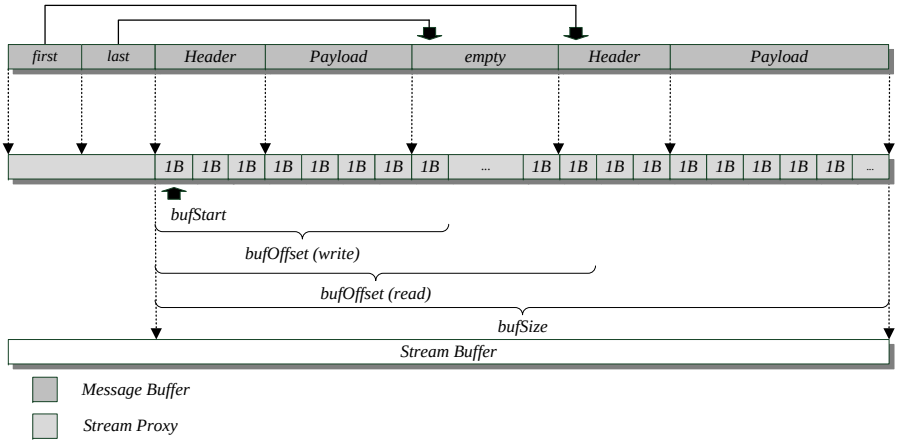


Abbildung A.3.: Abbildung statischer und dynamischer Ringpuffer in den MPB

A.2.3.2. Wechselseitiger Ausschluss

Ohne garantierten wechselseitigen Ausschluss kann das Ergebnis von Speicherzugriffen von einer zufälligen Zugriffsreihenfolge abhängen. Solche sog. Race Conditions können dann auftreten, wenn Speicherzugriffe nicht atomar sind. Bei Verwendung des MPMT+L1C Modus ist atomarer Zugriff nur für Schreiboperationen garantiert, die an eine Cachline (32 Byte) angepasst sind und die eine Cacheline in der Größe weder über- noch unterschreiten (vgl. Abschnitt 2.4.3.2). Für alle anderen Zugriffe muss wechselseitiger Ausschluss mit zusätzlichen Mechanismen hergestellt werden. Die implementierte Methode unterscheidet wechselseitigen Ausschluss zwischen Sendern, die auf ein und denselben *Stream Buffer* zugreifen wollen, von wechselseitigem Ausschluss zwischen einem Sender und einem Empfänger.

Im ersten Fall wir im MPB Bereich eines jeden SCC Kerns ein Datenfeld instanziiert, welches die *processID* des Senders speichert, der aktuell exklusiven Schreibzugriff auf den zugehörigen *Stream Buffer* hat. Setzen oder Löschen des Datenfeldes wird über das Test-and-Set Register des jeweiligen Kerns geschützt. Die Methode entspricht der Implementierung von Synchronization Flags in der RCCE Bibliothek [266].

der zweite Fall kann vollständig ohne Test-and-Set Register umgesetzt werden. Dazu werden die *first* und *last* Zeiger ebenfalls in separate Cachelines geschrieben. Da Schreibzugriffe bei Verwendung des MPMT+L1C Modus bis 32 Byte atomar erfolgen, können Sender und Empfänger ohne zusätzliche Synchronisation auf den *Stream Buffer* zugreifen.

A.2.3.3. (De-)Serialisierung und Flusssteuerung

Das Senden einer Nachricht beinhaltet die Serialisierung von Daten, die in *Message* Objekten gespeichert sind, in einen *Payload*, das Hinzufügen eines *Header*s und den Transfer in den *Stream Buffer* mit Hilfe des *Stream Proxy* Objekts. Ein Lesezugriff auf Empfängerseite beinhaltet das Auslesen von Daten, die Extraktion von *Header* und *Payload* sowie die anschließende Deserialisierung in *Message* Objekte.

Um Datenverlust zu vermeiden, werden Nachrichten im *Send Buffer* zwischengespeichert, sobald der *Stream Proxy* einen vollen *Stream Buffer* meldet. Der zweistufige Pufferungsmechanismus mit dem zusätzlichen Puffer im privaten Speicher ist damit Voraussetzung für nicht-blockierende Kommunikation. Eine generelle Zwischenpufferung im *Send Buffer* ist optional möglich.

B. Services der HLA

B.1. Kategorien

Die verschiedenen Kategorien der HLA Schnittstellen [23] haben folgende Aufgaben:

1. Die *Federation Management (FM)* Services sind für die Erzeugung, dynamische Kontrolle, Modifikation und Löschung von Federations verantwortlich.
2. Die *Declaration Management (DM)* Services werden von Federates zur Deklaration von Information genutzt, die von ihnen generiert (*published*) und konsumiert (*subscribed*) werden kann.
3. Die *Object Management (OM)* Services dienen zur Registrierung, Modifikation und Löschung von Instanzen von HLA Objects und Interactions. Insbesondere kann für jedes Federate spezifiziert werden, welche Aktualisierungen für welche HLA Object / Interaction Classes es publizieren (*publish*) oder abonnieren (*subscribe*) möchte.
4. Die *Ownership Management (OWM)* Services werden von Federates und RTI genutzt, um den Besitz von Instanzen von HLA Attributes zwischen Federates zu übertragen.
5. Die *Time Management (TM)* Services stellen Mechanismen bereit, um die Reihenfolge der Zustellung von Nachrichten während der Ausführung einer Federation zu ordnen.
6. Die *Data Distribution Management (DDM)* Services erlauben es den Federates, die Menge der zu sendenden und emfangenden Informationen anhand von zusätzlichen Informationen über deren Relevanz gezielt zu reduzieren.
7. Die *Support Services* fassen sonstige Dienste zusammen, welcher z.B. zur Transformation von Namen auf Handles und umgekehrt, zum Auslesen von Variablen oder zum Setzen von Flags dienen.

B.2. Verwendete HLA 1.3 Services

Folgende DoD HLA 1.3 Services (einige Support Services ausgenommen) wurden im Rahmen dieser Arbeit verwendet (vgl. [257]):

Federation Management:

- *A.1.1 createFederationExecution()*: Erzeugt eine aktive Federation und registriert diese bei der RTI.
- *A.1.2 destroyFederationExecution()*: Löscht die Registrierung einer aktiven Federation bei der RTI und fährt diese herunter.
- *A.1.10 joinFederationExecution()*: Stellt eine Anfrage zum Beitritt in eine Federation und initialisiert den RTI Ambassador mit federationspezifischen Daten (FED Datei).
- *A.1.12 registerFederationSynchronizationPoint()*: Initiiert die Registrierung eines Haltepunktes, der zur Synchronisation einiger oder aller Federates dient. Die Synchronisation selbst erfolgt im Rahmen einer federationspezifischen Semantik.
- *A.1.18 resignFederationExecution()*: Beendet die Teilnahme eines Federates innerhalb einer Federation.
- *A.1.22 synchronizationPointAchieved()*: Informiert die Federation darüber, dass das Federate die federationspezifischen Kriterien erfüllt hat, die mit der Erreichung des Synchronisationspunktes verbunden sind, der zuletzt registriert wurde.
- *B.1.1 announceSynchronizationPoint()*: Informiert das Federate darüber, dass im Rahmen einer federationspezifischen Semantik die Registrierung eines Synchronisationspunktes angefordert wurde.
- *B.1.7 federationSynchronized()*: Informiert ein Federate darüber, dass ein zuvor beim Federate registrierter Synchronisationspunkt von allen relevanten Federates erreicht wurde.
- *B.1.15 synchronizationPointRegistrationFailed()*: Informiert ein Federate darüber, dass der Versuch, einen Synchronisationspunkt zu registrieren, fehlgeschlagen ist.
- *B.1.16 synchronizationPointRegistrationSucceeded()*: Informiert ein Federate darüber, dass der Versuch, einen Synchronisationspunkt zu registrieren, erfolgreich war.

Declaration Management:

- *A.2.2 publishObjectClass()*: Informiert die RTI über die Absicht eines Federates, Instanzen von Attributen einer bestimmten Object Class zu erzeugen und zu aktualisieren.

- *A.2.5 subscribeObjectClassAttributes()*: Informiert die RTI über die Absicht eines Federates, Aktualisierungen (sog. Refelctions) für eine bestimmte Menge von Attributen einer Object Class zu empfangen.

Object Management:

- *A.3.7 registerObjectInstance()*: Erzeugt eine neue Instanz einer Object Class innerhalb der Federation. Alle Attribute einer Object Class, die vom Federate publiziert werden, müssen dann vom Federate auf dieser Instanz auch aktualisiert werden (sog. Instanz-Attribute).
- *A.3.12 updateAttributeValues()*: Informiert die Federation über die Änderung des Wertes eines oder mehrerer Instanz-Attribute einer Object Class Instanz.
- *B.3.4 discoverObjectInstance()*: Informiert ein Federate über die Existenz einer HLA Object Instance innerhalb der Federation, welche für das Federate, aufgrund einer zuvor getätigten Subskription, relevant ist.
- *B.3.7 reflectAttributeValues()*: Informiert ein Federate über die Aktualisierung der Werte einer Menge von Instanz-Attributen, welche für das Federate, aufgrund einer zuvor getätigten Subskription, relevant ist.

Time Management:

- *A.5.7 enableTimeConstrained()*: Instruiert die RTI, den Zeitfortschritt eines Federates unter Berücksichtigung der Zeit der Federation zu beschränken und Nachrichten mit Zeitstempel in der korrekten Reihenfolge zu übermitteln.
- *A.5.8 enableTimeRegulation()*: Instruiert die Federation, die lokale Zeit des Federates bei der Berechnung der Zeit der Federation zu berücksichtigen.
- *A.5.12 nextEventRequestAvailable()*: Fragt einen Zeitfortschritt bis zu einem als Parameter übergegebenen Zeitstempel an. Bei einem darauffolgenden *timeAdvanceGrant()* ist nicht garantiert, dass alle Aktualisierungen für einen bestimmten Zeitpunkt vollständig übermittelt worden sind.
- *B.5.2 timeAdvanceGrant()*: Informiert ein Federate darüber, dass eine vorausgegangene Anfrage bzgl. eines Zeitfortschritts abgeschlossen ist. Dabei wird der neue Wert der lokalen Zeit zurückgeliefert.
- *B.5.3 timeConstrainedEnabled()*: Informiert das Federate, dass die zuvor per *enableTimeConstrained()* getätigte Anfrage erfolgreich war.
- *B.5.4 timeRegulationEnabled()*: Informiert das Federate, dass die zuvor per *enableTimeRegulation()* getätigte Anfrage erfolgreich war.

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

Abkürzungen ....

AP	Abstract Representation
API	Application Programming Interface
AST	Abstract Syntax Tree
AT	Approximately-Timed
BIM	Behavioral Interface Model
BIMM	Behavioral Interface Metamodel
BOM	Base Object Model
CAE	Computer-aided Engineering
CAL	Cycle-Approximate Level
CAM	Cooperative Awareness Message
CAP	Channel Activity Property
CAS	Compare-and-Set
CAS	Cycle-Approximate Simulator
CC-NUMA	Cache-Coherent Non-Unified Memory Access
CC-UMA	Cache-Coherent Unified Memory Access
CM	Cached Mode
CMB	Chandy-Misra-Bryant
COMA	Cache Only Memory Access
COW	Cluster of Workstations
CPI	Cycles Per Instruction
CPS	Cyber-Physical-System
CPU	Central Processing Unit
CT	Continuous Time
DAG	Directed Acyclic Graph
DAR	Dynamic Abstract Representation
DBP	Delta-Boundedness Property
DDG	Dynamic Dependency Graph
DDM	Data Distribution Management
DdW	Drive-by-Wire
DE	Discrete Event
DES	Discrete Event Simulation
DLP	Data-Level Parallelism
DLP	Dynamische Latenzprädiktion

DM ..... Declaration Management
DM ..... Distributed Memory
DMSO ..... Defense Modeling and Simulation Office
DoD ..... Department of Defense
DSC ..... Domain Starvation Condition
DSEEP ..... Distributed Simulation Engineering and Execution Process
DSM ..... Distributed Shared Memory
E/E ..... Elektrik/Elektronik
E/U ..... Evaluate/Update
ECU ..... Electronic Control Unit
EDA ..... Electronic Design Automation
ELOCC ..... Extended Local Causality Condition
ESL ..... Electronic System-Level
FDD ..... Federation Object Model Document Data
FED ..... Federation Execution Data
FIFO ..... First-In-First-Out
FM ..... Federation Management
FMI ..... Functional Mock-up Interface
FMU ..... Functional Mock-up Unit
FOM ..... Federation Object Model
FSM ..... Finite State Machine
GME ..... General Modeling Environment
GPGPU ..... General Purpose Computation on Graphics Processing Unit
GPU ..... Graphics Processing Unit
HDL ..... Hardware Description Language
HdS ..... Hardware-dependent Software
HeMPS ..... Hermes Multiprocessor System
HLA ..... High Level Architecture
HMT ..... Hardwareseitiges Multithreading
HW ..... Hardware
I/O ..... Input/Output
IAA ..... Intra-Phasen-Abhängigkeiten
IDM ..... Intelligent Driver Model
IEEE ..... Institute of Electrical and Electronics Engineers
IFW ..... Interface Wrapper
IKT ..... Informations- und Kommunikationstechnologie
ILP ..... Instruction-Level Parallelism
IPC ..... Inter-Process Communication
IR ..... Intermediate Representation
IRA ..... Inter-Phasen-Abhängigkeiten
ISO ..... International Organization for Standardization
JITC ..... Just-in-Time Compilation

LAN ..... Local Area Network
LAP ..... Link Activity Property
LBP ..... Link Boundedness Property
LCC1 ..... First Link Criticality Condition
LCC2 ..... Second Link Criticality Condition
LCIM ..... Levels of Conceptual Interoperability Model
LDP ..... Link Delay Property
LLVM ..... Low Level Virtual Machine
LOBC ..... Local Bound Condition
LOCC ..... Local Causality Condition
LT ..... Loosely-Timed
LUT ..... Lookup Table
M&S ..... Modellierung und Simulation
MDA ..... Model-driven Architecture
MESI ..... Modified Exclusive Shared Invalid
MIC ..... Many-Integrated-Core
MIC ..... Model Integrated Computing
MIMD ..... Multiple Instruction Multiple Data
MIPS ..... Microprocessor without Interlocked Pipeline Stages
MISD ..... Multiple Instruction Single Data
MIU ..... Mesh Interface Unit
ML ..... Mixed Level
MLCC1 ..... Modified First Link Criticality Condition
MoC ..... Model of Computation
MoML ..... Modeling Markup Language
MP ..... Message Passing
MPB ..... Message Passing Buffer
MPI ..... Message Passing Interface
MPMT ..... Message Passing Memory Type
MPP ..... Massively Parallel Processors
MPSoC ..... Multiprocessor System-on-Chip
NCC-NUMA ..... Non-Cache-Coherent Non-Unified Memory Access
NCC-UMA ..... Non-Cache-Coherent Unified Memory Access
NERA ..... Next Event Request Available
NI ..... Network Interface
NMA ..... Null Message Algorithmus
NoC ..... Network-on-Chip
NOW ..... Network of Workstations
NUMA ..... Non-Unified Memory Access
OEM ..... Original Equipment Manufacturer
OM ..... Object Management
OM ..... Object Model

OMT ..... Object Model Template
OS ..... Operating System
OSCI ..... Open SystemC Initiative
OSI ..... Open Systems Interconnection
OSI-RM ..... Open Systems Interconnection Reference Model
OWM ..... Ownership Management
PA-CAL ..... Pin Accurate Cycle-Approximate Level
PBD ..... Platform-based Design
PDE ..... Parallel Discrete Event
PDES ..... Parallel Discrete Event Simulation
PE ..... Processing Element
PtII ..... Ptolemy II
RAM ..... Random-Access Memory
RAW ..... Read-After-Write
RCCE ..... Rapidly Communicating Cores Environment
RMW ..... Read Modify Write
RO ..... Router
RT ..... Register Transfer
RTI ..... Runtime Infrastructure
RTIA ..... Runtime Infrastructure Ambassador
RTIG ..... Runtime Infrastructure Gateway
RTL ..... Register Transfer Level
RTOS ..... Real-time Operating System
SAR ..... Static Abstract Representation
SCC ..... Single-chip Cloud Computer
SCC ..... Strongly Connected Component
SDEM ..... Simulation Data Exchange Model
SDEMM ..... Simulation Data Exchange Metamodel
SDF ..... Synchronous Dataflow
SHM ..... Shared Memory
SIMD ..... Single Instruction Multiple Data
SISD ..... Single Instruction Single Data
SLD ..... System-Level Design
SLDL ..... System-Level Design Language
SM ..... State Machine
SoC ..... System-on-Chip
SOM ..... Simulation Object Model
SOMM ..... Simulation Object Metamodel
SW ..... Software
SWIG ..... Simplified Wrapper and Interface Generator
TAG ..... Time Advance Grant
THLP ..... Thread-Level Parallelism

TL	Transaction Level
TLM	Transaction Level Model
TLM-DT	Transaction Level Modeling with Distributed Time
TLP	Task-Level Parallelism
TM	Time Management
TNS	Test-and-Set
TP	Task Repository
UCM	Uncached Mode
UE	Unit of Execution
UMA	Unified Memory Access
UML	Unified Modeling Language
UT	Untimed
V2X	Vehicle-to-X
V2XC	Vehicle-to-X Communication
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VNI	Virtual Network Interface
VWI	Virtual Wireless Interface
WAR	Write-After-Read
WAW	Write-After-Write
WCB	Write-Combine Buffer
XML	Extensible Markup Language

Literatur- und Quellennachweise

[1] *Accelera Systems Initiative*. http://www.accellera.org/home/. [Online; accessed 11-July-2014].

[2] *Doxygen - Generate documentation from source code*. http://www.stack.nl/~dimitri/doxygen/. [Online; accessed 16-July-2014].

[3] *Forwardsim HLA Blockset for Simulink*. http://www.forwardsim.com/products/hla-blockset/. [Online; accessed 14-August-2014].

[4] *Forwardsim HLA Toolbox for MATLAB*. http://www.forwardsim.com/products/hla-toolbox/. [Online; accessed 14-August-2014].

[5] *GCC, the GNU Compiler Collection*. http://gcc.gnu.org/. [Online; accessed 16-July-2014].

[6] *IPG CarMaker*. http://ipg.de/de/simulationsolutions/carmaker/. [Online; accessed 28-August-2014].

[7] *KaSCPar - Karlsruhe SystemC parser suite*. http://kmir.de/downloads/sim/archives/kascpar-documentation.pdf. [Online; accessed 16-July-2014].

[8] *The MathWorks: Simulink Simulation and Model-Based Design*. http://www.mathworks.de/products/simulink/. [Online; accessed 15-August-2014].

[9] *OMG Model Driven Architecture*. http://www.omg.org/mda/. [Online; accessed 15-September-2014].

[10] *OVP - Open Virtual Platform*. http://www.ovpworld.org/. [Online; accessed 15-August-2014].

[11] *Pitch Make Your Systems Work Together*. http://www.pitch.se/products/prti. [Online; accessed 21-August-2014].

[12] *The poRTIco project*. http://www.porticoproject.org. [Online; accessed 21-August-2014].

[13] *PTV Vissim*. http://vision-traffic.ptvgroup.com/en-us/products/ptv-vissim/. [Online; accessed 28-August-2014].

[14] *PTV Visum*. http://vision-traffic.ptvgroup.com/en-us/products/ptv-visum/. [Online; accessed 28-August-2014].

[15] *RCCE: a Small Library for Many-Core Communication*. https://communities.

intel.com/servlet/JiveServlet/previewBody/5628-102-3-22522/RCCE_Specification.pdf. [Online; accessed 13-January-2014].

[16] *VT MÄC A company of VT Systems*. http://www.mak.com/products.html. [Online; accessed 21-August-2014].

[17] *IEEE Standard for Verilog Hardware Description Language*. IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001), 2006.

[18] *Behavioural languages - Part 6: VHDL Analog and Mixed-Signal Extensions*. IEEE Std 1076.1 IEC 61691-6 Edition 1.0 2009-12, S. 1–0, Dec 2009.

[19] *IEEE Standard VHDL Language Reference Manual*. IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002), S. c1–626, Jan 2009.

[20] *Nationale Roadmap Embedded Systems*. ZVEI Zentralverband Elektrotechnik und Elektronikindustrie e.V. Kompetenzzentrum Embedded Software & Systems, 2009.

[21] *IEEE Standard for Information technology– Local and metropolitan area networks– Specific requirements– Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 6: Wireless Access in Vehicular Environments*. IEEE Std 802.11p-2010 (Amendment to IEEE Std 802.11-2007 as amended by IEEE Std 802.11k-2008, IEEE Std 802.11r-2008, IEEE Std 802.11y-2008, IEEE Std 802.11n-2009, and IEEE Std 802.11w-2009), S. 1–51, July 2010.

[22] *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Framework and Rules*. IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000), S. 1–38, Aug 2010.

[23] *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)– Federate Interface Specification*. IEEE Std 1516.1-2010 (Revision of IEEE Std 1516.1-2000), S. 1–378, Aug 2010.

[24] *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)– Object Model Template (OMT) Specification*. IEEE Std 1516.2-2010 (Revision of IEEE Std 1516.2-2000), S. 1–110, Aug 2010.

[25] *IEEE Guide–Adoption of the Project Management Institute (PMI(R)) Standard A Guide to the Project Management Body of Knowledge (PMBOK(R) Guide)– Fourth Edition*. IEEE Std 1490-2011, S. 1–508, Nov 2011.

[26] *IEEE Recommended Practice for Distributed Simulation Engineering and Execution Process (DSEEP)*. IEEE Std 1730-2010 (Revision of IEEE Std 1516.3-2003), S. 1–79, Jan 2011.

[27] *IEEE Standard for Standard SystemC Language Reference Manual*. IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005), S. 1–638, 2012.

[28] *IEEE Standard for System and Software Verification and Validation*. IEEE Std

1012-2012 (Revision of IEEE Std 1012-2004), S. 1–223, May 2012.

[29] *Functional Mock-up Interface for Model Exchange and Co-Simulation V2.0*. Techn. Ber., MODELISAR consortium and Modelica Association Project "FMI", 2014.

[30] *IEEE Guide for Wireless Access in Vehicular Environments (WAVE) - Architecture*. IEEE Std 1609.0-2013, S. 1–78, March 2014.

[31] AARTS, E. und S. MARZANO: *The New Everyday: Views on Ambient Intelligence*. 010 Publishers, 2003.

[32] ACATECH: *Cyber-physical Systems: Driving Force for Innovation in Mobility, Health, Energy and Production*. Acatech position paper. Springer, 2011.

[33] ADAK, M., O. TOPÇU und H. OGUZTÜZÜN: *Model-based code generation for HLA federates*. Software: Practice and Experience, 40(2):149–175, 2010.

[34] AGRAWAL, H. und J. R. HORGAN: *Dynamic Program Slicing*. SIGPLAN Not., 25(6):246–256, Juni 1990.

[35] AKL, S. G.: *Parallel computation: models and methods*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[36] AMDAHL, G. M.: *Validity of the single processor approach to achieving large scale computing capabilities*. In: *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), S. 483–485, New York, NY, USA, 1967. ACM.

[37] ANDERSON, J., Y.-J. KIM und T. HERMAN: *Shared-memory mutual exclusion: major research trends since 1986*. Distributed Computing, 16(2-3):75–110, 2003.

[38] ASHTON, K.: *That Ïnternet of Things"Thing*. http://www.rfidjournal.com/articles/view?4986. [Online; accessed 17-October-2014].

[39] ATTARZADEH NIAKI, S. und I. SANDER: *Co-simulation of embedded systems in a heterogeneous MoC-based modeling framework*. In: *Industrial Embedded Systems (SIES), 2011 6th IEEE International Symposium on*, S. 238–247, June 2011.

[40] ATTARZADEH NIAKI, S. und I. SANDER: *Semi-formal refinement of heterogeneous embedded systems by foreign model integration*. In: *Specification and Design Languages (FDL), 2011 Forum on*, S. 1–8, Sept 2011.

[41] AUSTIN, T. M. und G. S. SOHI: *Dynamic Dependency Analysis of Ordinary Programs*. In: *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, S. 342–351, New York, NY, USA, 1992. ACM.

[42] AWAIS, M., P. PALENSKY, A. ELSHEIKH, E. WIDL und S. MATTHIAS: *The high level architecture RTI as a master to the functional mock-up interface components*. In: *Computing, Networking and Communications (ICNC), 2013 Inter-*

national Conference on, S. 315–320, Jan 2013.

[43] BAGRODIA, R. L. und M. TAKAI: *Performance Evaluation of Conservative Algorithms in Parallel Simulation Languages*. IEEE Trans. Parallel Distrib. Syst., 11(4):395–411, Apr. 2000.

[44] BAILEY, M. L., J. V. BRINER, JR. und R. D. CHAMBERLAIN: *Parallel logic simulation of VLSI systems*. ACM Comput. Surv., 26(3):255–294, Sep. 1994.

[45] BALARIN, F., Y. WATANABE, H. HSIEH, L. LAVAGNO, C. PASSERONE und A. SANGIOVANNI-VINCENTELLI: *Metropolis: an integrated electronic system design environment*. Computer, 36(4):45–52, April 2003.

[46] BARON, M.: *The Single-chip Cloud Computer*. http://www.MPRonline.com, 04 2010.

[47] BEAZLEY, D. M.: *Automated Scientific Software Scripting with SWIG*. Future Gener. Comput. Syst., 19(5):599–609, Juli 2003.

[48] BELL, C., D. BONACHEA, R. NISHTALA und K. YELICK: *Optimizing bandwidth limited problems using one-sided communication and overlap*. In: *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, S. 10 pp.–, April 2006.

[49] BENGEL, G., C. BAUN, M. KUNZE und K.-U. STUCKY: *Masterkurs Parallele und Verteilte Systeme*, 2008.

[50] BENINI, L. und G. DE MICHELI: *Networks on chips: a new SoC paradigm*. Computer, 35(1):70–78, Jan 2002.

[51] BERNARD, M., C. BUCKL, V. DOERICHT, M. FEHLING, L. FIEGE, H. VON GROLMAN, N. IVANDIC, C. JANELLO, C. KLEIN, K.-J. KUHN, C. PATZLAFF, B. C. RIEDL, B. SCHUETZ und C. STANEK: *Mehr Software (im) Wagen: Informations- und Kommunikationstechnik (IKT) als Motor der Elektromobilitaet der Zukunft*. Techn. Ber., fortiss GmbH, 2011.

[52] BERNER, D., J. PIERRE TALPIN, H. PATEL, D. A. MATHAIKUTTY und E. SHUKLA: *SystemCXML: An extensible SystemC front end using XML*. In: *In Proceedings of the Forum on specification and design languages (FDL*, 2005.

[53] BERTACCO, V., D. CHATTERJEE, N. BOMBIERI, F. FUMMI, S. VINCO, A. KAUSHIK und H. D. PATEL: *On the use of GP-GPUs for accelerating compute-intensive EDA applications*. In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, S. 1357–1366, March 2013.

[54] BINKERT, N., B. BECKMANN, G. BLACK, S. K. REINHARDT, A. SAIDI, A. BASU, J. HESTNESS, D. R. HOWER, T. KRISHNA, S. SARDASHTI, R. SEN, K. SEWELL, M. SHOAIB, N. VAISH, M. D. HILL und D. A. WOOD: *The Gem5 Simulator*. SIGARCH Comput. Archit. News, 39(2):1–7, Aug. 2011.

[55] BLACK, D. C. und J. DONOVAN: *SystemC: from the ground up*. Springer

Science+Business Media, LLC, 2004.

[56] BLAKE, G., R. DRESLINSKI und T. MUDGE: *A survey of multicore processors*. Signal Processing Magazine, IEEE, 26(6):26–37, November 2009.

[57] BLANC, N., D. KROENING und N. SHARYGINA: *Scoot: A Tool for the Analysis of SystemC Models*. In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, S. 467–470, Berlin, Heidelberg, 2008. Springer-Verlag.

[58] BLUMOFE, R. D. und C. E. LEISERSON: *Scheduling Multithreaded Computations by Work Stealing*. J. ACM, 46(5):720–748, Sep. 1999.

[59] BOMBIERI, N., G. DI GUGLIELMO, M. FERRARI, F. FUMMI, G. PRAVADELLI, F. STEFANNI und A. VENTURELLI: *HIFSuite: Tools for HDL Code Conversion and Manipulation*. EURASIP Journal on Embedded Systems, 2010(1):436328, 2010.

[60] BOMBIERI, N., F. FUMMI und D. QUAGLIA: *System/Network Design-space Exploration Based on TLM for Networked Embedded Systems*. ACM Trans. Embed. Comput. Syst., 9(4):37:1–37:32, Apr. 2010.

[61] BOMBIERI, N., F. FUMMI und S. VINCO: *On the automatic generation of GPU-oriented software applications from RTL IPs*. In: *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013 International Conference on*, S. 1–10, Sept 2013.

[62] BONONI, L., M. D. FELICE, G. D'ANGELO, M. BRACUTO und L. DONATIELLO: *MoVES: A framework for parallel and distributed simulation of wireless vehicular ad hoc networks*. Computer Networks, 52(1):155 – 179, 2008. (1) Performance of Wireless Networks (2) Synergy of Telecommunication and Broadcasting Networks.

[63] BORKAR, S.: *Thousand core chips: a technology perspective*. In: *Proceedings of the 44th annual Design Automation Conference*, DAC '07, S. 746–749, New York, NY, USA, 2007. ACM.

[64] BOUCHHIMA, F., G. NICOLESCU, E. ABOULHAMID und M. ABID: *Discrete-continuous simulation model for accurate validation in component-based heterogeneous SoC design*. In: *Rapid System Prototyping, 2005. (RSP 2005). The 16th IEEE International Workshop on*, S. 181–187, June 2005.

[65] BOUCHHIMA, F., G. NICOLESCU, E. M. ABOULHAMID und M. ABID: *Generic discrete/continuous simulation model for accurate validation in heterogeneous systems design*. Microelectronics Journal, 38:805 – 815, 2007.

[66] BRESLAU, L., D. ESTRIN, K. FALL, S. FLOYD, J. HEIDEMANN, A. HELMY, P. HUANG, S. MCCANNE, K. VARADHAN, Y. XU und H. YU: *Advances in network simulation*. Computer, 33(5):59–67, May 2000.

[67] BROOKS, C., E. A. LEE, X. LIU, S. NEUENDORFFER, Y. ZHAO und H. ZHENG: *Heterogeneous Concurrent Modeling and Design in Java (Volume 3: Ptolemy II Domains)*. Techn. Ber. UCB/EECS-2007-9, EECS Department, University of California, Berkeley, Jan 2007.

[68] BROOKS, III, E. D.: *The butterfly barrier*. Int. J. Parallel Program., 15(4):295–307, Okt. 1986.

[69] BRYANT, R. E.: *SIMULATION OF PACKET COMMUNICATION ARCHITECTURE COMPUTER SYSTEMS*. Techn. Ber., Cambridge, MA, USA, 1977.

[70] BUCK, J., S. HA, E. A. LEE und D. G. MESSERSCHMITT: *Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems*, 1992.

[71] BUCKL, C., A. CAMEK, G. KAINZ, C. SIMON, L. MERCEP, H. STAHLE und A. KNOLL: *The software car: Building ICT architectures for future electric vehicles*. In: *Electric Vehicle Conference (IEVC), 2012 IEEE International*, S. 1–8, 2012.

[72] BURTON, M., J. ALDIS, R. GÜNZEL und W. KLINGAUF: *Transaction Level Modelling: A reflection on what TLM is and how TLMs may be classified*. In: *Forum on specification and Design Languages, FDL 2007, September 18-20, 2007, Barcelona, Spain, Proceedings*, S. 92–97, 2007.

[73] BUTKO, A., R. GARIBOTTI, L. OST und G. SASSATELLI: *Accuracy evaluation of GEM5 simulator system*. In: *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*, S. 1–7, July 2012.

[74] CAI, L. und D. GAJSKI: *Transaction level modeling: an overview*. In: *Hardware/Software Codesign and System Synthesis, 2003. First IEEE/ACM/IFIP International Conference on*, S. 19–24, 2003.

[75] CARARA, E., R. DE OLIVEIRA, N. L. V. CALAZANS und F. MORAES: *HeMPS - a framework for NoC-based MPSoC generation*. In: *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on*, S. 1345–1348, 2009.

[76] CATALDO, A., E. A. LEE, X. LIU, E. D. MATSIKOUDIS und H. ZHENG: *A Constructive Fixed-Point Theorem and the Feedback Semantics of Timed Systems*. Techn. Ber. UCB/EECS-2006-4, EECS Department, University of California, Berkeley, Jan 2006.

[77] CESARIO, W., D. LYONNARD, G. NICOLESCU, Y. PAVIOT, S. YOO, A. JERRAYA, L. GAUTHIER und M. DIAZ-NAVA: *Multiprocessor SoC platforms: a component-based design approach*. Design Test of Computers, IEEE, 19(6):52–63, Nov 2002.

[78] CHANDY, K. und J. MISRA: *Distributed Simulation: A Case Study in Design and Verification of Distributed Programs*. Software Engineering, IEEE Tran-

sactions on, SE-5(5):440–452, 1979.

[79] CHANDY, K. M. und J. MISRA: *Asynchronous distributed simulation via a sequence of parallel computations*. Commun. ACM, 24(4):198–206, Apr. 1981.

[80] CHANG, X.: *Network Simulations with OPNET*. In: *Proceedings of the 31st Conference on Winter Simulation: Simulation—a Bridge to the Future - Volume 1*, WSC '99, S. 307–314, New York, NY, USA, 1999. ACM.

[81] CHEN, J., M. ANNAVARAM und M. DUBOIS: *SlackSim: A Platform for Parallel Simulations of CMPs on CMPs*. SIGARCH Comput. Archit. News, 37(2):20–29, Juli 2009.

[82] CHEN, W. und R. DOEMER: *A Distributed Parallel Simulator for Transaction Level Models with Relaxed Timing*. In: *Center for Embedded Computer Systems, University of California, Technical Report*, 2011.

[83] CHEN, W. und R. DÖMER: *Optimized Out-of-order Parallel Discrete Event Simulation Using Predictions*. In: *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, S. 3–8, San Jose, CA, USA, 2013. EDA Consortium.

[84] CHEN, W., X. HAN und R. DOEMER: *Multicore Simulation of Transaction-Level Models Using the SoC Environment*. IEEE Design&Test of Computers, 28(3):20–31, 2011.

[85] CHEN, W., X. HAN und R. DOMER: *Out-of-order parallel simulation for ESL design*. In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, S. 141–146, March 2012.

[86] CHOPARD, B., P. COMBES und J. ZORY: *A Conservative Approach to Systemc Parallelization*. In: *Proceedings of the 6th International Conference on Computational Science - Volume Part IV*, ICCS'06, S. 653–660, Berlin, Heidelberg, 2006. Springer-Verlag.

[87] CHURCH, A.: *An Unsolvable Problem of Elementary Number Theory*. American Journal of Mathematics, 58(2):345–363, April 1936.

[88] CLANG-PROJECT: *clang: a C language family frontend for LLVM*. http://clang.llvm.org/. [Online; accessed 21-March-2014].

[89] CLAUSS, C., S. LANKES, T. BEMMERL, J. GALOWICZ und S. PICKARTZ: *iRCCE: A Non-blocking Communication Extension to the RCCE Communication Library for the Intel Single-Chip Cloud Computer*. Chair for Operating Systems, RWTH Aachen University, 2011.

[90] COMBES, P., E. CARON, F. DESPREZ, B. CHOPARD und J. ZORY: *Relaxing Synchronization in a Parallel SystemC Kernel*. In: *Parallel and Distributed Processing with Applications, 2008. ISPA '08. International Symposium on*, S. 180–187, 2008.

[91] CORPORATION, I.: *Intel Xeon Phi Produktreihe*. http://www.intel.de/content/www/de/de/processors/xeon/xeon-phi-detail.html. [Online; accessed 12-August-2014].

[92] CORPORATION, I.: *SCC External Architecture Specification (EAS) Revision 1.1*. Techn. Ber., Intel Labs, 2010.

[93] COX, D. R.: *RITSim: Distributed SystemC Simulation*. Diplomarbeit, Rochester Institute of Technology, 2005.

[94] DAVARE, A., D. DENSMORE, L. GUO, R. PASSERONE, A. L. SANGIOVANNI-VINCENTELLI, A. SIMALATSAR und Q. ZHU: *metroII: A Design Environment for Cyber-physical Systems*. ACM Trans. Embed. Comput. Syst., 12(1s):49:1–49:31, März 2013.

[95] DAVARE, A., D. DENSMORE, T. MEYEROWITZ, A. PINTO, A. SANGIOVANNI-VINCENTELLI, G. YANG, H. ZENG und Q. ZHU: *A Next-Generation Design Framework for Platform-based Design*. In: *DVCon 2007*, February 2007.

[96] DENG, F. und J. JONES: *Weighted System Dependence Graph*. In: *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, S. 380–389, April 2012.

[97] DINAN, J., S. OLIVIER, G. SABIN, J. PRINS, P. SADAYAPPAN und C.-W. TSENG: *Dynamic Load Balancing of Unbalanced Computations Using Message Passing*. In: *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, S. 1–8, March 2007.

[98] DOEMER, R., A. GERSTLAUER und D. GAJSKI: *SpecC Language Reference Manual, Version 2.0*. Techn. Ber., SpecC Technology Open Consortium, 2002.

[99] DOMER, R., A. GERSTLAUER und W. MULLER: *Introduction to Hardware-dependent Software design*. In: *Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific*, S. 290–292, 2009.

[100] DONLIN, A.: *Transaction level modeling: flows and use models*. In: *Hardware/-Software Codesign and System Synthesis, 2004. CODES + ISSS 2004. International Conference on*, S. 75–80, 2004.

[101] EGGERS, S., J. EMER, H. LEBY, J. LO, R. STAMM und D. TULLSEN: *Simultaneous multithreading: a platform for next-generation processors*. Micro, IEEE, 17(5):12–19, 1997.

[102] EKER, J., J. JANNECK, E. LEE, J. LIU, X. LIU, J. LUDVIG, S. NEUENDORFFER, S. SACHS und Y. XIONG: *Taming heterogeneity - the Ptolemy approach*. Proceedings of the IEEE, 91(1):127–144, Jan 2003.

[103] EL-REWINI, H. und M. ABD-EL-BARR: *Advanced Computer Architecture*

and Parallel Processing (Wiley Series on Parallel and Distributed Computing). Wiley-Interscience, 2005.

[104] ENGBLOM, J.: *Multicore pain (and gain) from a Virtual Platform Perspective*. In: *Second Swedish Workshop on Multicore Computing*, 2009.

[105] EZUDHEEN, P., P. CHANDRAN, J. CHANDRA, B. SIMON und D. RAVI: *Parallelizing SystemC Kernel for Fast Hardware Simulation on SMP Machines*. In: *Principles of Advanced and Distributed Simulation, 2009. PADS '09. ACM/IEEE/SCS 23rd Workshop on*, S. 80–87, June 2009.

[106] FERNANDEZ, M.: *Models of Computation: An Introduction to Computability Theory*. Springer Publishing Company, Incorporated, 1st Aufl., 2009.

[107] FERRARI, A. und A. SANGIOVANNI-VINCENTELLI: *System design: traditional concepts and new paradigms*. In: *Computer Design, 1999. (ICCD '99) International Conference on*, S. 2–12, 1999.

[108] FERSCHA, A. und S. K. TRIPATHI: *Parallel and distributed simulation of discrete event systems*. Techn. Ber., College Park, MD, USA, 1994.

[109] FLYNN, M.: *Very high-speed computing systems*. Proceedings of the IEEE, 54(12):1901–1909, 1966.

[110] FORUM, M. P. I.: *MPI: A Message-Passing Interface Standard Version 3.0*, 09 2012. Chapter author for Collective Communication, Process Topologies, and One Sided Communications.

[111] FOSTER, I.: *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[112] FUJIMOTO, R. M.: *Parallel discrete event simulation*. Commun. ACM, 33(10):30–53, Okt. 1990.

[113] FUJIMOTO, R. M.: *Parallel and Distribution Simulation Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1st Aufl., 1999.

[114] GAJSKI, D. D. (Hrsg.): *SPECC : specification language and methodology*. Kluwer Academic Publishers, Boston, 2. pr. Aufl., 2001. Includes bibliographical references and index; : £80.00.

[115] GAJSKI, D. D., S. ABDI, A. GERSTLAUER und G. SCHIRNER: *Embedded System Design: Modeling, Synthesis and Verification*. Springer Publishing Company, Incorporated, 1st Aufl., 2009.

[116] GAJSKI, D. D. und R. H. KUHN: *New VLSI Tools*. Computer, 16(12):11–14, Dez. 1983.

[117] GAMMA, E., R. HELM, R. JOHNSON und J. VLISSIDES: *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[118] GEM5-PROJECT: *Parallel M5*. http://m5sim.org/Parallel_M5. [Online; accessed 07-September-2014].

[119] GERSTLAUER, A. (Hrsg.): *System design : a practical guide with SpecC*. Kluwer, Boston [u.a.], 2001.

[120] GERSTLAUER, A., C. HAUBELT, A. PIMENTEL, T. STEFANOV, D. GAJSKI und J. TEICH: *Electronic System-Level Synthesis Methodologies*. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 28(10):1517–1530, 2009.

[121] GLAS, B.: *Trusted computing für adaptive Automobilsteuergeräte im Umfeld der Inter-Fahrzeug-Kommunikation*. Steinbuch series on advances in information technology. KIT Scientific Publ., 2011.

[122] GODERIS, A., C. BROOKS, I. ALTINTAS, E. A. LEE und G. CAROL: *Composing Different Models of Computation in Kepler and Ptolemy II*. In: *2007 Proceedings*, S. 182–190. International Conference on Computational Science (ICCS), May 2007. A later version has been <a href="http://chess.eecs.berkeley.edu/pubs/533.html»published in Future Generation Computer Systems (FGCS), Elsevier.

[123] GOMBERT, B.: *From the intelligent wheel bearing to the "robot wheel"*. In: *Schaeffler Symposium Book*. Schaeffler Technologies GmbH & Co. KG, 2010.

[124] GRAHAM, S. L., P. B. KESSLER und M. K. MCKUSICK: *Gprof: A Call Graph Execution Profiler*. In: *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, S. 120–126, New York, NY, USA, 1982. ACM.

[125] GROUP, S. B. O. M. P. D.: *Base Object Model (BOM) Template Specification*. Techn. Ber., Simulation Interoperability and Standards Organization, 2006.

[126] GROUP, S. B. O. M. P. D.: *Guide for Base Object Model (BOM) Use and Implementation*. Techn. Ber., Simulation Interoperability and Standards Organization, 2006.

[127] HANSMANN, U.: *Pervasive Computing: The Mobile World*. Springer Professional Computing. Springer, 2003.

[128] HAREL, D. und B. RUMPE: *Meaningful modeling: what's the semantics of ßemantics"?*. Computer, 37(10):64–72, Oct 2004.

[129] HARPER, P. R.: *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2012.

[130] HAUBELT, C.: *Digitale Hardware/Software-Systeme: Spezifikation und Verifikation*. eXamen-press. Springer, 2010.

[131] HEMINGWAY, G., H. NEEMA, H. NINE, J. SZTIPANOVITS und G. KARSAI: *Rapid Synthesis of High-level Architecture-based Heterogeneous Simulation: A Model-based Integration Approach*. Simulation, 88(2):217–232, Feb. 2012.

[132] HENDERSON, T. R., S. ROY, S. FLOYD und G. F. RILEY: *Ns-3 Project Goals*. In: *Proceeding from the 2006 Workshop on Ns-2: The IP Network Simulator*, WNS2 '06, New York, NY, USA, 2006. ACM.

[133] HENNESSY, J. und D. PATTERSON: *Computer Architecture: A Quantitative Approach, Fifth Edition*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2011.

[134] HENZINGER, T.: *The theory of hybrid automata*. In: *Logic in Computer Science, 1996. LICS '96. Proceedings., Eleventh Annual IEEE Symposium on*, S. 278–292, Jul 1996.

[135] HENZINGER, T., B. HOROWITZ und C. KIRSCH: *Giotto: a time-triggered language for embedded programming*. Proceedings of the IEEE, 91(1):84–99, Jan 2003.

[136] HERRERA, F. und E. VILLAR: *A Framework for Heterogeneous Specification and Design of Electronic Embedded Systems in SystemC*. ACM Trans. Des. Autom. Electron. Syst., 12(3):22:1–22:31, Mai 2008.

[137] HERZBERG, D.: *Modeling Telecommunication Systems: From Standards to System Architectures*. Doktorarbeit, Rheinisch-Westfaelische Technische Hochschule Aachen, 2003.

[138] HERZBERG, D. und M. BROY: *Modeling layered distributed communication systems*. Form. Asp. Comput., 17(1):1–18, Mai 2005.

[139] HILL, M., N. JOUPPI und G. SOHI: *Readings in computer architecture*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, 2000.

[140] HÖHN, R., A. RAUSCH, S. HÖPPNER und K. BERGNER: *Das V-Modell XT: Grundlagen, Methodik und Anwendungen*. Springer-Lehrbuch. Springer, 2008.

[141] HORWITZ, S., T. REPS und D. BINKLEY: *Interprocedural Slicing Using Dependence Graphs*. In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, S. 35–46, New York, NY, USA, 1988. ACM.

[142] HOSSEINABADY, M. und J. NUNEZ-YANEZ: *SystemC architectural transaction level modelling for large NoCs*. In: *Specification Design Languages (FDL 2010), 2010 Forum on*, S. 1–6, Sept 2010.

[143] HOSSEINABADY, M. und J. NUNEZ-YANEZ: *Fast and low overhead architectural transaction level modelling for large-scale network-on-chip simulation*. Computers Digital Techniques, IET, 6(6):384–395, November 2012.

[144] HOWARD, J., S. DIGHE, S. VANGAL, G. RUHL, N. BORKAR, S. JAIN, V. ERRAGUNTLA, M. KONOW, M. RIEPEN, M. GRIES, G. DROEGE, T. LUND-

LARSEN, S. STEIBL, S. BORKAR, V. DE und R. VAN DER WIJNGAART: *A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and DVFS for Performance and Power Scaling*. Solid-State Circuits, IEEE Journal of, 2011.

[145] HUANG, K., S. IL HAN, K. POPOVICI, L. BRISOLARA, X. GUERIN, L. LI, X. YAN, S.-I. CHAE, L. CARRO und A. JERRAYA: *Simulink-Based MPSoC Design Flow: Case Study of Motion-JPEG and H.264*. In: *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, S. 39–42, June 2007.

[146] ISERMANN, R., R. SCHWARZ und S. STOLZL: *Fault-tolerant drive-by-wire systems*. Control Systems, IEEE, 22(5):64–81, Oct 2002.

[147] *Information Technology — Open Systems Interconnection — Basic Reference Model: The Basic Model*. ISO/IEC 7498-1:1994, ISO, Geneva, Switzerland, Nov. 1994.

[148] ITRS: *International Technology Roadmap for Semiconductors*. Techn. Ber., 2011.

[149] JAFER, S., Q. LIU und G. WAINER: *Synchronization methods in parallel and distributed discrete-event simulation*. Simulation Modelling Practice and Theory, 30(0):54 – 73, 2013.

[150] JEFFERSON, D., B. BECKMAN, F. WIELAND, L. BLUME und M. DILORETO: *Time warp operating system*. In: *Proceedings of the eleventh ACM Symposium on Operating systems principles*, SOSP '87, S. 77–93, New York, NY, USA, 1987. ACM.

[151] JOVEN, J., O. FONT-BACH, D. CASTELLS-RUFAS, R. MARTINEZ, L. TERES und J. CARRABINA: *xENoC - An eXperimental Network-On-Chip Environment for Parallel Distributed Computing on NoC-based MPSoC Architectures*. In: *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, PDP '08, S. 141–148, Washington, DC, USA, 2008. IEEE Computer Society.

[152] JUNGNICKEL, D.: *Graphs, Networks and Algorithms*. Springer Publishing Company, Incorporated, 3rd Aufl., 2007.

[153] KAHN, G.: *The Semantics of Simple Language for Parallel Programming.*. In: *IFIP Congress*, S. 471–475, 1974.

[154] KARSAI, G., A. LANG und S. NEEMA: *Design patterns for open tool integration*. Software & Systems Modeling, 4(2):157–170, 2005.

[155] KARYPIS, G.: *METIS A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*. Department of Computer Science & Engineering University of Minnesota, Minneapolis, MN 55455, Version 5.1.0 Aufl., March, 30 2013.

[156] KARYPIS, G. und V. KUMAR: *A Fast and High Quality Multilevel Scheme*

for Partitioning Irregular Graphs. SIAM J. Sci. Comput., 20(1):359–392, Dez. 1998.

[157] KARYPIS, G. und V. KUMAR: *Multilevel K-way Partitioning Scheme for Irregular Graphs*. J. Parallel Distrib. Comput., 48(1):96–129, Jan. 1998.

[158] KAUSHIK, A. und H. D. PATEL: *Systemc-clang: An open-source framework for analyzing mixed-abstraction SystemC models*. In: *Specification Design Languages (FDL), 2013 Forum on*, S. 1–8, Sept 2013.

[159] KESTING, A.: *Microscopic Modeling of Human and Automated Driving: Towards Traffic-Adaptive Cruise Control*. Doktorarbeit, Faculty of Traffic Sciences "Friedrich List" Technische Universität Dresden (Germany), 2008.

[160] KESTING, A.: *Microscopic Modeling of Human and Automated Driving: Towards Traffic-Adaptive Cruise Control*. Doktorarbeit, Technische Universität Dresden, 2008.

[161] KESTING, A. und M. TREIBER: *How Reaction Time, Update Time, and Adaptation Time Influence the Stability of Traffic Flow*. Computer-Aided Civil and Infrastructure Engineering, 23(2):125–137, 2008.

[162] KESTING, A., M. TREIBER, M. SCHÖNHOF und D. HELBING: *Adaptive cruise control design for active congestion avoidance*. Transportation Research Part C: Emerging Technologies, 16(6):668 – 683, 2008.

[163] KEUTZER, K., A. NEWTON, J. RABAEY und A. SANGIOVANNI-VINCENTELLI: *System-level design: orthogonalization of concerns and platform-based design*. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 19(12):1523–1543, 2000.

[164] KIM, H., L. GUO, E. LEE und A. SANGIOVANNI-VINCENTELLI: *A tool integration approach for architectural exploration of aircraft electric power systems*. In: *Cyber-Physical Systems, Networks, and Applications (CPSNA), 2013 IEEE 1st International Conference on*, S. 38–43, Aug 2013.

[165] KÖPKE, A., M. SWIGULSKI, K. WESSEL, D. WILLKOMM, P. T. K. HANEVELD, T. E. V. PARKER, O. W. VISSER, H. S. LICHTE und S. VALENTIN: *Simulating Wireless and Mobile Networks in OMNeT++ the MiXiM Vision*. In: *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, Simutools '08, S. 71:1–71:8, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[166] KRAJZEWICZ, D., J. ERDMANN, M. BEHRISCH und L. BIEKER: *Recent Development and Applications of SUMO - Simulation of Urban MObility*. International Journal On Advances in Systems and Measurements, 5(3&4):128–138, December 2012.

[167] KRANKE, F. und H.-J. STAUSS: *Forschungsinitiative INVENT: Intelligenter Verkehr und nutzergerechte Technik. Schlussbericht VM 2010 zu dem Teilprojekt VLA - Verkehrsleitungsassistenz*. Techn. Ber., Volkswagen AG, 2006.

[168] KUMAR, V., L. LIN, D. KRAJZEWICZ, F. HRIZI, O. MARTINEZ, J. GOZALVEZ und R. BAUZA: *iTETRIS: Adaptation of ITS Technologies for Large Scale Integrated Simulation*. In: *Vehicular Technology Conference (VTC 2010-Spring), 2010 IEEE 71st*, S. 1–5, May 2010.

[169] LANTZ, R.: *Parallel SimOS: Scalability and Performance for Large System Simulation*. Doktorarbeit, Stanford University, 2007.

[170] LÄSCHE, C., V. GOLLÜCKE und A. HAHN: *Using an HLA Simulation Environment for Safety Concept Verification of Offshore Operations*. In: *Proceediings 27th European Conference on Modelling and Simulation*, S. 156ff, 05 2013.

[171] LASNIER, G., J. CARDOSO, P. SIRON, C. PAGETTI und P. DERLER: *Distributed Simulation of Heterogeneous and Real-Time Systems*. In: *Distributed Simulation and Real Time Applications (DS-RT), 2013 IEEE/ACM 17th International Symposium on*, S. 55–62, Oct 2013.

[172] LAVAGNO, L., L. SCHEFFER und G. MARTIN: *EDA for IC Implementation, Circuit Design, and Process Technology*. Electronic Design Automation for Integrated Circuits Hdbk. Taylor & Francis, 2006.

[173] LAW, A. M. und W. D. KELTON: *Simulation Modeling and Analysis*. McGraw-Hill Higher Education, 2nd Aufl., 1997.

[174] LEE, E.: *Cyber Physical Systems: Design Challenges*. In: *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, S. 363–369, May 2008.

[175] LEE, E. und A. SANGIOVANNI-VINCENTELLI: *A framework for comparing models of computation*. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 17(12):1217–1229, Dec 1998.

[176] LEE, E. und A. SANGIOVANNI-VINCENTELLI: *Component-based design for the future*. In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, S. 1–5, March 2011.

[177] LEE, E. A. und S. NEUENDORFFER: *MoML - A Modeling Markup Language in XML - Version 0.4*, 2000.

[178] LEE, E. A., S. NEUENDORFFER und M. J. WIRTHLIN: *Actor-Oriented Design Of Embedded Hardware And Software Systems*. Journal of Circuits, Systems, and Computers, 12:231–260, 2003.

[179] LEE, E. A. und H. ZHENG: *Operational semantics of hybrid systems*. In: *Hybrid Systems: Computation and Control (HSCC), volume LNCS 3414*, S. 25–53. Springer-Verlag, 2005.

[180] LEUPERS, R., L. EECKHOUT, G. MARTIN, F. SCHIRRMEISTER, N. TOPHAM und X. CHEN: *Virtual Manycore platforms: Moving towards 100+ processor cores*. In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, S. 1–6, 2011.

[181] LI, T., Y. GUO und S. KUN LI: *Design and implementation of a parallel Verilog simulator: PVSim*. In: *VLSI Design, 2004. Proceedings. 17th International Conference on*, S. 329–334, 2004.

[182] LIS, M., P. REN, M. H. CHO, K. S. SHIM, C. FLETCHER, O. KHAN und S. DEVADAS: *Scalable, accurate multicore simulation in the 1000-core era*. In: *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, S. 175–185, April 2011.

[183] LIS, M., K. S. SHIM, M. H. CHO, P. REN, O. KHAN und S. DEVADAS: *DARSIM: a parallel cycle-level NoC simulator*. In: EECKHOUT, L. und T. WENISCH (Hrsg.): *MoBS 2010 - Sixth Annual Workshop on Modeling, Benchmarking and Simulation*, Saint Malo, France, 2010.

[184] LIU, J., X. WU und E. LEE: *Interoperation of Heterogeneous CAD Tools in Ptolemy II*. In: *Symposium on Design, Test, and Microfabrication of MEMS/-MOEMS*, 1999.

[185] LIU, X., Y. XIONG und E. LEE: *The ptolemy II framework for visual languages*. In: *Human-Centric Computing Languages and Environments, 2001. Proceedings IEEE Symposia on*, S. 50–51, 2001.

[186] LLVM-PROJECT: *The LLVM Compiler Infrastructure*. http://llvm.org/. [Online; accessed 21-March-2014].

[187] LUBACHEVSKY, B. D.: *Efficient distributed event-driven simulations of multiple-loop networks*. Commun. ACM, 32(1):111–123, Jan. 1989.

[188] LUNGEANU, D. und C.-J. R. SHI: *Parallel and Distributed VHDL Simulation*. In: *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '00, S. 658–662, New York, NY, USA, 2000. ACM.

[189] LYONNARD, D., S. YOO, A. BAGHDADI und A. JERRAYA: *Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip*. In: *Design Automation Conference, 2001. Proceedings*, S. 518–523, 2001.

[190] MALER, O., Z. MANNA und A. PNUELI: *From Timed to Hybrid Systems*. In: *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, S. 447–484, London, UK, UK, 1992. Springer-Verlag.

[191] MARIA, A.: *Introduction to modeling and simulation*. In: *Proceedings of the 29th conference on Winter simulation*, WSC '97, S. 7–13, Washington, DC, USA, 1997. IEEE Computer Society.

[192] MARQUET, K., B. KARKARE und M. MOY: *A Theoretical and Experimental Review of SystemC Front-ends*. In: *FDL*, S. 124–129, 2010.

[193] MARQUET, K. und M. MOY: *PinaVM: A systemC Front-end Based on an Executable Intermediate Representation*. In: *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '10, S. 79–88, New York, NY, USA, 2010. ACM.

[194] MARTIN, M. M. K., D. J. SORIN, B. M. BECKMANN, M. R. MARTY, M. XU, A. R. ALAMELDEEN, K. E. MOORE, M. D. HILL und D. A. WOOD: *Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset*. SIGARCH Comput. Archit. News, 33(4):92–99, Nov. 2005.

[195] MARWEDEL, P.: *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems*. Embedded Systems. Springer, 2010.

[196] MATTERN, F.: *Efficient algorithms for distributed snapshots and global virtual time approximation*. J. Parallel Distrib. Comput., 18(4):423–434, Aug. 1993.

[197] MATTERN, F. und C. FLÖRKEMEIER: *Vom Internet der Computer zum Internet der Dinge*. Informatik-Spektrum, 33(2):107–121, 2010.

[198] MATTSON, T., R. VAN DER WIJNGAART, M. RIEPEN, T. LEHNIG, P. BRETT, W. HAAS, P. KENNEDY, J. HOWARD, S. VANGAL, N. BORKAR, G. RUHL und S. DIGHE: *The 48-core SCC Processor: the Programmer's View*. In: *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, S. 1–11, 2010.

[199] MELLO, A., I. MAIA, A. GREINER und F. PECHEUX: *Parallel simulation of systemC TLM 2.0 compliant MPSoC on SMP workstations*. In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, S. 606–609, March 2010.

[200] MENDOZA CERVANTES, F.: *A Problem-Oriented Approach for Dynamic Verification of Heterogeneous Embedded Systemss*. Doktorarbeit, Karlsruhe Institute of Technology (KIT), Karlsruhe, 2014. Zugl.: Karlsruhe, KIT, Diss., 2013.

[201] MILLER, J., H. KASTURE, G. KURIAN, C. GRUENWALD, N. BECKMANN, C. CELIO, J. EASTEP und A. AGARWAL: *Graphite: A distributed parallel simulator for multicores*. In: *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, S. 1–12, Jan 2010.

[202] MOORE, G.: *Cramming More Components Onto Integrated Circuits*. Proceedings of the IEEE, 86(1):82–85, 1998.

[203] MORAES, F., N. CALAZANS, A. MELLO, L. MÖLLER und L. OST: *HERMES: an infrastructure for low area overhead packet-switching networks on chip*. Integration, the {VLSI} Journal, 38(1):69 – 93, 2004.

[204] MOY, M., F. MARANINCHI und L. MAILLET-CONTOZ: *Pinapa: An Extrac-*

tion Tool for SystemC Descriptions of Systems-on-a-chip. In: *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT '05, S. 317–324, New York, NY, USA, 2005. ACM.

[205] NANJUNDAPPA, M., H. PATEL, B. JOSE und S. SHUKLA: *SCGPSim: A fast SystemC simulator on GPUs*. In: *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, S. 149–154, Jan 2010.

[206] NAROSKA, E.: *Parallel VHDL Simulation*. In: *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '98, S. 159–165, Washington, DC, USA, 1998. IEEE Computer Society.

[207] NICOL, D. M.: *The Cost of Conservative Synchronization in Parallel Discrete Event Simulations*. J. ACM, 40(2):304–333, Apr. 1993.

[208] NITZBERG, B. und V. LO: *Distributed Shared Memory: A Survey of Issues and Algorithms*. Computer, 24(8):52–60, Aug. 1991.

[209] NOULARD, E., J.-Y. ROUSSELOT und P. SIRON: *CERTI, an Open Source RTI, why and how*. In: *Spring Simulation Interoperability Workshop*, S. pp. 1–11, San Diego, United States, 2009.

[210] OTTENSTEIN, K. J. und L. M. OTTENSTEIN: *The Program Dependence Graph in a Software Development Environment*. SIGPLAN Not., 19(5):177–184, Apr. 1984.

[211] PADUA, D. A. (Hrsg.): *Encyclopedia of Parallel Computing*. Springer, 2011.

[212] PATEL, H. und S. SHUKLA: *Towards a heterogeneous simulation kernel for system level models: a SystemC kernel for synchronous data flow models*. In: *VLSI, 2004. Proceedings. IEEE Computer society Annual Symposium on*, S. 241–242, Feb 2004.

[213] PAULIN, P., C. PILKINGTON und E. BENSOUDANE: *StepNP: A System-Level Exploration Platform for Network Processors*. IEEE Des. Test, 19(6):17–26, Nov. 2002.

[214] PEETERS, J., N. VENTROUX, T. SASSOLAS und L. LACASSAGNE: *A systemc TLM framework for distributed simulation of complex systems with unpredictable communication*. In: *Design and Architectures for Signal and Image Processing (DASIP), 2011 Conference on*, S. 1–8, Nov 2011.

[215] PENRY, D., D. FAY, D. HODGDON, R. WELLS, G. SCHELLE, D. AUGUST und D. CONNORS: *Exploiting parallelism and structure to accelerate the simulation of chip multi-processors*. In: *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, S. 29–40, Feb 2006.

[216] PIÓRKOWSKI, M., M. RAYA, A. L. LUGO, P. PAPADIMITRATOS, M. GROSSGLAUSER und J.-P. HUBAUX: *TraNS: Realistic Joint Traffic and Network Simulator for VANETs*. SIGMOBILE Mob. Comput. Commun. Rev., 12(1):31–33,

Jan. 2008.

[217] PTOLEMAEUS, C. (Hrsg.): *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[218] QUECK, T., B. SCHUNEMANN, I. RADUSCH und C. MEINEL: *Realistic Simulation of V2X Communication Scenarios*. In: *Asia-Pacific Services Computing Conference, 2008. APSCC '08. IEEE*, S. 1623–1627, Dec 2008.

[219] RADETZKI, M. und R. S. KHALIGH: *Modelling Alternatives for Cycle Approximate Bus TLMs*. In: *FDL*, S. 74–79, 2007.

[220] RHOADS, S.: *Plasma CPU*. http://opencores.org/project,plasma. [Online; accessed 27-February-2014].

[221] RONDINONE, M., J. MANEROS, D. KRAJZEWICZ, R. BAUZA, P. CATALDI, F. HRIZI, J. GOZALVEZ, V. KUMAR, M. RÖCKL, L. LIN, O. LAZARO, J. LEGUAY, J. HÄRRI, S. VAZ, Y. LOPEZ, M. SEPULCRE, M. WETTERWALD, R. BLOKPOEL und F. CARTOLANO: *iTETRIS: A modular simulation platform for the large scale evaluation of cooperative {ITS} applications*. Simulation Modelling Practice and Theory, 34(0):99 – 125, 2013.

[222] ROTTA, R.: *On Efficient Message Passing on the Intel SCC*. In: DIANA GÖHRINGER, MICHAEL HÜBNER, J. B. (Hrsg.): *3rd Many-core Applications Research Community (MARC) Symposium*. KIT Scientific Publishing, 2011.

[223] ROTTA, R., T. PRESCHER, J. TRAUE und J. NOLTE: *Data Sharing Mechanisms for Parallel Graph Algorithms on the Intel SCC*. In: NOULARD, E. und S. VERNHES (Hrsg.): *Proceedings of the 6th Many-core Applications Research Community (MARC) Symposium*, S. 13–18, Toulouse, France, Juli 2012. ONERA, The French Aerospace Lab.

[224] ROWSON, J. A. und A. SANGIOVANNI-VINCENTELLI: *Interface-based design*. In: *Proceedings of the 34th annual Design Automation Conference*, DAC '97, S. 178–183, New York, NY, USA, 1997. ACM.

[225] SALIHUNDAM, P., S. JAIN, T. JACOB, S. KUMAR, V. ERRAGUNTLA, Y. HOSKOTE, S. VANGAL, G. RUHL, P. KUNDU und N. BORKAR: *A 2Tb/s 6x4 mesh network with DVFS and 2.3Tb/s/W router in 45nm CMOS*. In: *VLSI Circuits (VLSIC), 2010 IEEE Symposium on*, S. 79–80, 2010.

[226] SANDER, I. und A. JANTSCH: *System modeling and transformational design refinement in ForSyDe [formal system design]*. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 23(1):17–32, Jan 2004.

[227] SANDER, O.: *Skalierbare adaptive System-on-Chip-Architekturen für Inter-Car und Intra-Car Kommunikationsgateways*. Steinbuch series on advances in information technology. KIT Scientific Publ., 2010.

[228] SANGIOVANNI-VINCENTELLI, A.: *Quo Vadis, SLD? Reasoning About the Trends and Challenges of System Level Design*. Proceedings of the IEEE, 95(3):467–506, 2007.

[229] SANGIOVANNI-VINCENTELLI, A. und M. DI NATALE: *Embedded System Design for Automotive Applications*. Computer, 40(10):42–51, Oct 2007.

[230] SANGIOVANNI-VINCENTELLI, A. und G. MARTIN: *Platform-based design and software design methodology for embedded systems*. Design Test of Computers, IEEE, 18(6):23–33, 2001.

[231] SCHALLER, R.: *Moore's law: past, present and future*. Spectrum, IEEE, 34(6):52–59, Jun 1997.

[232] SCHUMACHER, C., R. LEUPERS, D. PETRAS und A. HOFFMANN: *parSC: Synchronous Parallel Systemc Simulation on Multi-core Host Architectures*. In: *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10, S. 241–246, New York, NY, USA, 2010. ACM.

[233] SCHUMACHER, C., J. WEINSTOCK, R. LEUPERS und G. ASCHEID: *Cause and effect of nondeterministic behavior in sequential and parallel SystemC simulators*. In: *High Level Design Validation and Test Workshop (HLDVT), 2012 IEEE International*, S. 124–131, 2012.

[234] SCHUMACHER, C., J. WEINSTOCK, R. LEUPERS, G. ASCHEID, L. TOSORATTO, A. LONARDO, D. PETRAS und T. GROTKER: *legaSCi: Legacy SystemC Model Integration into Parallel Systemc Simulators*. In: *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, S. 2188–2193, May 2013.

[235] SCHUTZ, J.: *A 3.3V 0.6 /spl mu/m BiCMOS superscalar microprocessor*. In: *Solid-State Circuits Conference, 1994. Digest of Technical Papers. 41st ISSCC., 1994 IEEE International*, S. 202–203, 1994.

[236] SHASHA, D. und M. SNIR: *Efficient and correct execution of parallel programs that share memory*. ACM Trans. Program. Lang. Syst., 10(2):282–312, Apr. 1988.

[237] SINHA, R., A. PRAKASH und H. PATEL: *Parallel simulation of mixed-abstraction SystemC models on GPUs and multicore CPUs*. In: *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, S. 455–460, Jan 2012.

[238] SOMMER, C.: *Veins The Open Source Vehicular Network Simulation Framework*. http://veins.car2x.org/. [Online; accessed 23-June-2014].

[239] SOMMER, C., R. GERMAN und F. DRESSLER: *Bidirectionally Coupled Network and Road Traffic Simulation for Improved IVC Analysis*. Mobile Computing, IEEE Transactions on, 10(1):3–15, Jan 2011.

[240] STACHOWIAK, H.: *Allgemeine Modelltheorie*. Springer-Verlag, 1973.

[241] STEINMANN, J.: *SPEEDES: Synchronous Parallel Environment for Emulation and Discrete-Event Simulation*. In: *SCS Western Multiconference on Advances in Parallel and Distributed Simulation (PADS91)*, 1991.

[242] SUNG, C. und T. G. KIM: *Framework for Simulation of Hybrid Systems: Interoperation of Discrete Event and Continuous Simulators Using HLA/RTI*. In: *Proceedings of the 2011 IEEE Workshop on Principles of Advanced and Distributed Simulation*, PADS '11, S. 1–8, Washington, DC, USA, 2011. IEEE Computer Society.

[243] SUTTER, H.: *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*. Dr. Dobbs Journal, 30(3):202–210, 2005.

[244] SYNOPSIS, INC.: *Describing Synthesizable RTL in SystemC*, 2002.

[245] SZTIPANOVITS, J. und G. KARSAI: *Model-Integrated Computing*. IEEE Computer, 30:110–112, April 1997.

[246] TABUADA, P.: *Cyber-Physical Systems: Position Paper*. In: *NSF Workshop on Cyber Physical Systems*, 2006.

[247] TANENBAUM, A.: *Structured Computer Organization*. Alternative Etext Formats. Pearson Prentice Hall, 2006.

[248] TEICH, J. und C. HAUBELT (Hrsg.): *Digitale Hardware/Software-Systeme : Synthese und Optimierung*. Springer Berlin Heidelberg, 2., erweiterte Auflage Aufl., 2007.

[249] TOL, M. W. VAN, R. BAKKER, M. VERSTRAATEN, C. GRELCK und C. R. JESSHOPE: *Efficient Memory Copy Operations on the 48-core Intel SCC Processor.*. In: GÖHRINGER, D., M. HÜBNER und J. BECKER (Hrsg.): *MARC Symposium*, S. 13–18. KIT Scientific Publishing, Karlsruhe, 2011.

[250] TOLK, A.: *Interoperability, Composability, and Their Implications for Distributed Simulation: Towards Mathematical Foundations of Simulation Interoperability*. In: *Distributed Simulation and Real Time Applications (DS-RT), 2013 IEEE/ACM 17th International Symposium on*, S. 3–9, Oct 2013.

[251] TOLK, D. A. und J. A. MUGUIRA: *The Levels of Conceptual Interoperability Model*. In: *in 2003 Fall Simulation Interoperability Workshop*, 2003.

[252] TOPÇU, O., M. ADAK und H. OĞUZTÜZÜN: *A Metamodel for Federation Architectures*. ACM Trans. Model. Comput. Simul., 18(3):10:1–10:29, Juli 2008.

[253] TREIBER, M., A. HENNECKE und D. HELBING: *Congested Traffic States in Empirical Observations and Microscopic Simulations*. Rev. E 62, Issue, 62:2000, 2000.

[254] TREIBER, M. und A. KESTING: *Verkehrsdynamik und -Simulation: Daten,*

Modelle Und Anwendungen Der Verkehrsflussdynamik. Springer-Lehrbuch. Springer, 2010.

[255] TRIPAKIS, S., C. STERGIOU, C. SHAVER und E. A. LEE: *A modular formal semantics for Ptolemy*. Mathematical Structures in Computer Science, 23(04):834–881, August 2013.

[256] TURING, A. M.: *On Computable Numbers, with an Application to the Entscheidungsproblem*. Proceedings of the London Mathematical Society, s2-42(1):230–265, Jan. 1937.

[257] U.S. DEPARTMENT OF DEFENSE: *High Level Architecture (HLA) – Version 1.3*, 1998.

[258] VARGA, A. und R. HORNIG: *An Overview of the OMNeT++ Simulation Environment*. In: *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, Simutools '08, S. 60:1–60:10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[259] VIAUD, E., D. POTOP-BUTUCARU und A. GREINER: *An Efficient TLM/T Modeling and Simulation Environment Based on Conservative Parallel Discrete Event Principles*. In: *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, Bd. 1, S. 1–6, March 2006.

[260] VINCO, S., D. CHATTERJEE, V. BERTACCO und F. FUMMI: *SAGA: SystemC Acceleration on GPU Architectures*. In: *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, S. 115–120, New York, NY, USA, 2012. ACM.

[261] WACHTER, E., C. LUCAS, E. CARARA und F. MORAES: *An open-source framework for heterogeneous MPSoC generation*. In: *Programmable Logic (SPL), 2012 VIII Southern Conference on*, S. 1–6, March 2012.

[262] WANG, W., A. TOLK und W. WANG: *The Levels of Conceptual Interoperability Model: Applying Systems Engineering Principles to M&S*. In: *Proceedings of the 2009 Spring Simulation Multiconference*, SpringSim '09, S. 168:1–168:9, San Diego, CA, USA, 2009. Society for Computer Simulation International.

[263] WEHRLE, K., J. GROSS und M. GÜNES: *Modeling and Tools for Network Simulation*. Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg, 2010.

[264] WEISER, M.: *The Computer for the 21st Century*. SIGMOBILE Mob. Comput. Commun. Rev., 3(3):3–11, Juli 1999.

[265] WETTER, M.: *Co-simulation of building energy and control systems with the Building Controls Virtual Test Bed*. Journal of Building Performance Simulation, 4(3):185–203, 2011.

[266] WIJNGAART, R. F. VAN DER, T. G. MATTSON und W. HAAS: *Light-weight communications on Intel's single-chip cloud computer processor*. SIGOPS Oper. Syst. Rev., 45(1):73–83, Feb. 2011.

[267] YEW, P.-C., N.-F. TZENG und D. H. LAWRIE: *Distributing Hot-Spot Addressing in Large-Scale Multiprocessors*. Computers, IEEE Transactions on, C-36(4):388–395, 1987.

[268] YI, Y., D. KIM und S. HA: *Fast and Accurate Cosimulation of MPSoC Using Trace-Driven Virtual Synchronization*. Trans. Comp.-Aided Des. Integ. Cir. Sys., 26(12):2186–2200, Dez. 2007.

[269] ZEIGLER, B. P., T. G. KIM und H. PRAEHOFER: *Theory of Modeling and Simulation*. Academic Press, Inc., Orlando, FL, USA, 2nd Aufl., 2000.

[270] ZIMMERMANN, H.: *OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection*. Communications, IEEE Transactions on, 28(4):425–432, 1980.

[271] ZIYU, H., Q. LEI, L. HONGLIANG, X. XIANGHUI und Z. KUN: *A Parallel SystemC Environment: ArchSC*. In: *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, S. 617–623, Dec 2009.

Betreute studentische Arbeiten

[Buc12] BUCIUMAN, MARIUS-FLORIN: *Entwurf eines Network-on-Chip Modells zur schnellen parallelen Simulation auf dem Single-chip Cloud Computer*. Bachelorarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2012.

[Bwa09] BWADKJI, MOHAMED: *Modellierung einer Car-to-X Communication Unit mittels SystemC zur Entwurfsraumexploration*. Diplomarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2009.

[Erd12] ERDOGAN, GÖKHAN: *Erweiterung eines Code-Generators hinsichtlich der Möglichkeit zur Erzeugung flexibel partitionierbarer SystemC Modelle*. Studienarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2012.

[Ker12] KERN, MATTHIAS: *Entwicklung eines Many-Core Prozessormodells auf Basis eines Network-on-Chip mit QoS Unterstützung*. Bachelorarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2012.

[LZ10] LASZLO ZSOLT, DÉNES: *Entwicklung eines Simulationsframeworks für System-on-Chip basierte Netzwerke*. Bachelorarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2010.

[Ras12] RASTETTER, ROUVEN: *Portierung von PtidyOS auf ein Multiprozessorsystem zur echtzeitfähigen Ausführung verteilter Tasks*. Bachelorarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2012.

[Red11] REDER, SIMON: *Entwicklung eines Konzepts zur synchronen parallelen SystemC Simulation auf dem Single-chip Cloud Computer*. Bachelorarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2011.

[Red14] REDER, SIMON: *Adaptiver Algorithmus und Tool Chain zur Beschleunigung von SystemC auf Many-Core Architekturen*. Masterarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2014.

[Sch13] SCHULTSCHIK, SVEN: *Entwicklung einer Multi-Domänen Simulation zur interdisziplinären Betrachtung vernetzter eingebetteter Systeme*. Studienarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2013.

Veröffentlichungen

Konferenz- & Zeitschriftenbeiträge

[ADS+10] ALBERS, A., T. DÜSER, O. SANDER, C. ROTH und J. HENNING: *Development Platform for Vehicles, Control Units and Communication Systems*. ATZelektronik worldwide, 5(5):50–54, 2010.

[BNR+13] BRITO, A., A. NEGREIROS, C. ROTH, O. SANDER und J. BECKER: *Development and Evaluation of Distributed Simulation of Embedded Systems using Ptolemy and HLA*. In: *17th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, Delft, 2013.

[BSR12] BECKER, J., O. SANDER und C. ROTH: *Processor Solutions for Smart Mobility*. In: *48. Workshop der Multi Projekt Chip - Gruppe der Hochschulen in Baden Württemberg*, Seiten 1–10, 2012.

[DSK+13] DRESCHMANN, M., O. SANDER, A. KLIMM, C. ROTH und J. BECKER: *Addiguration: Exploring configuration behavior of Spartan-3 devices*. In: *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2013 8th International Workshop on*, Seiten 1–6, 2013.

[KBR+11a] KUEHNLE, M., A. BRITO, C. ROTH, K. DAGAS und J. BECKER: *The Study of a Dynamic Reconfiguration Manager for Systems-on-Chip*. In: *VLSI (ISVLSI), 2011 IEEE Computer Society Annual Symposium on*, Seiten 13–18, 2011.

[KBR+11b] KUEHNLE, M., A. BRITO, C. ROTH, M. KRUESSELIN und J. BECKER: *An approach for power and performance evaluation of reconfigurable SoC at mixed abstraction levels*. In: *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2011 6th International Workshop on*, Seiten 1–8, 2011.

[RAS+11] ROTH, C., G.M. ALMEIDA, O. SANDER, L. OST, N. HEBERT, G. SASSATELLI, P. BENOIT, L. TORRES und J. BECKER: *Modular Framework for Multi-level Multi-device MPSoC Simulation*. In: *Parallel and Dis-*

tributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on, Seiten 136–142, 2011.

[RBB+14] ROTH, C., H. BUCHER, A. BRITO, O. SANDER und J. BECKER: *A Simulation Tool Chain for Investigating Future V2X-based Automotive E/E Architectures*. In: *Proceedings of Embedded Real Time Software and Systems Conference*, 2014.

[RBR+13a] ROTH, C., H. BUCHER, S. REDER, F. BUCIUMAN, O. SANDER und J. BECKER: *A SystemC Modeling and Simulation Methodology for Fast and Accurate Parallel MPSoC Simulation*. In: *26th Symposium on Integrated Circuits and Systems Design*, Curitiba - Brazil, 2013.

[RBR+13b] ROTH, C., H. BUCHER, S. REDER, O. SANDER und J. BECKER: *Improving parallel MPSoC simulation performance by exploiting dynamic routing delay prediction*. In: *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2013 8th International Workshop on*, Seiten 1–8, 2013.

[RMR+12] ROTH, C., J. MEYER, M. RÜCKAUER, O. SANDER und J. BECKER: *Efficient Execution of Networked MPSoC Models by Exploiting Multiple Platform Levels*. Int. J. Reconfig. Comput., 2012:27–39, Januar 2012.

[RRB+14] ROTH, C., S. REDER, H. BUCHER, O. SANDER und J. BECKER: *Adaptive Algorithm and Tool Flow for Accelerating SystemC on Many-Core Architectures*. In: *2014 Euromicro Conference on Digital System Design (DSD)*, Verona, Italy, 2014.

[RRE+12] ROTH, C., S. REDER, G. ERDOGAN, O. SANDER, G.M. ALMEIDA, H. BUCHER und J. BECKER: *Asynchronous parallel MPSoC simulation on the Single-Chip Cloud Computer*. In: *System on Chip (SoC), 2012 International Symposium on*, 2012.

[RRS+12] ROTH, C., S. REDER, O. SANDER, M. HÜBNER und J. BECKER: *A Framework for Exploration of Parallel SystemC Simulation on the Single-chip Cloud Computer*. In: *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques*, SIMUTOOLS '12, Seiten 202–207. ICST, 2012.

[RSB11] ROTH, CHRISTOPH, OLIVER SANDER und JÜRGEN BECKER: *Flexible and efficient co-simulation of networked embedded devices*. In: *Proceedings of the 24th symposium on Integrated circuits and systems design*, SBCCI '11, Seiten 61–66, New York, NY, USA, 2011. ACM.

[RSG+10] ROTH, C., O. SANDER, B. GLAS, J. BECKER, T. DÜSER, A. SEIFERMANN, A. ALBERS, K. D. MÜLLER-GLASER und J. HENNING: *Car-to-X-in-the-Loop - Development Environment for Vehicles, Control Units and Communication Systems in the Context of future Mobility*

Concepts. In: *Fahrerassistenz und Integrierte Sicherheit. 26. VDI/VW-Gemeinschaftstagung*, VDI-Berichte, Seiten 67–80. VDI Verlag GmbH, 2010.

[RSHB10a] ROTH, C., O. SANDER, M. HUEBNER und J. BECKER: *Car-to-X Simulation Environment for Comprehensive Design Space Exploration Verification and Test*. SAE Int. J. Passeng. Cars - Electron. Electr. Syst., 2010:17–26, 2010.

[RSHB10b] ROTH, C., O. SANDER, M. HUEBNER und J. BECKER: *Car-to-X Simulation Environment for Comprehensive Design Space Exploration Verification and Test*. In: *SAE 2010 World Congress*, 2010.

[RSKB11] ROTH, C., O SANDER, M. KÜHNLE und J. BECKER: *HLA-based simulation environment for distributed SystemC simulation*. In: *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, SIMUTools '11, Seiten 108–114. ICST, 2011.

[SGR+09a] SANDER, O., B. GLAS, C. ROTH, J. BECKER und K. D. MUELLER-GLASER: *Real time information processing for car to car communication applications*. In: *12th EAEC European Automotive Congress*, 2009.

[SGR+09b] SANDER, O., B. GLAS, C. ROTH, J. BECKER und K.D. MUELLER-GLASER: *Design of a Vehicle-to-Vehicle communication system on reconfigurable hardware*. In: *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, Seiten 14–21, 2009.

[SGR+09c] SANDER, O., B. GLAS, C. ROTH, J. BECKER und K.D. MUELLER-GLASER: *Priority-based packet communication on a bus-shaped structure for FPGA-systems*. In: *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, Seiten 178–183, 2009.

[SGR+09d] SANDER, O., B. GLAS, C. ROTH, J. BECKER und K.D. MUELLER-GLASER: *Testing of an FPGA Based C2X-Communication Prototype with a Model Based Traffic Generation*. In: *Rapid System Prototyping, 2009. RSP '09. IEEE/IFIP International Symposium on*, Seiten 68–71, 2009.

[SRGB13] SANDER, O., C. ROTH, B. GLAS und J. BECKER: *Towards Design and Integration of a Vehicle-to-X Based Adaptive Cruise Control*. In: *Proceedings of the FISITA 2012 World Automotive Congress*, Band 200 der Reihe *Lecture Notes in Electrical Engineering*, Seiten 87–99. Springer Berlin Heidelberg, 2013.

[SRSB09] SANDER, OLIVER, CHRISTOPH ROTH, VITALI STUCKERT und JÜRGEN BECKER: *System concept for an FPGA based real-time capable automotive ECU simulation system*. In: *Proceedings of the 22nd Annual Symposium on Integrated Circuits and System Design: Chip on the Dunes*, SBCCI '09, Seiten 34:1–34:6, New York, NY, USA, 2009. ACM.

Christoph Roth

Parallele und kooperative Simulation für eingebettete Multiprozessorsysteme

Die Entwicklung von eingebetteten Systemen wird durch die stetig steigende Anzahl und Integrationsdichte neuer Funktionen in Kombination mit einem erhöhten Interaktionsgrad zunehmend zur Herausforderung. Vor diesem Hintergrund stellen Methoden zur Simulationsbeschleunigung sowie zur Verbesserung der Interoperabilität zwei zentrale Teilaspekte von zukünftigen simulationsbasierten Entwicklungsprozessen dar.

Hinsichtlich des ersten Teilaspekts werden in dieser Arbeit unterschiedliche Strategien für die SystemC-basierte parallele Simulation von eingebetteten Multiprozessorsystemen auf zukünftigen Manycore Architekturen entwickelt. Dabei stehen zyklenakkurate und zyklenapproximative Modelle im Fokus der Betrachtung. Die verschiedenen Strategien werden implementiert, experimentell untersucht und bewertet.

Bezüglich des zweiten Teilaspekts wird eine neuartige modellbasierte Methode zur Verbesserung der Interoperabilität zwischen heterogenen Simulationswerkzeugen vorgestellt. Realisierbarkeit und Anwendbarkeit der Methode werden anhand einer Werkzeugkette und verschiedener Fallstudien demonstriert.