CARL VON OSSIETZKY universität OLDENBURG
FAKULTÄT II
INFORMATIK, WIRTSCHAFTS- UND RECHTSWISSENSCHAFTEN
Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

# Effiziente Integration von Hardwarebeschreibungen in Simulink/TDF-Simulationen

Dissertation zur Erlangung des Grades eines
Doktors der Ingenieurwissenschaften

von

**Dipl. Ing. (FH) Ralph Görgen**

Gutachter:

**Prof. Dr.-Ing. Wolfgang Nebel**
**Prof. Dr. Martin Fränzle**

Tag der Disputation: 28. Januar 2014

# Kurzfassung

Simulation ist ein zentrales Werkzeug bei der Entwicklung und Verifikation eingebetteter Systeme. Sie steht jedoch durch die stetig steigende Komplexität der Systeme, zunehmende Miniaturisierung und die immer dichtere Integration der Komponenten vor neuen Herausforderungen. Zur Realisierung komplexer, heterogener Systeme als System-in-Package, das Software, digitale Hardware, analoge elektronische Anteile, oder auch optische Komponenten auf einem einzelnen Chip integriert, ist es nicht ausreichend die jeweiligen Komponenten als Einzelteile zu entwickeln. Um die starke Interaktion der Komponenten untereinander berücksichtigen zu können, ist es vielmehr erforderlich, die Summe der Einzelteile als Gesamtsystem zu betrachten. Nur so können die hohen Anforderungen bezüglich Leistungsfähigkeit, Energiebedarf und Zuverlässigkeit erfüllt werden.

Problematisch ist dabei, dass zur Implementierung der einzelnen Komponenten in der Regel domänenspezifische Werkzeuge eingesetzt werden, wie zum Beispiel Hardwarebeschreibungssprachen für digitale Hardware oder SPICE für analoge Elektronik. Zur abstrakten Modellierung des Gesamtsystems werden hingegen Umgebungen wie MATLAB/Simulink verwendet, die speziell zur Modellierung von heterogenen Multidomänensystemen geeignet sind. Für eine Simulation der Komponenten im Zusammenhang des Gesamtsystems sind deshalb Methoden notwendig, die eine Integration der Komponentenmodelle in das Systemmodell ermöglichen.

In dieser Arbeit wird eine Methode zur effizienten Integration von digitalen Hardwarekomponenten in Systemmodelle vorgestellt. Das Modell der Hardwarekomponente liegt in einer Hardwarebeschreibungssprache vor, das Gesamtsystem ist in Simulink modelliert. Während bei existierenden Methoden zur Ko-Simulation großer Mehraufwand zur Synchronisation der Simulatoren und zum Datenaustausch erforderlich ist, basiert der hier entwickelte Ansatz auf einer Transformation des Hardwaremodells, das dann direkt im Kontext des Simulink-Modells ausgeführt werden kann. Da kein eigenständiger Simulator für das Hardwaremodell benötigt wird, ist der Mehraufwand für Synchronisation und Datenaustausch stark reduziert und die Simulation des Gesamtsystems wird deutlich schneller. Die wesentliche Herausforderung ergibt sich dabei aus den unterschiedlichen Simulationssemantiken. Die Simulation in Simulink ist zeitgetrieben und datenflussorien-

tiert, wohingegen für Hardwarebeschreibungssprachen ereignisgetriebene Simulation eingesetzt wird. Darüber hinaus wird eine weitere Optimierung der Simulationsgeschwindigkeit vorgestellt, die durch die Transformation des Hardwaremodells möglich wird und auf dem Überspringen von Phasen ohne Zustandsänderungen in der Simulation basiert.

# Abstract

Simulation is an essential tool for development and verification of embedded systems. But key challenges arise from the growing complexity of such systems, ongoing miniaturization, and the increasing integration density. Complex heterogeneous systems are realised as System-in-Package containing software, digital hardware, analogue electronic parts, as well as optical components. In their development it is no longer sufficient to design the components as individual parts. In regards to the strong interaction between components, it is necessary to consider the sum of all parts as an entire system. This is the only way to meet the high requirements regarding performance, energy consumption, and reliability.

Problems arise in this context concerning the use of many domain-specific tools to implement the particular components, for instance hardware description languages for digital hardware or SPICE for analogue electronics. However, the abstract modelling of the overall system is performed in specific modelling environments like MATLAB/Simulink, which are suitable for heterogeneous multi-domain systems. To run simulations of the components in the context of the entire system, methods that allow the integration of component models into the system model are required.

This thesis describes a method for the efficient integration of digital hardware components into system models. The model of the hardware component is given in a hardware description language. The entire system is modelled in Simulink. Existing co-simulation methods require a lot of additional computational efforts to synchronise the simulator tools and exchange data between them. The approach presented here is based on a transformation of the hardware model, which allows its execution directly in the context of the Simulink model. As no separate simulator is needed to simulate the hardware component, the overhead for synchronisation and data exchange is reduced significantly. Hence, the overall system simulation becomes faster. The key challenge arises from the different simulation semantics. In Simulink, simulation is time-driven and data-flow oriented while event-driven simulation is used for hardware description languages. In addition, a further optimisation of the simulation performance is presented. The transformed hardware model enables this improvement and is based on skipping phases without state changes in the simulation.

# Inhaltsverzeichnis

# 1 Einleitung

Die technologische Entwicklung der letzten Jahre im Bereich der Mikroelektronik eröffnet ständig neue Möglichkeiten und Anwendungen. Immer kleinere Strukturgrößen ermöglichen eine dichtere Integration von Komponenten und die Implementierung immer komplexerer Systeme auf einem einzelnen Mikrochip. Aufgrund dieser hochdichten Integration konnte die Leistungsfähigkeit mikroelektronischer Systeme immer weiter gesteigert werden, obwohl die Taktfrequenzen zuletzt nicht mehr so stark angestiegen sind.

Neben der zunehmenden Integration digitaler elektronischer Komponenten, sind in den vergangenen Jahren, insbesondere im Bereich der eingebetteten Systeme, Begriffe wie *More-than-Moore* oder *Cyber-Physical Systems* immer populärer geworden. Sie stehen für die Bestrebung, Komponenten aus unterschiedlichen Domänen in einem System-in-Package (SiP) zusammenzubringen. Systeme, bestehend aus Software, digitaler Hardware, analogen elektronischen und elektrischen Anteilen, sowie mechanischen oder optischen Komponenten, sollen integriert in einem einzelnen Chip realisiert werden. Die räumliche Nähe der Komponenten zueinander ermöglicht eine höhere Leistung und oft einen geringeren Energiebedarf. Die Entwicklung solcher Systeme bringt aber auch zusätzliche Herausforderungen mit sich. Es ist nicht länger möglich, die Komponenten als Einzelteile zu entwickeln. Vielmehr muss im Entwurfsprozess das Gesamtsystem betrachtet werden, um die starke Interaktion der Komponenten untereinander berücksichtigen zu können. Dazu kommen die hohen Anforderungen an moderne eingebettete Systeme bezüglich Leistungsfähigkeit und Energiebedarf, sowie Zuverlässigkeit und Robustheit in einer oftmals rauen Umgebung. Um diese Anforderungen erfüllen zu können und bei steigendem Kostendruck eine hohe Qualität heterogener eingebetteter Systeme sicherstellen zu können, werden neue Methoden zum Entwurf und zur Verifikation solcher Systeme benötigt. Dabei spielt die Simulation der Systeme eine wichtige Rolle.

Dieser Bedarf wird auch in aktuellen Ausschreibungen zur Forschungsförderung auf europäischer Ebene thematisiert. Im CATRENE White Book [CAT11] zu den Herausforderungen der nächsten Jahre ist zum Beispiel zum Thema intelligente elektrische Fahrzeuge zu lesen, dass innovative Systeme durch “neue System-in-Package Technologien und Design- und Verifikationsmethoden ermög-

licht werden"[1]. Im Abschnitt zum Umgang mit der Vielfalt in heterogenen SiPs wird "eine effiziente A/MS-Simulation[2] für große Systeme"[3] als eine der momentanen Hauptherausforderungen bezeichnet. In einem speziellen Abschnitt zum Thema System-in-Package wird der Bedarf an ganzheitlichen Ansätzen, wie zum Beispiel "Methoden und Werkzeuge für Ko-Design auf Systemebene"[4], ausgedrückt. Die HiPEAC-Roadmap [DBSYB11] beschreibt ebenfalls im Bereich heterogener Systeme "neue Ansätze zur Reduktion von Simulations- und Validierungszeit"[5] als eine der Herausforderungen.

## 1.1 Entwurf heterogener eingebetteter Systeme

Die Entwicklung komplexer, heterogener eingebetteter Systeme ist ohne strukturierte Entwurfsprozesse nicht möglich. Ein Vorgehensmodell, das dazu häufig als Grundlage dient, ist der Top-Down-Entwurf nach dem V-Modell (Abbildung 1.1). Beginnend mit einer abstrakten Spezifikation auf Systemebene, wird der Entwurf Schritt für Schritt verfeinert, bis schließlich eine Implementierung des Systems erreicht ist. Parallel dazu werden auf allen Ebenen Test- und Validierungsschritte durchgeführt.

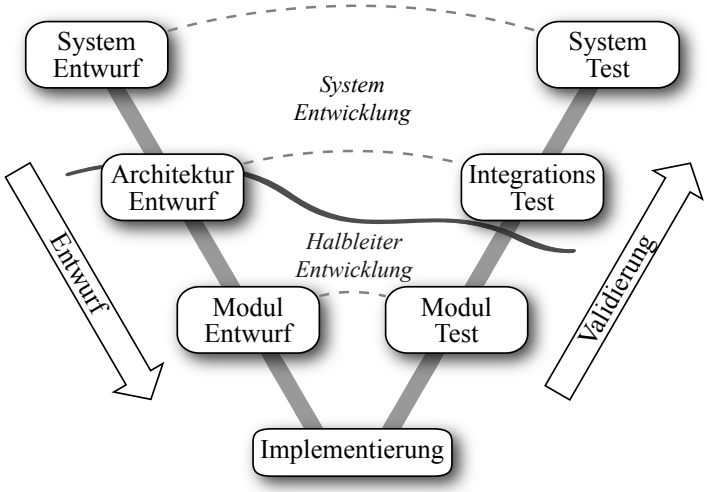



Abbildung 1.1: V-Modell

[1] [CAT11], Seite 68
[2] A/MS: Analog/Mixed Signal; Systeme, die analoge und digitale Anteile beinhalten
[3] [CAT11], Seite 120
[4] [CAT11], Seite 132
[5] [DBSYB11], Seite 10

Im industriellen Alltag ist ein striktes Vorgehen nach diesem Modell jedoch oft nicht realisierbar. Entwicklungsprozesse verlaufen nicht ausschließlich Top-Down. Ein Beispiel ist die Validierung verfeinerter Komponenten im Systemkontext. Zwar liegen aus vorhergehenden Entwurfsschritten Modelle dieser Komponenten auf Systemebene vor, die letztendliche Implementierung kann jedoch davon abweichen. Gründe dafür können Fehler und nachträgliche Änderungen oder Optimierungen sein, die aus Kostengründen nicht konsistent über alle Entwurfsebenen nachgeführt wurden. Ein weiteres Beispiel ist die Wiederverwendung bereits existierender Komponenten. Wiederverwendung und Zukauf sogenannter IP-Komponenten[6] spielt heutzutage eine sehr wichtige Rolle. Selbst die Realisierung neuer Anwendungen ausschließlich aus existierenden Komponenten, die lediglich in einer neuen Kombination zusammengestellt werden, ist nicht selten. Für diese Komponenten stehen oftmals keine Modelle für alle Abstraktionsebenen zur Verfügung. Dennoch muss eine Verifikation des Gesamtsystems, beziehungsweise der Komponenten im Kontext des Gesamtsystems, durchgeführt werden.

Ein weiteres Problem, das die nahtlose Entwicklung behindert, ist die Kooperation zwischen verschiedenen Abteilungen oder Unternehmen. In Abbildung 1.1 ist als Beispiel die Schnittstelle zwischen den Abteilungen Systementwicklung und Halbleiterentwicklung angedeutet. Neben unterschiedlichem Know-How in den einzelnen Abteilungen unterscheiden sich auch die Werkzeuge und Arbeitsumgebungen, die zum Einsatz kommen. Auf Systemebene werden in der Regel Modellierungs- und Simulationsumgebungen eingesetzt, die für heterogene Multidomänensysteme geeignet sind. Ein weit verbreitetes Beispiel ist MATLAB/Simulink. Zur Implementierung der einzelnen Komponenten auf niedrigeren Abstraktionsebenen werden jedoch meist domänenspezifische Werkzeuge eingesetzt. Beispiele dafür sind Hardwarebeschreibungssprachen zur Modellierung digitaler Schaltungen, virtuelle Plattformen und Instruction-Set-Simulatoren zur Entwicklung von Software sowie spezielle Simulatoren wie SPICE zur Simulation analoger Elektronik. Ist die Entwicklung auf Komponentenebene abgeschlossen, so müssen die Spezialabteilungen Modelle ihrer Komponenten zur Verfügung stellen, die seitens der Systementwicklungsabteilung in das Modell des Gesamtsystems integriert werden können. In dieser Arbeit liegt der Fokus speziell auf der Integration digitaler Hardwarekomponenten, die in einer Hardwarebeschreibungssprache implementiert sind, in Gesamtsysteme, die als Simulink-Modell vorliegen. Im Folgenden wird ein solcher Ablauf am Beispiel eines Entwicklungsprozesses für eine MEMS-Komponente[7] näher beschrieben.

[6] IP: Intellectual Property
[7] MEMS: Micro-Electro-Mechanical System

### 1.1.1 Entwurfsprozess einer MEMS-Komponente

Abbildung 1.2 zeigt das Modell eines elektro-mechanischen Sensors. Seine Grundstruktur ist typisch für MEMS-Bausteine. Das eigentliche Sensorelement transformiert eine Eigenschaft der Umgebung, zum Beispiel den Luftdruck, in ein Widerstands- oder Kapazitätssignal. Das analoge Frontend wandelt dieses Signal in ein analoges Strom- oder Spannungssignal um, welches dann von einem AD-Wandler in digitale Werte konvertiert wird. Die digitalen Werte werden schließlich von einer digitalen Kontrolleinheit entgegengenommen. Darin werden dann Filteralgorithmen oder ähnliche Schritte zur Aufbereitung der Daten durchgeführt. Schließlich wird das Ergebnis über eine Standardschnittstelle nach außen zur Verfügung gestellt.

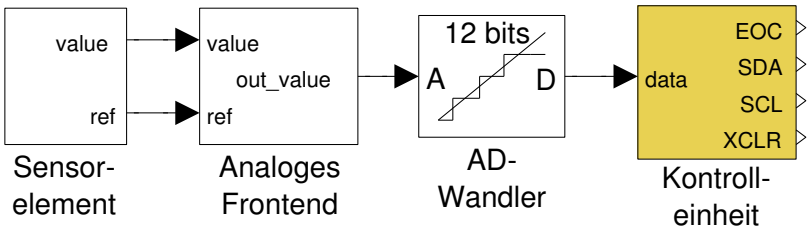



Abbildung 1.2: Architektur eines elektro-mechanischen Sensors

Der Entwicklungsprozess eines solchen Systems beginnt mit einem Modell der Gesamtanwendung auf Systemebene, das in Simulink erstellt wird. Anschließend wird eine Grenzlinie zwischen den Verarbeitungsschritten eingeführt, die durch das analoge Frontend realisiert werden, und denen, die von der digitalen Kontrolleinheit übernommen werden sollen. An dieser Grenze wird dann ein AD-Wandler eingebaut, und die abstrakte Beschreibung der digitalen Verarbeitungsschritte wird zu einem Datenflussmodell in Simulink verfeinert. In der finalen Implementierung soll die Digitaleinheit durch einen digitalen Signalprozessor (DSP) realisiert werden. Der nächste Schritt wäre nun die Auswahl eines passenden DSP, das Finden von dessen optimaler Konfiguration für diese Anwendung und schließlich Implementierung und Test des DSP-Programms. Die Halbleiterentwicklungsabteilung kann auf eine ganze Reihe von existierenden, konfigurierbaren DSPs aus vorangegangenen Projekten zurückgreifen. Diese liegen alle in Form von synthetisierbaren Hardwarebeschreibungen vor, und ihre Programmierung geschieht über einen speziellen Satz an Assembler-Instruktionen. Um auf Systemebene die Konfiguration und Programmierung des DSP vornehmen zu können, benötigt der Systementwickler nun ein Modell des DSP, das in das Gesamtmodell in Simulink integriert werden kann. Damit dabei ein möglichst gutes Ergebnis erreicht wird, muss dieses Integra-

tionsmodell das Verhalten des Originalmodells sowohl bezüglich der funktionalen als auch bezüglich der zeitlichen Eigenschaften exakt nachbilden.

Dies ist nur ein Beispiel von zahlreichen, in dem eine Methode zur Integration einer Hardwarebeschreibung in ein Systemmodell notwendig ist. In der industriellen Praxis müssen Entwickler von Hardwarekomponenten häufig Modelle zur Verfügung stellen, die eine Simulation ihrer Komponenten im Systemkontext ermöglichen.

## 1.2 Integration von Hardwarebeschreibungen in Simulink-Simulationen

Die Problematik an dieser Stelle erwächst aus den unterschiedlichen Werkzeugumgebungen in den einzelnen Bereichen. Die Synthese digitaler Schaltungen direkt aus einem Simulink-Modell ist nicht immer möglich oder zielführend. In Branchen wie der Automobilindustrie, in denen Mikrochips in sehr große Stückzahlen hergestellt werden, ist beispielsweise die Größe der Fläche der einzelnen Chips einer der dominierenden Kostenfaktoren. Der erhöhte Flächenbedarf, der bei einer automatischen Synthese im Vergleich zu einer hochoptimierten HDL-Beschreibung auftritt, ist in solchen Fällen mit größeren Kosten verbunden als die manuelle Implementierung und Optimierung in einer Hardwarebeschreibungssprache. Genauso unbefriedigend ist die Modellierung und Analyse komplexer, heterogener Systeme, wenn als einziges Werkzeug eine Hardwarebeschreibungssprache wie VHDL zur Verfügung steht. Um dennoch einen Austausch von Modellen zwischen den Bereichen vornehmen zu können, wird daher eine Methode benötigt, die einen Weg von einer Hardwarebeschreibung zu einem Simulink-Teilmodell ermöglicht. Wesentliche Anforderungen, die eine solche Methode erfüllen muss, sind Effizienz bei der Simulation und der Verzicht auf manuelle Schritte. Effizienz ist wichtig, da bei industriellen Systemen, insbesondere in sicherheitskritischen Anwendungen, viele und umfangreiche Testfälle simuliert werden müssen, um die notwendige Testabdeckung zu erreichen. Bei komplexen Systemen sind diese Simulationen sehr zeitaufwändig und jede Steigerung der Simulationsgeschwindigkeit hilft dabei den Entwicklungsprozess zu beschleunigen. Manuelle Schritte sind ebenfalls oft sehr zeitaufwändig und darüber hinaus eine mögliche Fehlerquelle. Die Methode soll deshalb automatisiert und ohne manuelle Transformationsschritte anwendbar sein. Ein weiterer wichtiger Punkt ist die exakte Wiedergabe des Verhaltens der Hardwarekomponente. Besonders bei Implementierung und Test eines DSP-Programms oder ähnlichen Entwurfsschritten ist eine genaue Simulation sowohl des funktionalen als auch des zeitlichen Verhaltens äußerst wichtig.

Der unterstützte Funktionsumfang der Hardwarebeschreibungssprache kann auf die synthetisierbare Untermenge beschränkt werden, da die Hardwarekomponenten als synthetisierbare Modelle vorliegen. Weitere Einschränkungen bezüglich der Hardwarebeschreibungssprache sollen nicht notwendig sein. Eine Methode, die den Funktionsumfang von Simulink einschränkt, ist ebenfalls unerwünscht. Solche Limitierungen betreffen das Modell des Gesamtsystems und müssten deshalb im gesamten Entwurfsprozess auf Systemebene berücksichtigt werden.

Aus praktischen Gesichtspunkten sollen außerdem die Kontrolle und der Ablauf der Simulation in Simulink stattfinden. Simulink bietet einige besondere und wichtige Funktionen, wie zum Beispiel umfangreiche Möglichkeiten zur Visualisierung von Simulationsergebnissen oder die Modifikation von Parametern in einer laufenden Simulation. Außerdem ist der Systementwickler mit dieser Umgebung vertraut und eine Umstellung der Arbeitsumgebung ist immer mit erheblichem Aufwand verbunden.

Existierende Ansätze bieten keine zufriedenstellende Lösung für dieses Problem. Bei einigen sind manuelle Schritte zur Transformation der Modelle notwendig, die Methoden sind somit nicht automatisiert anwendbar. Andere sind nicht effizient, da sie für allgemeinere Anwendungsfälle konzipiert sind und erheblicher zusätzlicher Rechenaufwand zur Synchronisation und zum Datenaustausch zwischen den Teilmodellen notwendig ist.

Ziel dieser Arbeit ist es, eine automatisch anwendbare Methode zur Integration von Hardwarebeschreibungen in Simulink-Simulationen zu entwickeln. Dabei sollen die speziellen Eigenschaften des Anwendungsszenarios ausgenutzt werden, um eine im Vergleich mit existierenden Lösungen effizientere Simulation zu ermöglichen.

Zusammengefasst ergeben sich aus dem Anwendungsfall die folgenden Anforderungen:

1. Hohe Effizienz und Geschwindigkeit bei der Simulation des Gesamtsystemmodells.
2. Die Methode soll automatisch ohne manuelle Schritte anwendbar sein.
3. Das Verhalten, sowohl funktional als auch zeitlich, soll in der Systemsimulation exakt wiedergegeben werden.
4. Unterstützung für Hardwaremodelle, die als synthetisierbare Hardwarebeschreibungen gegeben sind.
5. Keine Einschränkung bei der Verwendung von Simulink-Funktionalität im Simulink-Teil des Modells.
6. Das Hardwaremodell soll in Simulink und unter der Kontrolle von Simulink

simulierbar sein.

Um dieses Ziel erreichen zu können, müssen zunächst die Simulationssemantiken von Hardwarebeschreibungssprachen und Simulink formal erfasst werden. Basierend auf dieser Formalisierung müssen dann die Eigenschaften des Anwendungsfalls untersucht werden. Daraus ist dann eine Methode zu entwickeln, die eine effiziente gemeinsame Simulation der Modellteile erlaubt, bei der das Verhalten des Gesamtmodells exakt wiedergegeben wird. Hierbei ist zusätzlich der Umgang mit Takt-synchronen Hardwarekomponenten zu berücksichtigen, bei denen unabhängig von der Simulink-Simulation eine vorgegebene Taktrate realisiert werden muss. Schließlich muss die formal entwickelte Methode auf den praktischen Anwendungsfall, die Simulation einer Hardwarebeschreibung in einer Simulink-Umgebung, abgebildet werden, und es muss gezeigt werden, dass die gestellten Anforderungen erfüllt sind.

Die wesentlichen Beiträge der vorliegenden Arbeit sind:

- Abbildung der abstrakten Simulationssemantik von Simulink und Hardwarebeschreibungssprachen auf die Formalismen zur Systemspezifikation *Discrete Time System Specification* (DTSS) und *Discrete Event System Specification* (DEVS).
- Formulierung der speziellen Eigenschaften und Anforderungen des Anwendungsfalls als Einschränkungen eines DEVS-Systems.
- Transformation des eingeschränkten DEVS-Systems zu einem DTSS-System.
- Erweiterung der Transformation auf Takt-synchrone Hardwaremodelle.
- Beschreibung eines Konzepts zur automatisierten Anwendung der Transformationsmethode auf Hardwarebeschreibungen zur Simulation in einer Simulink-Umgebung.
- Beschreibung einer weiteren Optimierung der Simulation, die auf bestimmten Verhaltensmustern in Hardwarekomponenten basiert und durch die zuvor entwickelte Transformation möglich wird.
- Evaluation und Diskussion der entwickelten Methoden bezüglich der gestellten Anforderungen.

Eine detaillierte Beschreibung der Problemstellung und des Beitrags dieser Arbeit folgt in Kapitel 4.

## 1.3 Aufbau der Arbeit

Die vorliegende Arbeit ist wie folgt gegliedert: Kapitel 2 führt grundlegende Begriffe der computergestützten Simulation ein und stellt die für diese Arbeit relevanten Modellierungs- und Simulationsumgebungen vor: Hardwarebeschreibungssprachen und Simulink. In Kapitel 3 wird der Stand der Technik betrachtet. Existierende Ansätze im Zusammenhang dieser Arbeit werden vorgestellt und hinsichtlich der genannten Anforderungen untersucht. Anschließend wird in Kapitel 4 die Problemstellung dieser Arbeit nochmals im Detail erarbeitet und das Grundkonzept der vorgeschlagenen Methode erläutert. In Kapitel 5 werden Modellierungs- und Simulationsumgebungen zunächst auf formale Modelle zur Systemspezifikation abgebildet auf deren Basis dann die Integration der Modelle theoretisch betrachtet wird. Danach werden die theoretischen Betrachtungen in Kapitel 6 auf Takt-synchrone Hardwarekomponenten erweitert, bei denen die Taktfrequenz nicht gleich der Samplefrequenz im Simulink-Modell ist. Im Anschluss daran werden in Kapitel 7 die theoretischen Ergebnisse auf die realen Simulationsumgebungen übertragen und die praktische Umsetzung der Methode beschrieben, bevor in Kapitel 8 eine Evaluation der vorgestellten Methode anhand von verschiedenen Beispielmodellen durchgeführt wird. In Kapitel 9 wird eine Erweiterung der bisherigen Methode vorgestellt und untersucht, die eine weitere Steigerung der Simulationsgeschwindigkeit erlaubt. Kapitel 10.1 fasst die Ergebnisse der Evaluation zusammen und bewertet die Gesamtmethode anhand der zuvor definierten Anforderungen. In Kapitel 10.2 wird schließlich ein Ausblick auf mögliche weiterführende Arbeiten gegeben bevor Zusammenfassung und Fazit in Kapitel 10.3 die Arbeit abschließen.

# 2 Grundlagen

Im ersten Abschnitt werden zunächst Definitionen der Grundbegriffe von Modellierung und Simulation nach [ZPK07] eingeführt, und verschiedene Simulationskonzepte werden vorgestellt. Danach folgt eine Einführung in die beiden in dieser Arbeit relevanten Modellierungs- und Simulationsumgebungen, Hardwarebeschreibungssprachen und MATLAB/Simulink.

## 2.1 Modellierung und Simulation

In [ZPK07] wird die Theorie von Modellierung und Simulation ausführlich behandelt und Definitionen der Grundbegriffe und Basiselemente eingeführt. In den folgenden Abschnitten werden zunächst die wesentlichen Teile dieser Grundlagen zusammengefasst, bevor verschiedene Simulatorkonzepte vorgestellt werden.

### Basiselemente von Modellierung und Simulation und ihre Beziehungen

Das Ziel computergestützter Simulation ist es, die Eigenschaften eines *Quellsystems* in Form eines Computerprogramms abzubilden. Das Quellsystem stellt eine natürliche oder virtuelle Vorlage dar. Die Simulation soll sein Verhalten bezüglich bestimmter Eigenschaften unter bestimmten Bedingungen wiedergeben.

In der Regel ist es nicht möglich, aber auch nicht notwendig, sämtliche Eigenschaften eines Quellsystems in beliebigem Detailgrad und unter allen möglichen äußeren Bedingungen zu simulieren. Dies würde sehr großen Aufwand bedeuten, einerseits für die Berechnung und Ausführung der Simulation im Computer, aber auch für die Erstellung der Modelle. Da Simulation verwendet wird, um eine bestimmte Fragestellung zu beantworten, ist es ausreichend, die für die Fragestellung relevanten Eigenschaften in einem geeigneten Detailgrad und unter den durch die Fragestellung induzierten Bedingungen korrekt wiederzugeben. Diese Eigenschaften und Bedingungen werden als *experimenteller Rahmen* bezeichnet.

Als *Modell* bezeichnet man einen Satz von Instruktionen oder Regeln zur Generierung eines Eingabe/Ausgabe-Verhaltens (E/A-Verhalten). Ein *Simulator* stellt

einen Algorithmus oder einen Computer dar, der in der Lage ist, das Modell auszuführen und das entsprechende Verhalten zu erzeugen. Tabelle 2.1 zeigt diese Begriffe nochmals als Übersicht.

Tabelle 2.1: Definition der Basiselemente von Modellierung und Simulation nach [ZPK07]

| Basiselement | Definition |
|---|---|
| Quellsystem | Reales oder virtuelles System, das in der Simulation abgebildet werden soll |
| Experimenteller Rahmen | Bedingungen unter denen das System beobachtet oder untersucht werden soll |
| Modell | Regeln zur Generierung eines E/A-Verhaltens |
| Simulator | Algorithmus zur Erzeugung des E/A-Verhaltens aus dem Modell |

Auf Basis dieser Definitionen können nun verschiedene Beziehungen zwischen den Elementen festgelegt werden. Die beiden wichtigsten sind die *Modellierungsbeziehung Validität* und die *Simulationsbeziehung Simulatorkorrektheit*. Die Modellierungsbeziehung *Validität* bezeichnet eine Beziehung zwischen einem Modell, einem Quellsystem und einem experimentellen Rahmen. Sie bewertet, ob ein Modell das Verhalten eines Quellsystems in für die Fragestellung geeigneter Weise abbildet. In anderen Worten: Die Validitätsbeziehung beantwortet die Frage, ob das Quellsystem und das Modell innerhalb eines gegebenen experimentellen Rahmens unterscheidbar sind. Validität ist gegeben, wenn eine solche Unterscheidung nicht möglich ist. Die *Simulationsbeziehung Simulatorkorrektheit* ist eine Beziehung zwischen Simulator und Modell. Sie beschreibt ob ein Simulator ein Modell korrekt simuliert. Korrektheit ist gegeben, wenn ein Simulator in der Lage ist, das im Modell beschriebene E/A-Verhalten fehlerfrei zu erzeugen.

### Betrachtung von Zeit

Implizit stellt Zeit eine Möglichkeit dar, Beobachtungen oder Ereignisse zu ordnen. Im Zusammenhang von Simulation muss allerdings zwischen zwei Interpretationen von Zeit unterschieden werden, der *physikalischen Zeit* und der *logischen*

*Zeit*. *Physikalische Zeit* ist die Zeit, die in der realen Welt vergeht. Die abstrakte Zeit, die in einer Simulation zur Ordnung von Ereignissen verwendet wird, wird als *logische Zeit* oder als *Zeitbasis der Simulation* bezeichnet. Wie *physikalische Uhren* in der realen Welt werden entsprechend *logische Uhren* zur Messung der Zeit in der Simulation verwendet. Darüber hinaus kann Zeit als *lokal* oder *global* charakterisiert werden, wobei die Lokalität von physikalischer Zeit in diesem Zusammenhang nicht weiter relevant ist. Ist eine logische Zeit nur für eine Komponente oder einen begrenzten Teil eines Systems gültig, wird sie als *lokal* bezeichnet. Ist sie für das gesamte System gültig, wird sie *global* genannt.

Sind in einem Simulationsexperiment mehrere verschiedene Zeiten relevant, so müssen Beziehungen zwischen diesen hergestellt werden, um eine Synchronisation vornehmen zu können. Bei verteilter Simulation müssen zum Beispiel die lokalen, logischen Zeiten der einzelnen Simulatorknoten untereinander synchronisiert werden. Ein weiteres Beispiel sind Realzeitsimulationen. Dabei muss die logische Zeit in der Simulation mit der physikalische Zeit abgeglichen werden. Eine detailiertere Betrachtung der Probleme im Zusammenhang mit Synchronisation folgt in Abschnitt 3.4.

### Modellierungskonzepte und ihre Simulatoren

In diesem Abschnitt werden drei allgemeine Modellierungskonzepte vorgestellt, Discrete-Time, Continuous-Time und Discrete-Event. Darüber hinaus werden die Grundzüge der zugehörigen Simulatoren erläutert.

**Discrete-Time-Modellierung** Discrete-Time-Modellierung geht von der Ausführung der Simulation in diskreten Schritten aus, vergleichbar mit den Konzepten der Differenzengleichungen [Fö94] und der endlichen Automaten [HMU02]. Beginnend in einem bestimmten Zustand zu einem bestimmten Zeitpunkt, wird der Zustand aller Komponenten in jeden Schritt aktualisiert. Das Modell legt die Regeln fest, nach denen basierend auf dem aktuellen Zustand und den Eingangswerten der jeweilige Folgezustand bestimmt wird. Die Ausgangswerte werden entweder auf Basis der Eingangswerte und des Zustands ermittelt[1] oder ausschließlich auf Basis des Zustands[2]. Außerdem wird in jedem Simulationsschritt die logische Zeit um einen diskreten Wert erhöht.

Der grundlegende Algorithmus zur Simulation eines solchen Modells ist sehr einfach. Benötigt werden der Anfangszustand des Modells $q(0)$, Start- und Endzeitpunkt der Simulation $t_i$ und $t_f$, die Größe des Zeitschritts $c$ und die Eingangswerte

[1] Vergl. Mealy-Automat.
[2] Vergl. Moore-Automat.

zu den jeweiligen Zeitpunkten $x(0), ..., x(n)$. In einer Schleife können dann beginnend bei $t_i$ die jeweiligen Folgezustände und Ausgangswerte berechnet werden. In jedem Schritt wird die Zeit um $c$ erhöht bis $t_f$ erreicht ist.

**Continuous-Time-Modellierung** Die Modellierung zeitkontinuierlicher Systeme basiert auf der Spezifikation der Systemeigenschaften in Form von Differenzialgleichungen. Im Gegensatz zur Discrete-Time-Modellierung wird der Folgezustand nicht direkt, sondern in Form von Änderungsraten der Zustandsvariablen spezifiziert. Zu jedem Zeitpunkt ist lediglich die Änderungsrate des Zustands bekannt. Aus dieser Information, zusammen mit einem Startzustand und den Eingangswerten, kann dann der Zustand zu jedem beliebigen Zeitpunkt berechnet werden. In vielen Fällen ist es nicht möglich oder zumindest sehr aufwändig, eine geschlossene Lösung für komplexe Differentialgleichungssysteme zu finden. Deshalb werden Continuous-Time-Modelle in digitalen Computern in der Regel schrittweise unter Verwendung numerischer Integrationsverfahren simuliert.

**Discrete-Event-Modellierung** Die Grundidee bei der Discrete-Event-Modellierung ist, dass nur die interessanten Zeitpunkte in der Simulation berücksichtigt werden. Ähnlich wie bei der Discrete-Time-Modellierung spezifiziert das Modell die Regeln für Zustandsübergänge. Der Simulationsfortschritt erfolgt jedoch nicht in festen Zeitschritten, sondern durch Ereignisse, die über Zeitstempel bestimmten Zeitpunkten zugeordnet sind. Die Ereignisse triggern Zustandsübergänge in einzelnen Komponenten, die dann wiederum neue Ereignisse erzeugen können. Ein Vorteil dieses Vorgehens ist, dass der zeitliche Abstand zwischen zwei Zustandsübergängen abhängig von der tatsächlichen Aktivität des Systems variabel sein kann. Darüber hinaus müssen nicht alle Komponenten eines Systems zu jedem Zeitpunkt aktualisiert werden, sondern nur diejenigen, für die das aktuelle Ereignis relevant ist.

Die Simulationsalgorithmen für Discrete-Event-Modelle basieren in der Regel auf *Ereignislisten*. Darin werden die Ereignisse aufsteigend sortiert anhand ihrer Zeitstempel eingetragen. Der Simulator entnimmt das erste Element aus seiner Ereignisliste, setzt die logische Uhr auf den Zeitpunkt entsprechend des Zeitstempels und aktualisiert die Komponenten, für die das Ereignis relevant ist.

## 2.2 Formale Betrachtung von dynamischen Systemen

Neben der semantischen Beschreibung einzelner Modellierungskonzepte werden in [ZPK07] auch formale Beschreibungen definiert. Die für die vorliegende Arbeit relevanten Teile werden im Folgenden zusammengefasst. Zunächst wird eine formale Definition von Zeit und Dynamik, also Veränderung im Laufe der Zeit, eingeführt. Darauf basierend werden dann formale Modelle definiert, die Systeme und deren dynamisches Verhalten in einem bestimmten Modellierungskonzept beschreiben.

### 2.2.1 Zeitfunktion und dynamische Systeme

Grundlegend für die Beschreibung dynamischer Systeme ist ein Fortschreiten von Zeit. Zeit begreifen wir als etwas unabhängig Fließendes, das es ermöglicht dynamische Veränderungen zu ordnen.

Eine Zeitbasis ist definiert als

$$time = \langle T, < \rangle ,$$

wobei $T$ eine Menge ist und $<$ eine Ordnungsrelation auf $T$. $<$ ist transitiv, irreflexiv und antisymmetrisch, es handelt sich demnach um eine strenge Ordnung.

Diese Ordnung erlaubt es uns Vergangenheit, Gegenwart und Zukunft zu formulieren und insbesondere können so *Zeitintervalle* beschrieben werden. Die Menge $T_{[t_1,t_2]} = \{\tau | \tau \in T, t_1 \leq \tau \leq t_2\}$ beschreibt das abgeschlossene Zeitintervall $[t_1, t_2]$ beginnend mit dem Startzeitpunkt $t_1$ bis zum Endzeitpunkt $t_2$, wobei die Zeitpunkte $t_1$ und $t_2$ selbst Teil der Menge sind. Analog kann mit $T_{(t_1,t_2)}$ oder $(t_1, t_2) = \{\tau | \tau \in T, t_1 < \tau < t_2\}$ ein offenes Intervall beschrieben werden ($t_1$ und $t_2$ sind nicht enthalten). In manchen Fällen spielt es keine Rolle, ob ein Zeitintervall offen oder geschlossen ist. Dann kann mit $T_{\langle t_1,t_2 \rangle}$ oder $\langle t_1, t_2 \rangle$ ein beliebiges Intervall beschrieben werden.

Mit der Zeitbasis können wir nun sich mit der Zeit änderndes Verhalten beschreiben. Sei $A$ eine Menge, z. B. eine Menge von Eingangswerten, und $T$ eine Zeitbasis, dann können wir mit

$$f : T \longrightarrow A \tag{2.1}$$

eine sogenannte *Zeitfunktion*, auch *Trajektorie* genannt, beschreiben. Der Wert von $f$ zum Zeitpunkt $t$ ist durch $f(t)$ gegeben. Ein Beispiel für eine Zeitfunktion ist in Abbildung 2.1 dargestellt.

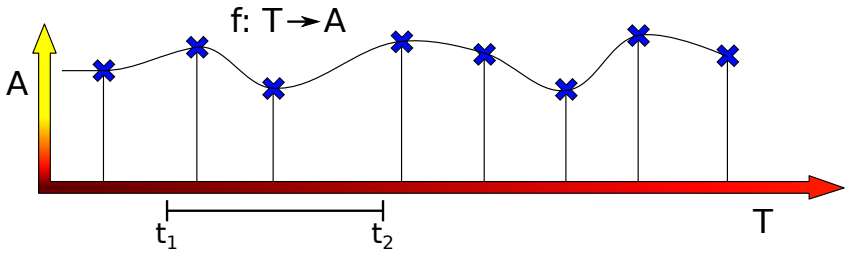



Abbildung 2.1: Zeitfunktion

Eine Einschränkung von $f$ auf eine Teilmenge $T'$ von $T$ wird mit

$$f|T' : T' \longrightarrow A \tag{2.2}$$

bezeichnet, dabei gilt:

$$f|T'(t) := f(t) \forall t \in T'. \tag{2.3}$$

Eine besondere Bedeutung hat die Einschränkung einer Zeitfunktion auf ein Zeitintervall $\langle t_1, t_2 \rangle$, die als *Segment* bezeichnet wird. Wir schreiben dafür

$$\omega : \langle t_1, t_2 \rangle \longrightarrow A \text{ oder } \omega_{\langle t_1, t_2 \rangle}. \tag{2.4}$$

Segmente über eine diskrete Zeitbasis werden als *Sequenz* bezeichnet.

Die Definitionen von Zeitbasis und Zeitfunktion können nun verwendet werden, um ein sich im Laufe der Zeit änderndes, ein *dynamisches*, E/A-System allgemein zu beschreiben.

$S = (T, X, \Omega, Y, Q, \Delta, \Lambda)$

$T$ ist die Zeitbasis

$X$ ist die Menge der Eingangswerte

$Y$ ist die Menge der Ausgangswerte

$\Omega$ ist die Menge aller zulässigen Segmente über $X$ und $T$

$Q$ ist die Menge aller Zustände

$\Delta : Q \times \Omega \longrightarrow Q$ ist die Zustandsübergangsfunktion

$\Lambda : Q \times X \longrightarrow Y$ (oder $\Lambda : Q \longrightarrow Y$ ) ist die Ausgangsfunktion[3]

Basierend auf diesen Definitionen werden in den folgenden Abschnitten Formalismen für die Beschreibung von Simulation eingeführt. Dazu werden Formalismen zur Systemspezifikation verwendet. Diese beschreiben einen Satz von Einschränkungen für die Elemente eines dynamischen Systems. Für die beiden hier

[3] Die Ausgangsfunktion $\Lambda$ kann von Zustand und Eingangswerten oder nur vom Zustand abhängen (vergl. Mealy/Moore-Automat).

relevanten Modellierungskonzepte Discrete-Time und Discrete-Event werden die Formalismen *Discrete Time System Specification (DTSS)* und *Discrete Event System Specification (DEVS)* vorgestellt.

### 2.2.2 DTSS

Das Prinzip der Discrete-Time-Modellierung und -Simulation wurde bereits in Abschnitt 2.1 beschrieben. Es wird nun in einen Formalismus zur Systemspezifikation übertragen. Dieser ist in [ZPK07] unter dem Namen *Discrete Time System Specification* (DTSS) wie folgt beschrieben:

$$\text{DTSS: } M =< X, Y, Q, \delta, \lambda, c > \text{ , mit} \tag{2.5}$$

- $X$ ist die Menge der Eingänge,
- $Y$ ist die Menge der Ausgänge,
- $Q$ ist die Menge der Zustände,
- $\delta : Q \times X \longrightarrow Q$ ist die Zustandsübergangsfunktion,
- $\lambda : Q \longrightarrow Y$ ist die Ausgangsfunktion,
- $c$ ist eine Konstante zur Spezifikation der Zeitbasis $c \bullet \mathbb{N}$.

Ein dynamisches DTSS-System kann damit als eine Struktur

$$S = (T, X, \Omega, Y, Q, \Delta, \Lambda) \tag{2.6}$$

beschrieben werden.

Dabei gilt:

- Die Zeitbasis $T$ ist eine Menge $c \bullet \mathbb{N}$ isomorph zu den natürlichen Zahlen.
- $X$, $Y$ und $Q$ entsprechen $X$, $Y$ und $Q$ des DTSS-Formalismus.
- Die Menge $\Omega$ ist die Menge aller Sequenzen über $X$ und $T$.
- Die Zustands- und Ausgangstrajektorien sind Sequenzen über $Q$ und $T$, beziehungsweise $Y$ und $T$.

Das dynamische Verhalten ist folgendermaßen definiert:

- Gegeben sei ein Eingangssegment $\omega : [t_1, t_2) \longrightarrow X$ und ein initialer Zustand $q$ zum Zeitpunkt $t_1$. Dann ist die globale Zustandsübergangsfunktion $\Delta$

definiert als:

$$\Delta(q,\omega) = \begin{array}{lll} (1) & q & \text{wenn } t_1 = t_2 \\ (2) & \delta(q,\omega(t_1)) & \text{wenn } t_1 = t_2 - c \\ (3) & \delta\left(\Delta(q,\omega[t_1,t_2-c)),\omega(t_2-c)\right) & \text{andernfalls} \end{array} \tag{2.7}$$

- Die Ausgangsfunktion $\Lambda$ eines dynamischen Systems ist:

$$\Lambda(q,x) = \lambda(q) \tag{2.8}$$

im Fall eines Moore-Systems und

$$\Lambda(q,x) = \lambda(q,x) \tag{2.9}$$

im Fall eines Mealy-Systems.

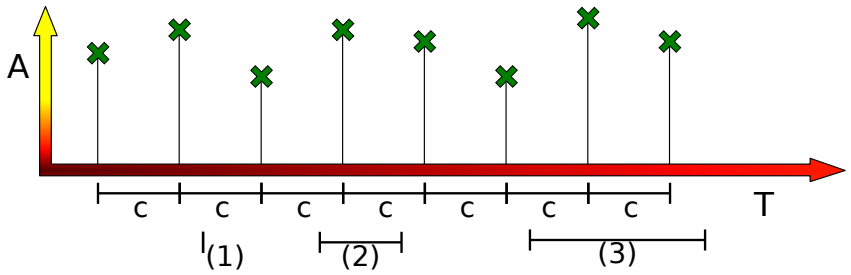



Abbildung 2.2: Zeitfunktion in einem DTSS-System

In der Zustandsübergangsfunktion $\Delta$ werden drei Fälle des Eingangssegments $[t_1, t_2)$ unterschieden. Ist $t_1 = t_2$, so handelt es sich um ein leeres Zeitintervall und der Zustand ändert sich nicht (1). Wenn $t_1 = t_2 - c$ ist, dann gibt es genau einen Zustandsübergang in diesem Intervall. Demnach wird die Zustandsübergangsfunktion genau einmal auf den Zustand $q$ mit den Eingangswerten des Zeitpunkts $t_1$ angewendet (2). Alle anderen Fälle können durch die rekursive Anwendung von $\Delta$ auf die ersten beiden Fälle zurückgeführt werden (3), da nach (2) festgelegt ist, dass es im Intervall $[t_2 - c, t_2)$ genau wie in allen Intervallen $[t_2 - (n+1)\cdot c, t_2 - n \cdot c)$ exakt einen Zustandsübergang gibt. Die Rekursion wird solange fortgesetzt, bis $t_2' = t_2 - nc < t_1$ ist, das Intervall $\left[t_1, t_2'\right)$ also ein leeres Intervall ist und Fall (1) angewendet werden kann.

### 2.2.3 DEVS

Entsprechend ist das Prinzip von Discrete-Event-Modellierung und -Simulation in [ZPK07] im Systemspezifikationsformalismus DEVS ( *Discrete Event System*

*Specification*) abgebildet. Hier wird eine leicht erweiterte Form von DEVS verwendet, DEVS mit Ports. Sie unterscheidet sich vom klassischen DEVS-Formalismus lediglich dadurch, dass statt eines einzelnen Eingangs und eines einzelnen Ausgangs jeweils Mengen von Ein- und Ausgängen und ihrer Werte spezifiziert werden können.

$$DEVS : M =< X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta > \tag{2.10}$$

Dabei gilt:

- $X = \left\{(p, v) | p \in InPorts, v \in X_p\right\}$ ist die Menge der Eingangsports und ihrer Werte.
- $Y = \left\{(p, v) | p \in OutPorts, v \in Y_p\right\}$ ist die Menge der Ausgangsports und ihrer Werte.
- $S$ ist die Menge der Zustände.
- $\delta_{int} : S \longrightarrow S$ ist die interne Zustandsübergangsfunktion.
- $\delta_{ext} : Q \times X \longrightarrow S$ ist die externe Zustandsübergangsfunktion.
  - $Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$ ist gesamte Menge der Zustände.
  - $e$ ist die vergangene Zeit seit dem letzten Zustandsübergang (*elapsed time*).
- $\lambda : S \longrightarrow Y$ ist die Ausgangsfunktion.
- $ta : S \longrightarrow \mathbb{R}_0^+ \cup \infty$ ist die Zeitfortschrittsfunktion.

Zu einem beliebigen Zeitpunkt ist das System in einem Zustand $s \in S$. Wenn kein externes Ereignis auftritt, bleibt das System für den Zeitraum $ta(s)$ in diesem Zustand. Dabei kann $ta(s)$ beliebige reelle Werte von 0 bis einschließlich $\infty$ annehmen. Ist $ta(s) = 0$, so handelt es sich bei $s$ um einen *transitorischen* Zustand. Das bedeutet, dass das System für einen so kurzen Zeitraum in Zustand $s$ bleibt, dass währenddessen kein externes Ereignis auftreten kann. Ist $ta(s) = \infty$ so bedeutet dies, dass das System in Zustand $s$ bleibt, bis ein externes Ereignis auftritt. Ist die Verweildauer verstrichen, ist also $e = ta(s)$, gibt das System den Wert von $\lambda(s)$ aus und wechselt in Zustand $\delta_{int}(s)$. Zu beachten ist an dieser Stelle, dass Ausgaben nur unmittelbar vor einem internen Zustandsübergang möglich sind. Tritt vor Ablauf der Verweildauer, also bei $e \leq ta(s)$, ein externes Ereignis $x \in X$ auf, dann ändert sich der Zustand des Systems zu $\delta_{ext}((s, e), x)$. Wenn mehrere Ereignisse gleichzeitig auftreten, so ist die Reihenfolge, in der sie verarbeitet werden, unbestimmt.

Formal kann die Charakteristik eines dynamischen DEVS-Systems beschrieben werden durch:

$$S = (T, X, \Omega, Y, Q, \Delta, \Lambda) \tag{2.11}$$

Dabei gilt:

- Die Zeitbasis $T$ ist $\mathbb{R}$ (oder eine Teilmenge davon).
- Die Menge der Eingänge $X$ ist $X_{DEVS} \cup \varnothing$, die Menge der Eingänge des DEVS-Systems vereinigt mit dem Symbol $\varnothing$, das für 'kein Ereignis' steht.
- Die Menge der Ausgänge $Y$ ist $Y_{DEVS} \cup \varnothing$ (analog zu den Eingängen).
- Die Menge der Zustände $Q$ ist $Q_{DEVS}$, die gesamte Menge der Zustände des DEVS-Systems.
- $\Omega$ ist die Menge aller DEVS-Segmente über $X$ und $T$.
- $\Delta$ ist folgendermaßen definiert:
  Sei $\omega : \langle t_1, t_2] \longrightarrow X^{\varnothing}$ ein Eingangssegment und Zustand $q = (s, e)$ der Zustand zu Zeitpunkt $t_1$. Dann ist:

$$
\begin{aligned}
&\Delta\left(q, \omega\langle t_1, t_2]\right) = \\
&(1)\quad (s, e + t_2 - t_1) \\
&\qquad \text{wenn } e + t_2 - t_1 < ta(s) \wedge \neg\exists t \in \langle t_1, t_2] : \omega(t) \neq \varnothing \\
&\qquad \text{(kein Ereignis)} \\
&(2)\quad \Delta\left(\left(\delta_{int}(s), 0\right), \omega\left[t_1 + ta(s) - e, t_2\right]\right) \\
&\qquad \text{wenn } e + t_2 - t_1 = ta(s) \wedge \neg\exists t \in \left\langle t_1, t_1 + ta(s) - e\right) : \omega(t) \neq \varnothing \\
&\qquad \text{(ein internes Ereignis)} \\
&(3)\quad \Delta\left(\left(\delta_{ext}\left((s, e + t - t_1), \omega(t)\right), 0\right), \omega\left[t, t_2\right] \setminus \omega(t)\right) \\
&\qquad \text{wenn } \exists t \in \left\langle t_1, min\left(t_2, t_1 + ta(s) - e\right)\right] : \omega(t) \neq \varnothing \\
&\qquad \wedge\neg\exists t' \in \langle t_1, t) : \omega(t') \neq \varnothing \\
&\qquad \text{(ein externes Ereignis)}
\end{aligned}
\tag{2.12}
$$

- $$
\Lambda(q, x) = \begin{cases} \lambda(s) & \text{wenn } e = ta(s) \\ \varnothing & \text{andernfalls} \end{cases}
\tag{2.13}
$$

Die rekursive Definition des dynamischen Verhaltens ist wie folgt zu interpretieren: Zu einem Zeitpunkt $t_1$ ist das System seit $e$ Zeiteinheiten in Zustand $s$. Gibt es im gesamten Zeitraum $\langle t_1, t_2]$ kein externes Ereignis und die gesamte vergangene Zeit $e + t_2 - t_i$ zum Zeitpunkt $t_2$ ist kleiner als $ta(s)$, so gibt es kein Ereignis in diesem Zeitraum. In diesem Fall muss lediglich der Wert der vergangenen Zeit $e$ aktualisiert werden (1). Gibt es vor dem Ablauf der Verweildauer $ta(s)$, also bis zum Zeitpunkt $t_1 + ta(s) - e$, kein externes Ereignis, tritt ein internes Ereignis ein und die interne Zustandsübergangsfunktion wird ausgeführt (2). Tritt vor dem

ersten internen Ereignis ein externes Ereignis auf, so wird die externe Zustandsübergangsfunktion ausgeführt (3). Die Notation $\omega\,[t, t_2] \setminus \omega\,(t)$ steht hier für das Segment $\omega\,[t, t_2]$ in dem das soeben verarbeitete externe Ereignis bei Zeitpunkt $t$ nicht mehr enthalten ist. Zeitpunkt $t$ muss allerdings Teil des Segments bleiben, da die Verweildauer $ta$ des neuen Zustands null sein kann und folglich noch interne Ereignisse zu Zeitpunkt $t$ auftreten können. In beiden Fällen (2 und 3) wird die globale Zustandsübergangsfunktion $\Delta$ rekursiv auf den neuen Zustand und das verbleibende Eingangssegment angewendet.

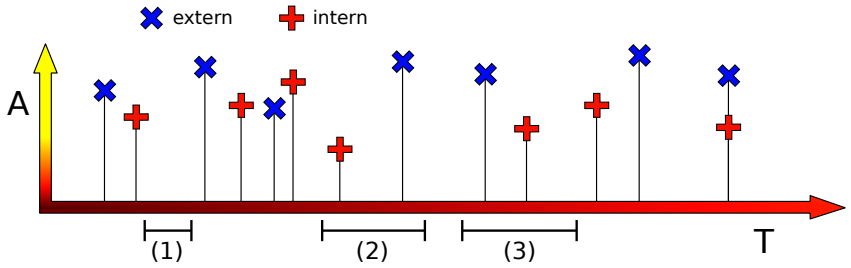



Abbildung 2.3: Zeitfunktion in einem DEVS-System

Aus dieser Definition ist ebenfalls zu ersehen, dass das dynamische Verhalten im Fall von mehreren gleichzeitig auftretenden Ereignissen nicht eindeutig festgelegt ist. Es bleibt dem Simulator überlassen eine Auswahlfunktion zu implementieren, die in diesem Fall die Reihenfolge der Verarbeitung der Ereignisse bestimmt. Es ist jedoch zu beachten, dass 'zum gleichen logischen Zeitpunkt' nicht gleichbedeutend mit 'gleichzeitig' ist. Treten mehrere externe Ereignisse gleichzeitig auf, so ist die Reihenfolge ihrer Verarbeitung nicht festgelegt. Gleiches gilt, wenn ein internes Ereignis gleichzeitig mit einem oder mehreren externen Ereignissen auftritt. Anders ist das jedoch, wenn die Verweildauer $ta(s) = 0$ ist. Stellen wir uns zum Beispiel ein System vor, dass in Zustand $s_1 \in S$ ist und durch ein externes Ereignis zu Zustand $s_2 \in S$ übergeht. Außerdem sei $ta(s_2) = 0$. Dann gibt es ein internes Ereignis wodurch das System in Zustand $s_3 \in S$ übergeht, ohne dass die logische Zeit voranschreitet. Die Zustandsübergänge von $s_1$ nach $s_2$ und von $s_2$ nach $s_3$ finden somit zum gleichen logischen Zeitpunkt statt. Die Verweildauer $ta(s) = 0$ drückt jedoch eine kausale Abhängigkeit aus, so dass der zweite Zustandsübergang nach dem ersten stattfinden muss.

## 2.3 Hardwarebeschreibungssprachen (HDL)

Hardwarebeschreibungssprachen (im Folgenden abgekürzt mit HDL, Hardware Description Language) stellen eine elementare Komponente in modernen Entwick-

lungsprozessen für digitale Hardwarekomponenten dar. Ein wesentliches Unterscheidungsmerkmal zu den üblichen Programmiersprachen für Software ist, dass HDLs neben der rein funktionalen Beschreibung des Verhaltens der Komponente auch die Beschreibung ihrer Struktur und ihres Zeitverhaltens erlauben. Elemente zur expliziten Implementierung von Nebenläufigkeit sowie Kommunikation und Synchronisation zwischen nebenläufigen Subkomponenten stehen als integrale Sprachbestandteile zur Verfügung. Außerdem werden spezielle, hardwarenahe Datentypen bereitgestellt, wie zum Beispiel Bitvektoren oder numerische Datentypen mit frei wählbaren Bitbreiten. Darüber hinaus bieten Hardwarebeschreibungssprachen eine definierte Ausführungssemantik zur Simulation der beschriebenen Hardware.

Neben der Modellierung und Simulation von Schaltungen ist die automatische Synthese zu Netzlisten ein Anwendungsbereich von HDLs. Für die Synthese ist typischerweise nur ein Teil des Sprachumfangs geeignet, das sogenannte Synthese-Subset. Welche Teile im Einzelnen zum unterstützten Synthese-Subset gehören, hängt vom jeweils verwendeten Synthesewerkzeug ab.

Die derzeit am weitesten verbreiteten Hardwarebeschreibungssprachen sind Verilog [VER06], VHDL [VHD11a] und SystemC [iee09]. Die grundlegenden Modellierungselemente und deren Simulationssemantik weisen in den drei genannten Sprachen sehr große Ähnlichkeit auf. Module dienen der Modellierung von struktureller Hierarchie, Prozesse stellen potenziell nebenläufige Funktionsblöcke dar und spezielle Kommunikationskanäle dienen dem Datenaustausch und der Synchronisation zwischen Prozessen. Die Simulationssemantik basiert in allen drei Fällen auf Prozessen, die durch Ereignisse getriggert werden, so dass zur Simulation Discrete-Event-Simulatoren naheliegen und in der Regel auch eingesetzt werden. Näheres dazu wird im Folgenden am Beispiel von VHDL erläutert.

### 2.3.1 VHDL Modellierung

In VHDL werden Module in Form von Paaren aus *Entitäten ('entity')* und *Architekturen ('architecture')*. beschrieben. Die Entität definiert die Schnittstelle des Moduls nach außen, insbesondere seine Ein- und Ausgänge sowie deren Datentypen. Ein Architekturrumpf kann zwei Arten von Anweisungen beinhalten, Anweisungen zur Beschreibung des Verhaltens und Anweisungen zur Beschreibung darunterliegender Strukturen, d.h. zur Instanziierung von Submodulen und deren Verbindung.

Das Verhalten eines Moduls wird in *Prozessen* implementiert. Prozesse beschreiben funktionale Blöcke, die parallel zueinander ausgeführt werden. Die Anweisungen im Inneren eines Prozesses werden sequenziell in einer Endlosschleife ausge-

führt. Neben Anweisungen zur Manipulation oder Zuweisung von Daten und Kontrollstrukturen wie Schleifen oder Bedingungen, stehen in Prozessen *Wait-Anweisungen*. Wait-Anweisungen dienen der Synchronisation der Prozesse untereinander. Sie ermöglichen das *Warten* für einen bestimmten Zeitraum oder auf bestimmte Ereignisse. Die Ausführung des Prozesses wird an dieser Stelle ausgesetzt, bis die entsprechende logische Zeit in der Simulation verstrichen ist oder das Ereignis eintritt. Die Instruktionen, die in einem Prozess zwischen zwei Wait-Anweisungen stehen, werden verarbeitet, ohne dass dabei logische Zeit vergeht. Deshalb muss jeder Prozess mindestens eine Wait-Anweisung enthalten, da sonst kein Zeitfortschritt in der Simulation stattfinden kann.

Die kleinste mögliche Zeiteinheit in der Simulation ist der sogenannte *Deltaschritt* oder *Deltazyklus*. Ein Deltaschritt stellt einen infinitesimal kleinen Zeitschritt dar. Er ermöglicht die Modellierung von Instruktionen als *zeitlich nacheinander*. Es vergeht jedoch keine messbar große Menge logischer Zeit, die logische Uhr schreitet von einem zum nächsten Deltazyklus nicht voran. Somit können zwischen zwei unterschiedlichen Zeitpunkten beliebig viele Deltaschritte liegen.

Die Kommunikation zwischen Prozessen findet über *Signale* statt. Im Unterschied zu gewöhnlichen Variablen haben Signale eine sogenannte *Deltazyklussemantik*. Das bedeutet, dass Signale den neuen Zustand nach einem Schreibzugriff nicht sofort übernehmen, sondern erst im folgenden Deltazyklus. Außerdem erzeugt das Schreiben eines neuen Wertes auf ein Signal ein Ereignis im nächsten Deltazyklus. Dadurch können Prozesse auf Wertänderungen von Signalen warten.

Neben einfachen Signalzuweisungen bietet VHDL *verzögerte Signalzuweisungen*. Dabei wird an der Signalzuweisung zusätzlich eine Verzögerungszeit annotiert. Die Wertänderung und das zugehörige Ereignis finden dann erst nach Ablauf der entsprechenden Zeit in der Simulation statt.

### 2.3.2 VHDL Simulation (DE)

Zur Simulation von VHDL-Modellen werden in der Regel Discrete-Event-Simulatoren eingesetzt, da die Ausführungssemantik das nahelegt. Prozesse warten auf Ereignisse und können ihrerseits durch Signalzuweisungen neue Ereignisse erzeugen. Technisch gesehen kann das Warten für einen Zeitraum auch als Warten auf ein Ereignis betrachtet werden. Der Simulator fügt ein Ereignis am Ende des Zeitraums in die Ereignisliste ein und der Prozess kann auf dieses Ereignis warten.

Die Funktionsweise eines VHDL-Simulators ist in Abbildung 2.4 schematisch dargestellt. Der Ablauf ist in zwei Teile unterteilt, die Initialisierung und die eigentliche Simulationsschleife. Die Initialisierung beginnt mit der Elaboration des

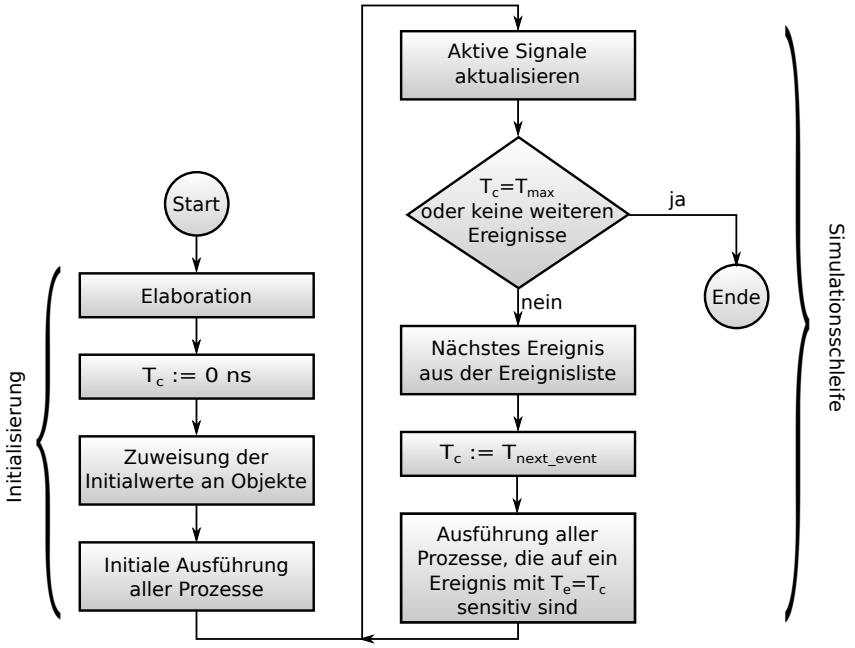



Abbildung 2.4: VHDL Simulationszyklus

Modells. Dabei wird die strukturelle Hierarchie des Modells analysiert und eine entsprechende Objekthierarchie aus Modulen, Prozessen und Signalen im Simulator aufgebaut. Dann wird die Simulationszeit mit null initialisiert und allen Signalen und Variablen werden ihre initialen Werte zugewiesen. Schließlich werden alle Prozesse im Modell einmal gestartet und bis zur ersten Wait-Anweisung ausgeführt[4].

Im Anschluss beginnt die eigentliche Simulationsschleife. Die erste Phase ist die sogenannte Aktualisierungsphase. Die Werte aller Signale, die während der vorangegangenen Ausführung der Prozesse beschrieben wurden, werden aktualisiert und die entsprechenden Ereignisse in die Ereignisliste eingetragen. Ist im Anschluss daran der vorgegebene Endzeitpunkt in der Simulation erreicht oder sind keine weiteren Ereignisse in der Ereignisliste vorhanden, so wird die Simulation beendet. Ist dies nicht der Fall wird das nächste Ereignis, das mit dem kleinsten Zeitstempel, aus der Ereignisliste entnommen. Gibt es mehrere Ereignisse mit diesem Zeitstempel werden sie alle entnommen. Dann wird die logische Uhr auf den Wert des Zeitstempels gesetzt. Die neue logische Uhrzeit ist nie kleiner als die vorhergehende, da keine Ereignisse in der logischen Vergangenheit erzeugt werden können. Sie kann jedoch gleich sein. Aufgrund der Deltazyklussemantik der Signale ist in

[4]Die initiale Ausführung einzelner Prozesse kann durch die Annotation *postponed* unterbunden werden.

der Aktualisierungsphase die Erzeugung von Ereignissen zum aktuellen logischen Zeitpunkt möglich. Es findet also ein Deltaschritt statt; die logische Uhrzeit bleibt jedoch gleich. Natürlich kann der Zeitstempel echt größer als die aktuelle logische Zeit sein; dann findet ein *echter* Zeitschritt statt.

Die letzte Phase, bevor die Simulationsschleife erneut mit der Aktualisierungsphase beginnt, ist die Ausführungsphase. Alle Prozesse, die auf eines der aktuellen Ereignisse sensitiv sind, werden ausgeführt. Die Ausführung beginnt an der Stelle, an der die Prozesse zuletzt unterbrochen wurden, und endet bei der nächsten Wait-Anweisung.

Hier zeigt sich eine grundlegende Grenze bei der Simulation paralleler Systeme. Aufgrund der Beschränkungen der Computerplattform, auf der die Simulation ausgeführt wird, können Prozesse nicht beliebig im Sinne der physikalische Zeit parallel ablaufen. In der VHDL-Simulation werden alle Prozesse, die in einer Ausführungsphase laufen sollen, nacheinander ausgeführt. Die Reihenfolge ist dabei nicht festgelegt und kann vom Simulator beliebig gewählt werden. Von entscheidender Bedeutung ist lediglich, dass alle Prozesse, die gleichzeitig – also auch im selben Deltaschritt – laufen sollen vor der nächsten Aktualisierungsphase ausgeführt werden.

Die durchgängige Verwendung von Signalen zur Interprozesskommunikation und deren Deltazyklussemantik erlauben es somit, Modelle zu implementieren, die sich in der Simulation deterministisch verhalten, auch wenn die Reihenfolge der Prozessausführung innerhalb eines Deltaschritts undefiniert ist.

### 2.3.3 Statische Simulation

Ein Konzept zur Beschleunigung der Simulation von digitalen Hardwarekomponenten ist die statische oder kompilierte Simulation [FLLO95, Pen06]. Dabei wird das Modell der Hardwarekomponente in Programmcode übersetzt, häufig C oder C++, und die Ausführung dieses Programms entspricht der Simulation des Modells. Für die Ablaufplanung der einzelnen Prozesse im Modell wird kein Discrete-Event-Simulator verwendet, sondern die Kontroll- und Datenabhängigkeiten werden direkt im Programmcode berücksichtigt. So kann bereits beim Kompilieren des Programms eine mehr oder weniger statische Ablaufplanung realisiert werden. Durch die Analyse von Datenabhängigkeiten können bei der Transformation des Modells Optimierungen, wie zum Beispiel die Eliminierung von überflüssigen Signalen oder Zuweisungen, durchgeführt werden. Zusammen mit der statischen Ablaufplanung kann so die Simulationsgeschwindigkeit erhöht werden, jedoch sind die Möglichkeiten der Beobachtung von internen Zuständen oder Variablen in der Regel eingeschränkt.

#### Taktzyklus-basierte Simulation

Eine Variante der statischen Simulation ist die Taktzyklus-basierte Simulation für taktgetriebene Hardwarekomponenten. Das Grundkonzept ähnelt der statischen Simulation, denn auch in diesem Fall wird das Modell in Programmcode übersetzt, in dem die Abhängigkeiten im Modell berücksichtigt werden. Außerdem wird ebenfalls eine statische Ablaufplanung realisiert, allerdings nicht für die gesamte Simulation sondern nur für ein Taktereignis. Alle Deltazyklen, die an ein Taktereignis anschließen, werden statisch eingeplant. Die Taktereignisse selbst bleiben dynamisch. Das beobachtbare Verhalten an den Taktgrenzen entspricht der Simulation des ursprünglichen Modells, das Verhalten zwischen den einzelnen Deltazyklen, die an eine Taktflanke anschließen, ist jedoch von außen nicht mehr sichtbar.

Neben akademischen Ansätzen wie [WLL95] gibt es auch kommerzielle Werkzeuge [Car, NMS+04], die diese Technik nutzen, um eine schnellere Simulation von Hardwaremodellen zu realisieren.

### 2.3.4 Logiksynthese von HDL-Beschreibungen

Neben der Simulation ist die wichtigste Eigenschaft von HDL-Modellen die Möglichkeit der *automatischen Hardwaresynthese*. Hardwaresynthese bedeutet in diesem Zusammenhang die Übersetzung der HDL-Beschreibung einer Schaltung in eine sogenannte *Netzliste*, eine Art Schaltplan bestehend aus Logikgattern, Speicherzellen und deren Verdrahtung [Ash01].

Normalerweise ist nicht der gesamte Umfang der Hardwarebeschreibungssprachen zur Synthese geeignet. Deshalb werden sogenannte *Synthese-Subsets* definiert, in denen die Grundmenge der Sprachelemente, die von Synthesewerkzeugen unterstützt werden sollen, festgelegt ist und ihre Synthesesemantik definiert wird [VHD04]. Der Sprachumfang wird dazu in drei Teile aufgeteilt: unterstützte Sprachelemente, nicht unterstützte Sprachelemente sowie Elemente, die bei der Synthese ignoriert werden. Zum ersten Teil gehören zum Beispiel hierarchische Modelle, logische Operationen auf bitorientierten Datentypen oder Interprozesskommunikation über Signale. Der zweite Teil, die nicht unterstützten Sprachelemente, sind solche, die in einem synthetisierbaren Modell nicht vorkommen dürfen. Der Grund dafür ist in der Regel, dass es für diese Elemente keine eindeutige oder effiziente Synthesesemantik gibt. Beispiele dafür sind das Rechnen mit Fließkommazahlen, die Verwendung von Zeigertypen oder Variablen, die von mehreren Prozessen gemeinsam genutzt werden. Der dritte Teil umfasst Sprachelemente, die in Modellen häufig zu Analysezwecken und zur Fehlersuche verwendet werden, wie zum Beispiel Assertion- oder Berichtsfunktionen. Sie dürfen in synthetisierba-

ren Modellen verwendet werden, werden aber von den Synthesewerkzeugen einfach ignoriert. Der Entwickler muss selbst darauf achten, dass relevanten Eigenschaften der modellierten Hardware nicht von der korrekten Umsetzung solcher Sprachelemente abhängt.

Zu den bei der Synthese ignorierten Sprachelementen gehören auch Zeitannotationen. Dies umfasst einerseits Wait-Anweisungen zum Warten für einen bestimmten Zeitraum, aber auch *verzögerte Signalzuweisungen*. Letztere sind Signalzuweisungen, bei denen die Wertänderung nicht direkt nach der Ausführung der Anweisung stattfindet, sondern erst nachdem der annotierte Zeitraum in der Simulation verstrichen ist. Zeitannotationen werden oft in der Implementierung von Testumgebungen, aber auch zur Analyse von Signallaufzeiten in kombinatorischen Prozessen verwendet. Wenn solche Anweisungen in Modellen, die zur Synthese vorgesehen sind, verwendet werden, so muss sichergestellt sein, dass sie lediglich der Analyse dienen. Die korrekte Funktion des Modells darf nicht vom annotierten Zeitverhalten abhängig sein, da die Zeitannotationen bei der Synthese ignoriert werden und somit auch keinen Einfluss im Syntheseergebnis haben.

### 2.3.5 Zusammenfassung HDL

Hardwarebeschreibungssprachen sind spezielle Programmiersprachen, die besondere Elemente zur Beschreibung digitaler Hardware zur Verfügung stellen. Hervorzuheben ist dabei die Möglichkeit zur expliziten Beschreibung von Zeitverhalten und Nebenläufigkeit. Zur Simulation des beschriebenen Verhaltens ist eine definierte Ausführungssemantik fester Bestandteil der Sprache. Diese entspricht der Discrete-Event-Simulationssemantik, weshalb HDL-Simulatoren üblicherweise als Discrete-Event-Simulatoren realisiert sind. Ein weiteres Merkmal ist die Möglichkeit eine Teilmenge der Sprachelemente automatisch zu Netzlisten zu synthetisieren.

## 2.4 MATLAB/Simulink

Der Entwickler von MATLAB und Simulink, das Unternehmen MathWorks, beschreibt das Werkzeug folgendermaßen: “Simulink ist eine Blockdiagrammumgebung für die Mehrdomänen-Simulation und Model-Based Design. Simulink unterstützt den Entwurf und die Simulation auf Systemebene und ermöglicht außerdem die automatische Codegenerierung und das kontinuierliche Testen und Verifizieren von Embedded Systems.

Simulink umfasst einen Grafikeditor, benutzerdefinierbare Blockbibliotheken sowie Solver für die Modellierung und Simulation von dynamischen Systemen. Simulink ist in MATLAB integriert, sodass Sie MATLAB-Algorithmen in Modelle aufnehmen und Simulationsergebnisse wiederum in MATLAB analysieren und weiterverarbeiten können.

Die Hauptmerkmale von Simulink sind:

- Grafikeditor für das Erstellen und Verwalten von hierarchischen Blockdiagrammen
- Bibliotheken mit vordefinierten Blöcken für die Modellierung von zeitkontinuierlichen und zeitdiskreten Systemen
- Simulations-Engine mit ODE-Solvern mit fester und variabler Schrittweite
- Scopes und Datendisplays für die Anzeige von Simulationsergebnissen
- Projekt- und Datenverwaltungswerkzeuge für die Verwaltung von Modelldateien und -daten
- Modellanalysewerkzeuge für die Verfeinerung der Modellarchitektur und die Steigerung der Simulationsgeschwindigkeit
- MATLAB-Funktionsblock für das Importieren von MATLAB-Algorithmen in Modelle
- Legacy Code-Werkzeug für das Importieren von C- und C++-Code in Modelle." [5]

In der Entwicklung eingebetteter Systeme ist Simulink weit verbreitet. Insbesondere seine Vielseitigkeit durch zahlreiche Erweiterungen macht es zu einem wertvollen Werkzeug beim Anwendungs- und Algorithmenentwurf auf Systemebene.

Die Modellierung in Simulink geschieht blockbasiert. Dazu steht eine Reihe von Bibliotheken mit vordefinierten Blöcken zur Verfügung. Zusätzlich gibt es die Möglichkeit eigene Blöcke zu erstellen, entweder als Komposition aus anderen Blöcken oder in Form von Programmcode in MATLAB [Theb] oder C/C++.

### 2.4.1 Simulation in Simulink

Die Simulation von Simulink-Modellen geschieht datenflussorientiert und zeitgetrieben [Thec]. Der Simulator bestimmt basierend auf dem Datenfluss im Modell eine Ordnung der einzelnen Blöcke. Bei Schleifen im Datenfluss können zusätzlich Blockprioritäten verwendet werden. Die Simulation findet dann in aufeinanderfol-

[5] Simulink Produktbeschreibung [Thed]

genden Zeitschritten statt. In jedem Zeitschritt werden die Blöcke des Modells in der Reihenfolge der zuvor bestimmten Ordnung evaluiert. Die Länge des Zeitraums zwischen zwei Schritten, genannt *Schrittweite*, hängt von mehreren Faktoren ab. Einer ist die Grundschrittweite. Sie wird aus Schrittweiten ermittelt, die am Gesamtmodell oder an einzelnen Blöcken annotiert werden können. Des Weiteren spielt die Art des für die Simulation verwendeten *Solvers* eine Rolle, insbesondere bei Modellen, die kontinuierliche Zustände enthalten.

### Solver

Die Simulation eines dynamischen Systems in Simulink basiert auf der Berechnung des Systemzustands in aufeinanderfolgenden Zeitschritten. In Simulink stehen dazu verschiedene Methoden, sogenannte *Solver*, zur Verfügung, um unterschiedlichen Eigenschaften von Modellen und Anforderungen an Simulationsexperimente gerecht werden zu können. Sie lassen sich in zwei Grundkategorien einordnen, Solver mit fester Schrittweite und Solver mit variabler Schrittweite.

**Feste Schrittweite** Solver mit fester Schrittweite (Fixed-step Solver) berechnen die Modellzustände in regelmäßigen Zeitintervallen. Die Schrittweite bleibt dabei über den gesamten simulierten Zeitraum gleich. Im Allgemeinen führt eine kleinere Schrittweite zu einer größeren Genauigkeit der Simulationsergebnisse, aber gleichzeitig auch zu einer Verringerung der Simulationsgeschwindigkeit.

**Variable Schrittweite** Solver mit variabler Schrittweite (Variable-step Solver) können während der Simulation dynamisch auf das Verhalten des Modells reagieren. Ändern sich die Zustände des Modells schnell, so wird die Schrittweite verringert, um die Simulationsgenauigkeit zu erhöhen. Ändern sich die Zustände des Modells langsam, wird die Schrittweite vergrößert, um die Simulationsgeschwindigkeit zu steigern. So lässt sich für die Gesamtsimulation häufig ein besserer Kompromiss zwischen Genauigkeit und Geschwindigkeit finden.

**Kontinuierliche und diskrete Solver** Enthält ein Modell Blöcke mit kontinuierlichen Zuständen, muss zur Simulation auch ein kontinuierlicher Solver verwendet werden. Dieser berechnet die Zustände in den einzelnen Zeitschritten mittels numerischer Integration. In Simulink stehen verschiedene kontinuierliche Solver zur Verfügung, die unterschiedliche Methoden zur numerischen Integration verwenden. Wenn ein Modell keine kontinuierlichen Zustände hat, wird ein diskreter Solver verwendet.

**Erkennung von Nulldurchgängen** Die Verwendung eines Solvers mit variabler Schrittweite kann in Kombination mit kontinuierlichen Zuständen zu unerwünschtem Verhalten führen. Wenn der zeitliche Verlauf eines Zustands Unstetigkeitsstellen aufweist, wird das als sehr schnelle Änderung des Zustands interpretiert. Also wird im Bereich um diese Stelle eine sehr kleine Schrittweite verwendet. Simulink verwendet eine Technik zur Erkennung von Nulldurchgängen (Zero-Crossing-Detection), um solche Unstetigkeitsstellen genau zu bestimmen ohne dazu viele kleine Zeitschritte in seiner Umgebung vorzunehmen.

#### Simulink-Simulationszyklus

Abbildung 2.5 zeigt die Grundzüge des Simulationszyklus in Simulink. Nach der Initialisierung besteht die eigentliche Simulationsschleife aus vier Schritten. Im ersten Schritt werden die neuen Werte der Ausgänge aller Blöcke in der zuvor ermittelten Ordnung berechnet. Im zweiten Schritt wird der Solver aufgerufen, um den Zustand des Modells zu aktualisieren. Beinhaltet ein Modell nur diskrete Zustände, so werden lediglich die *Update-Methoden* der einzelnen Blöcke aufgerufen. Gibt es kontinuierliche Zustände im Modell, so wird die *Derivatives-Methode* verwendet, um den neuen Zustand durch numerische Integration zu ermitteln. Auch hier wird die Reihenfolge durch die zuvor bestimmte Ordnung der Blöcke festgelegt. Der dritte Schritt wird nur bei kontinuierlichen Zuständen benötigt. Er dient zur Ermittlung von Unstetigkeitsstellen. Im vierten Schritt wird schließlich die Größe des nächsten Zeitschritts berechnet. Ist die spezifizierte Endzeit der Simulation erreicht, endet die Simulation. Andernfalls beginnt die Schleife von vorne.

### 2.4.2 S-Function-Interface

S-Functions bieten einen Mechanismus zur Integration generischer, benutzerdefinierter Funktionen in Simulink. Sie erlauben die Implementierung des Verhaltens eines Blocks mittels Programmcode, der in MATLAB, C/C++ oder Fortran geschrieben sein kann. Das S-Function-Interface definiert eine Reihe von Callback-Methoden, die der Benutzer implementieren muss. Die Methoden sind einzelnen Phasen im Simulationszyklus zugeordnet, z. B. `mdlOutputs()` der Output-Phase oder `mdlUpdate()` der Update-Phase. Der Simulator ruft dann in den einzelnen Phasen die jeweilige Methode auf.

Neben der Implementierung neuer Funktionen sind S-Functions auch eine häufig genutzte Methode zur Integration bereits bestehender Teile eines Systems, die zuvor in C oder C++ implementiert wurden.

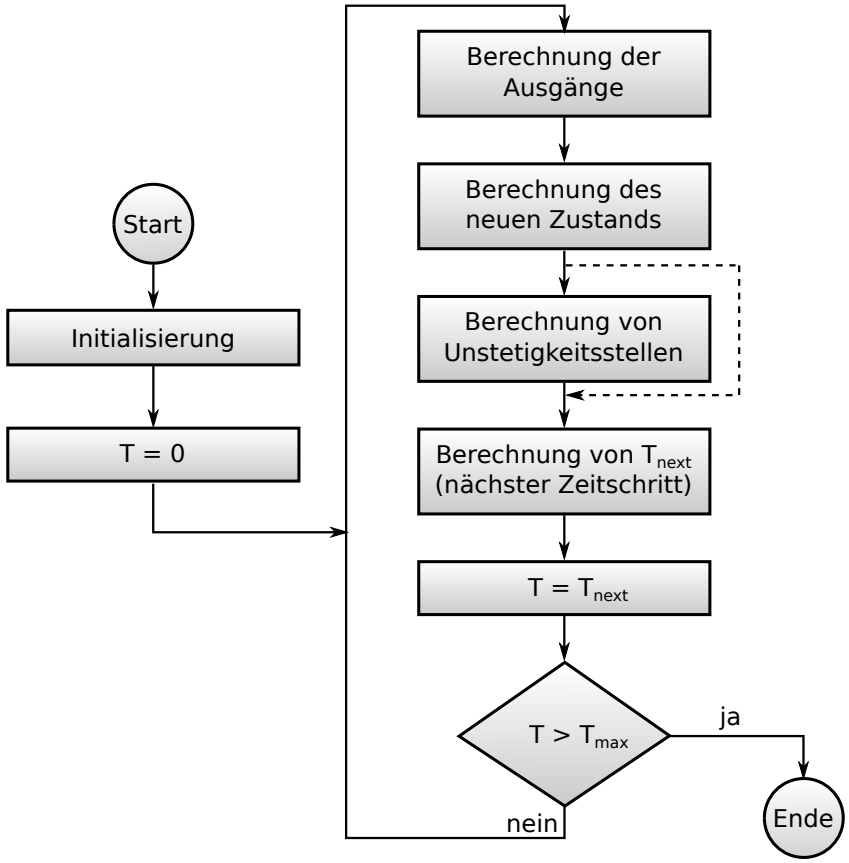



Abbildung 2.5: Simulink Simulationszyklus

### 2.4.3 Zusammenfassung der Begriffe

Zur besseren Übersicht werden an dieser Stelle die wesentlichen Begriffe zusammengefasst, die im Zusammenhang der Simulink-Simulation verwendet werden:

**Simulationsschritt** Ein Schleifendurchlauf im Simulationszyklus.

**Schrittweite** Länge des Zeitraums zwischen zwei Simulationsschritten.

**Samplerate oder Samplefrequenz** Kehrwert der Schrittweite.

**Samplezeitpunkt** Ein logischer Zeitpunkt, an dem eine Evaluation des Modells stattfindet.

**Solver** Lösungsverfahren zur numerischen Simulation. Hat Einfluss auf die die Schrittweite in der Simulaiton.

**Zero-Crossing-Detection** Methode zur Bestimmung von Unstetigkeitsstellen in kontinuierlichen Zuständen.

**S-Function** Benutzerdefinierter Block in Simulink, dessen Verhalten in Form von MATLAB oder C/C++ Code beschrieben ist.

### 2.4.4 Zusammenfassung Simulink

Die besonderen Stärken von Simulink liegen in der anschaulichen Modellierung und Simulation von Mehrdomänen-Systemen. Zur Modellierung steht eine Vielzahl an vorbereiteten Blockbibliotheken aus verschiedenen Anwendungsbereichen zu Verfügung. Die Simulation geschieht zeitgetrieben, vergleichbar mit einer Discrete-Time-Simulation, und kann dank einer großen Auswahl an Solvern gut an die Anforderungen eines bestimmten Systems angepasst werden.

## 2.5 Ko-Simulation

Ko-Simulation[6] ist eine Simulationsmethode, die es erlaubt einzelne Komponenten eines Systems in unterschiedlichen Simulationswerkzeugen zu simulieren. Die Simulationswerkzeuge laufen dabei gleichzeitig und tauschen untereinander Informationen aus. Zunächst wird nur die Ko-Simulation auf einem gemeinsamen Rechner betrachtet. Verteilte Simulation auf mehreren Rechnern folgt später in Abschnitt 3.4.

Die Simulationswerkzeuge sind in unserem Fall Simulink und HDL-Simulatoren. Der Teil des Systems, der in Simulink modelliert ist, wird in Simulink simuliert. Der Teil des Systems, der in einer Hardwarebeschreibungssprache modelliert ist, wird in einem HDL-Simulator simuliert. Um die Datenverbindungen zwischen den beiden Modellteilen zu realisieren, muss ein Datenaustausch zwischen den Werkzeugen stattfinden. Außerdem muss eine Synchronisation der logischen Uhren vorgenommen werden, damit Wertänderungen auf diesen Datenverbindungen zum richtigen Zeitpunkt in das Modell eingebracht werden können. Zusätzlich zu den beiden Simulatoren wird demnach ein Werkzeug zur Synchronisation und zum Datenaustausch benötigt. Neben der Synchronisation ist eine wesentliche Aufgabe dieses Werkzeugs die Konvertierung der auszutauschenden Daten. In Simulink wird im Allgemeinen mit Fließkommazahlen gerechnet. In speziellen Fällen können auch logische Werte, Ganzzahlen oder Fixpunktzahlen verwendet werden. Im Gegensatz dazu werden in HDL-Modellen Bitvektoren oder bitakkurate numerische Datentypen verwendet. Das Ko-Simulationswerkzeug muss deshalb eine Abbildung der jeweiligen Datentypen definieren und während der Simulation die entsprechenden Konvertierungen vornehmen.

[6] Kooperative Simulation

Der entscheidende Vorteil dieses Konzepts ist, dass beide Teile des Modells in ihrer nativen Simulationsumgebung simuliert werden. Es müssen demnach keine Änderungen an den Teilmodellen vorgenommen werden. Außerdem steht die volle Funktionalität des HDL-Simulators zur Verfügung, zum einen bezüglich des unterstützten Sprachumfangs, zum anderen aber auch bezüglich zusätzlicher Analysefunktionen wie zum Beispiel Waveformtracing oder Ähnliches. Verglichen mit der Simulation in nur einem Simulator hat das Konzept allerdings auch einen deutlichen Nachteil. Durch den Datenaustausch und die Synchronisation zwischen den Simulatoren entsteht zusätzlicher Rechenaufwand, der zu einer Verlangsamung der Gesamtsimulation führt. Weitere Einbußen in der Simulationsgeschwindigkeit entstehen dadurch, dass beide Simulatoren abwechselnd auf demselben Prozessor ausgeführt werden. Bei jedem Wechsel muss der Zustand des jeweiligen Simulators im Speicher des Prozessors wiederhergestellt werden. Zudem wirken sich die Wechsel zwischen den Werkzeugen negativ auf die Effizienz der Prozessorcaches aus.

### 2.5.1 Ko-Simulationsblöcke in Simulink

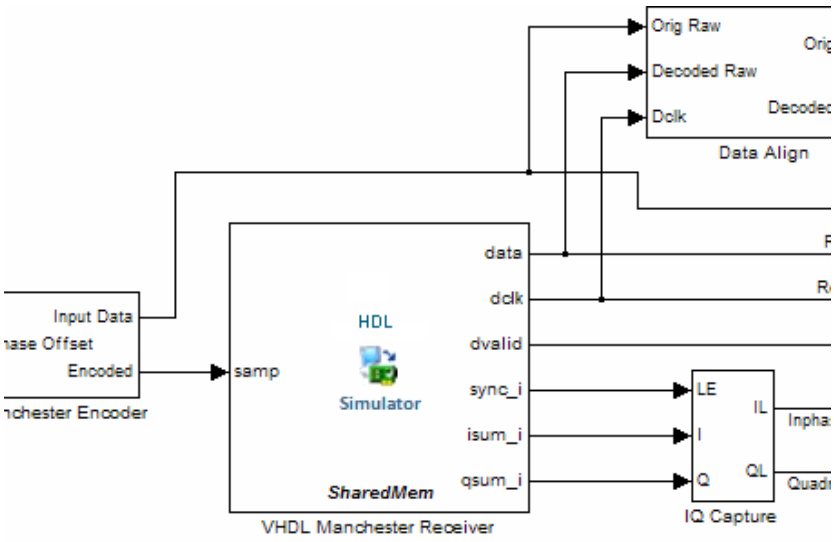



Abbildung 2.6: HDL Verifier Block in Simulink

In der Regel wird die Ko-Simulation in Simulink mit Ko-Simulationsblöcken realisiert (s. Abbildung 2.6). Der Ko-Simulationsblock repräsentiert das Modell der HDL-Komponente und wird wie jeder andere Block verwendet. Er wird dem Datenfluss entsprechend in die Evaluationsreihenfolge des Simulators eingeordnet und im Rahmen des Simulationszyklus ausgewertet. Der Simulink-Simulator

steuert die Evaluation des Ko-Simulationsblocks und dieser steuert den HDL-Simulator.

Bei der Evaluation werden die Werte der Eingänge des Blocks passend konvertiert und an den HDL-Simulator übertragen. Dann wird der HDL-Simulator so angesteuert, dass er die Simulation der HDL-Komponente für den seit dem letzten Simulationsschritt vergangenen logischen Zeitraum durchführt. Dieser Ablauf ist in Abbildung 2.7 schematisch dargestellt.

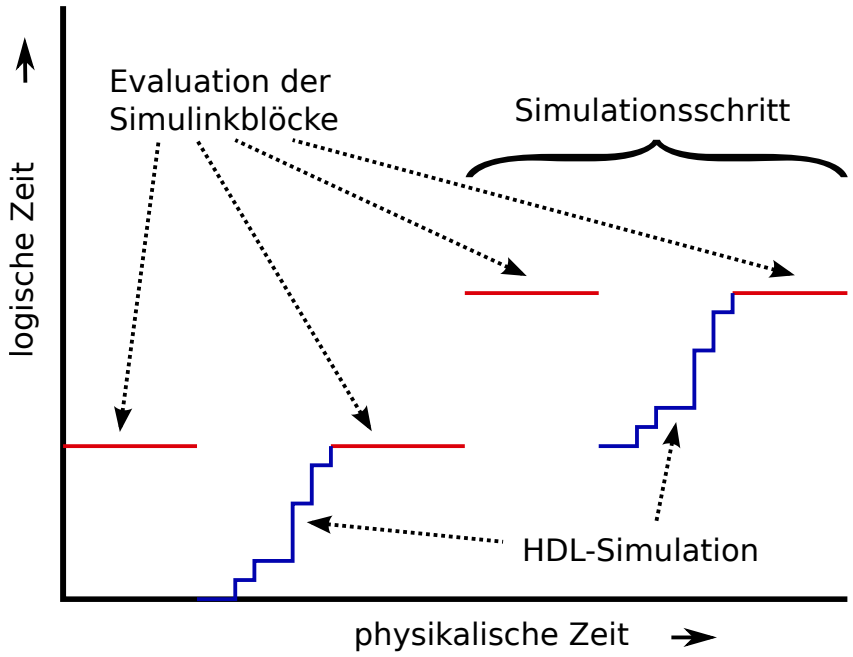



Abbildung 2.7: Ablaufschema Ko-Simulation

### 2.5.2 Aufwand für Ko-Simulation

Der zusätzliche Rechenaufwand, der bei der Ko-Simulation entsteht, hängt von verschiedenen Eigenschaften des zu simulierenden Modells ab. Genaue Messungen des Mehraufwands sind nicht ohne Weiteres möglich. Deshalb wird im Folgenden lediglich eine qualitative Bewertung der Eigenschaften angegeben.

#### Schrittweite der Simulation

Der zusätzliche Aufwand entsteht durch Datenaustausch, Kontextwechsel und Synchronisation zwischen den beiden Simulatoren. Da dies in jedem Simulations-

schritt stattfindet, hängt der zusätzliche Rechenaufwand für die Ko-Simulation direkt von der Schrittweite in der Simulation ab. Bei einer kleineren Schrittweite gibt es im gleichen simulierten Zeitraum mehr Synchronisationspunkte und damit mehr Aufwand.

#### Menge der auszutauschenden Daten

Der Aufwand für den Datenaustausch wird darüber hinaus durch die Menge der auszutauschenden Daten bestimmt. Eine größere Datenmenge erhöht sowohl den Aufwand für die Konvertierung der Daten als auch den für ihre Übertragung. Die Datenmenge bestimmt sich ihrerseits aus der Signalschnittstelle der HDL-Komponente, das heißt aus der Anzahl ihrer Ein- und Ausgänge und deren Bitbreiten. Somit hat die Größe der Signalschnittstelle Einfluss auf den Aufwand, der in der Ko-Simulation zusätzlich benötigt wird.

#### Rechenkomplexität des Modells

Der absolute Mehraufwand für die Ko-Simulation ist unabhängig von der Komplexität der für die eigentliche Simulation der Teilmodelle notwendigen Berechnungen. Der Mehraufwand spielt deshalb bei kleineren Modellen eine größere Rolle. Mit steigendem Aufwand für die eigentliche Simulation der Teilmodelle nimmt der relative Anteil des Mehraufwands am Gesamtaufwand ab.

### 2.5.3 Simulationsgeschwindigkeit und Schrittweite bei Ko-Simulation

In Simulink bedeutet eine größere Schrittweite in der Regel auch eine schnellere Simulation. Bei einer größeren Schrittweite werden weniger Schritte benötigt, um den gleichen Zeitraum zu simulieren. Da bei vielen Blöcken der Rechenaufwand zur Aktualisierung von Zustand und Ausgangswerten weitgehend unabhängig von der Größe des Zeitschritts ist, kann durch größere Simulationsschritte eine höhere Simulationsgeschwindigkeit erreicht werden.

Bei einer Ko-Simulation ist dieser Zusammenhang anders. Der Rechenaufwand für die Simulation der HDL-Komponente ist unabhängig von der Schrittweite in Simulink. Er hängt von der Komplexität der Komponente und den zu verarbeitenden Ereignissen ab. Lediglich der Mehraufwand der Ko-Simulation ist von der Anzahl der Zeitschritte abhängig, da Synchronisation, Datenaustausch und Kontextwechsel in jedem Zeitschritt stattfinden. Zusätzlich dazu kann es vorteilhaft

sein größere Zeitabschnitte in einem Stück zu simulieren, da das im Vergleich zu häufigen Wechseln zwischen den Simulatoren zu einer besseren Ausnutzung der Prozessorcaches führt. Dennoch fällt der Vorteil in der Simulationsgeschwindigkeit, der bei einer Vergrößerung der Schrittweite erwartet werden kann, bei einer Ko-Simulation deutlich geringer aus.

# 3 Stand der Technik

Nachdem im vorangegangenen Kapitel die Grundlagen der Simulation von Hardwarekomponenten und Simulink-Modellen eingeführt wurden, beleuchten die folgenden Abschnitte den aktuellen Stand der Technik im Bereich der Simulation heterogener Systeme. Verschiedene, existierende Ansätze im Zusammenhang der vorliegenden Arbeit werden vorgestellt und bezüglich der Erfüllung der gestellten Anforderungen untersucht. Außerdem werden verschiedene Ansätze zur formalen Beschreibung der Simulationssemantik von Hardwarebeschreibungssprachen und Simulink betrachtet und auf ihre Eignung für die in dieser Arbeit notwendigen Analysen geprüft.

## 3.1 Ko-Simulation

Nach der grundlegenden Betrachtung der Ko-Simulation von Simulink und Hardwarebeschreibungssprachen in Abschnitt 2.5 werden nun einige konkrete Ansätze und Werkzeuge aus diesem Bereich vorgestellt.

### 3.1.1 HDL-Verifier

Die HDL-Verifier Toolbox von Mathworks [Thea] ist eine erweiternde Blockbibliothek zu Simulink. Sie stellt Blöcke zur Ko-Simulation von Simulink mit den kommerziellen HDL-Simulatoren ModelSim von Mentor [Men] und Incisive von Cadence [Cad] zur Verfügung.

Der große Vorteil dieser Lösung ist, dass der volle Funktionsumfang der HDL-Simulatoren zur Verfügung steht. Neben dem großen Umfang der unterstützten Sprachelemente können auch die Möglichkeiten zur Ko-Simulation von mehreren HDLs[1], zur Visualisierung von Signalverläufen und zur Fehlersuche verwendet werden. Nachteilig ist in erster Linie der große Einfluss auf die Simulationsgeschwindigkeit. Da zwei umfangreiche Simulationswerkzeuge abwechselnd ausgeführt werden, ist der Mehraufwand für die Kontextwechsel besonders hoch. Au-

[1] Z. B. eine HDL-Komponente die teilweise in VHDL und teilweise in Verilog implementiert ist.

ßerdem ist die Verwendung von variablen Schrittweiten bei der Ko-Simulation mit HDL-Verifier nicht möglich.

In einem industriellen Umfeld werden hohe Kosten und höhere Know-How-Anforderungen bei der Verwendung der Toolbox häufig als weitere Nachteile genannt. Hohe Kosten ergeben sich daraus, dass Lizenzen für Simulink, den HDL-Simulator und das Ko-Simulationswerkzeug benötigt werden. Die Verwendung dieser Lösung wird dadurch erschwert, dass neben Kenntnissen in Simulink auch zumindest grundlegende Kenntnisse in der Verwendung des HDL-Simulators erforderlich sind.

### 3.1.2 SystemC und Simulink

Zur Ko-Simulation von SystemC-Modellen und Simulink stehen zwei unterschiedliche Konzepte zur Verfügung. Beide nutzen die Tatsache aus, dass SystemC eine C++-Bibliothek ist und SystemC-Modelle in Form von C++-Code implementiert werden.

Das erste Konzept verwendet die *MatlabEngine*, eine C-API zur Steuerung von MATLAB und Simulink [BTZ05]. Die SystemC-Simulation ist der Simulations-Master und spezielle SystemC-Module steuern die Simulink-Simulation und den Datenaustausch zwischen SystemC und Simulink.

Im zweiten Ansatz ist Simulink der Simulations-Master. Die SystemC-Komponenten werden mit Hilfe spezieller C/C++-S-Functions in Simulink instanziiert. Diese S-Functions steuern die Simulation der SystemC-Komponenten und übernehmen die Aufgaben der Synchronisation und des Datenaustauschs zwischen den Simulatoren. Die Literatur kennt eine Reihe von Realisierungen dieses Konzepts [HON08, BBN+06, HBAV12, MKBMG11], die sich in ihrer Flexibilität und in verschiedenen Details bezüglich Synchronisation und Datenaustausch unterscheiden. Allen Realisierungen ist jedoch gemeinsam, dass sie auf der Verwendung mehrerer Simulatoren basieren und den daraus resultierenden Mehraufwand für Kontextwechsel, Synchronisation und Datenaustausch mitbringen.

### 3.1.3 Verilog und MATLAB/Simulink

Unter [Rom] ist ein freies Werkzeug namens *Vmodel* erhältlich, das die Simulation von Verilog-Modellen in MATLAB und Simulink ermöglicht. Es basiert auf Verilator [Ver], einem quelloffenen Werkzeug zur Erzeugung von zyklusbasierten C++-Simulationsmodellen aus synthetisierbaren Verilog-Modellen. Vmodel erzeugt S-

Function-Wrapper, um die mit Verilator generierten C++-Modelle in MATLAB oder Simulink zu integrieren.

Neben dem Werkzeug selbst sind lediglich eine englische Installationsanleitung sowie eine einfache Anleitung zur Benutzung auf Russisch öffentlich zugänglich. Die interne Funktionsweise und die zugrunde liegenden Konzepte wurden nicht veröffentlicht. Deshalb kann eine Bewertung und Abgrenzung nur auf experimenteller Basis vorgenommen werden. Bereits einige Tests mit kleineren Verilog-Beispielen haben ergeben, dass bei der Simulation im Vergleich mit einer Ko-Simulation mit HDL-Verifier signifikante Unterschiede im Zeitverhalten auftreten. Bei gleicher Stimulation der Eingänge ist der zeitliche Verlauf der Ausgangssignale nicht identisch. Deshalb genügt das Werkzeug den in Abschnitt 1.2 gestellten Anforderungen nicht. Das beobachtbare Zeitverhalten wird nicht exakt wiedergegeben.

## 3.2 RTL-nach-TLM-Abstraktion

In [BFP10] wird eine Methode zur Integration von RTL-Komponenten[2] in TLM-Modelle[3] vorgestellt. TLM ist eine Methode zur abstrakteren Modellierung insbesondere der Kommunikationsanteile in Hardwarebeschreibungssprachen [CG03]. Dabei kommunizieren Prozesse nicht über einzelne Signale auf Bitebene. Stattdessen wird die Kommunikation über sogenannte Transaktionen realisiert. Eine Transaktion kann zum Beispiel die Übertragung eines Datenworts über einen Bus oder auch ein Zugriff auf eine Speicherkomponente sein. Da eine Transaktion eine ganze Reihe von Signalzugriffen, wie zum Beispiel ein vollständiges Busprotokoll, ersetzen kann, ist eine deutlich schnellere Simulation möglich. Im Gegenzug sinkt jedoch die zeitliche Genauigkeit in der Simulation.

Prinzipiell können TLM- und RTL-Modelle im selben Simulator simuliert werden, praktisch handelt es sich jedoch um eine Form von Ko-Simulation. Die TLM-Simulation ist in erster Linie über Transaktionen und Methodenaufrufe getrieben und nicht über Signalereignisse. Deshalb wird in [BFP10] ein Konzept zur automatischen Abstraktion von RTL-Komponenten zu äquivalenten TLM-Beschreibungen vorgestellt. Die Transformation basiert auf Methoden, die vergleichbar sind mit den Transformationen, die bei der Taktzyklus-basierten Simulation verwendet werden. Anstelle der Taktzyklen werden in diesem Fall jedoch funktional zusammenhängende Einheiten zusammengefasst. Diese funktionalen Einheiten werden dann in Form einer TLM-konformen Methodenschnittstelle nach außen zur Verfügung

[2] Register Transfer Level
[3] Transaction Level Modelling

gestellt. Dadurch ist eine schnellere Simulation der Komponente, insbesondere aber auch eine schnellere Simulation des Gesamtsystems möglich, da die Umsetzung von Transaktionen auf eine RTL-Signalschnittstelle nicht mehr notwendig ist. Auch bei diesem Konzept sinkt jedoch die zeitliche Genauigkeit, da das Zeitverhalten der RTL-Komponente teilweise abstrahiert wird. Eine Anwendung dieses Konzepts kommt in unserem Fall also nicht in Frage, weil Anforderung 3 aus Abschnitt 1.2 nicht erfüllt werden kann.

## 3.3 Heterogene Simulationsumgebungen

Neben der klassischen Ko-Simulation gibt es spezielle Simulationsumgebungen für heterogene Systeme, sogenannte *heterogene Simulationsumgebungen*. Mit ihnen können unterschiedliche Modellierungs- und Simulationskonzepte in einem Gesamtmodell eingesetzt werden. Dabei wird für einzelne Module oder Subsysteme individuell festgelegt, welches Modellierungskonzept verwendet wird. Zur Simulation des Gesamtmodells stehen dann Schnittstellen zur Verfügung, mit deren Hilfe die einzelnen Simulationskonzepte in ein übergeordnetes Simulationskonzept integriert werden. Prominente Vertreter in dieser Kategorie sind das Ptolemy II Rahmenwerk und die Analog/Mixed-Signal-Erweiterungen zu Hardwarebeschreibungssprachen.

### 3.3.1 Ptolemy II

Ptolemy II [Pto, EJL+03] ist ein quelloffenes Rahmenwerk zur experimentellen Analyse Aktor-orientierter Systementwürfe. Aktoren sind Softwarekomponenten, die parallel ausgeführt werden und über Nachrichten kommunizieren. Ein Modell ist eine hierarchische Zusammenschaltung von Aktoren. Die Simulationssemantik des Modells wird durch sogenannte *Direktoren* bestimmt, die ein bestimmtes Ausführungsmodell implementieren. Es stehen unter anderem Direktoren für Kahn-Prozessnetzwerke (KPN), Discrete-Event (DE), Datenfluss (SDF) und Continuous-Time (CT) zur Verfügung. In einem Modell kann jede Hierarchieebene einen eigenen Direktor haben. So lassen sich verschiedene Teile eines Modells in unterschiedlichen Modellierungskonzepten realisieren. Zur Simulation bietet Ptolemy II einen übergeordneten Simulator, der nach dem Discrete-Event-Konzept arbeitet. Alle Direktoren können in diesen Simulator integriert werden. So wird zum Beispiel in [Par95] gezeigt, wie Kahn-Prozessnetzwerke in Ptolemy II simuliert werden können. In [CHL97] wird die kombinierte Simulation von Discrete-Event-Modellen und Datenflussmodellen untersucht, und in [GBA+07] wird beschrieben,

wie Datenflussmodelle und Zustandsautomaten gemeinsam in Ptolemy II simuliert werden können.

Problematisch bei Ptolemy II ist, dass es sich um eine eigene Modellierungsumgebung handelt, die verglichen mit Simulink oder VHDL im industriellen Umfeld wenig verbreitet ist. Folglich ist auch die Anzahl bereits vorhandener Modelle relativ gering. Die Integration existierender Modelle aus anderen Modellierungsumgebungen kann in der Regel nicht automatisiert durchgeführt werden und ist mit erheblichem manuellem Aufwand verbunden. Für die hier gegebene Aufgabe ist somit die Anforderung der automatischen Anwendbarkeit nicht erfüllbar. Darüber hinaus kann die Anforderung, dass die Simulation von Simulink aus kontrolliert wird bei der Verwendung von Ptolemy II nicht erfüllt werden.

### 3.3.2 HDL-Erweiterungen für heterogene Systeme (AMS)

Für viele Hardwarebeschreibungssprachen gibt es Erweiterungen zur Modellierung analoger Schaltungsteile und zum Teil auch nicht-elektrischer physikalischer Zusammenhänge. Beispiele dafür sind VHDL-AMS [VHD09], Verilog-AMS [Ver09] und SystemC AMS [Sys10]. Sie erweitern den Sprachumfang der HDLs um Elemente zur Beschreibung von Continuous-Time- und Discrete-Time-Modellen. Diese Modellteile können dann direkt zusammen mit den digitalen Schaltungsteilen, die als Discrete-Event-Modell beschrieben sind, simuliert werden. Prinzipiell ist die Simulation eine Ko-Simulation, wobei der DE-Simulator die Kontrolle über die Gesamtsimulation übernimmt und der CT- bzw. DT-Simulator mit dem DE-Simulator synchronisiert wird. Der zusätzliche Aufwand für die Ko-Simulation ist in diesem Fall geringer, da nur eine Modellierungssprache zum Einsatz kommt und die Konvertierung der Datentypen normalerweise nicht erforderlich ist.

Die in Abschnitt 1.2 gestellten Anforderungen können jedoch auch in diesem Fall nicht vollständig erfüllt werden. Zur Integration von beliebigen Simulink-Modellen wäre eine Ko-Simulation notwendig, wie sie in Abschnitt 3.1.1 vorgestellt wurde. Eine Konvertierung von Simulink-Modellen nach zum Beispiel SystemC AMS ist nur mit Einschränkungen möglich. Darüber hinaus ist auch bei dieser Lösung die Kontrolle der Simulation nicht in Simulink möglich.

#### Simulink-nach-SystemC-AMS-Transformation

In [KXG+11] wird ein Konzept zur Generierung von SystemC-AMS-Modulen aus Simulink-Modellen vorgestellt. Basierend auf der C-Code-Generierung durch die Simulink Coder Toolbox von Mathworks werden sogenannte TDF-Module erzeugt, deren Simulationssemantik im Wesentlichen der von Simulink entspricht.

Das Verhalten des Moduls ist in Form einer C++-Funktion beschrieben, die zyklisch, mit einer festen Rate ausgeführt wird. Bei diesem Ansatz gibt es jedoch einige Einschränkungen, die in erster Linie auf Einschränkungen des Simulink-Coder zurückzuführen sind. Neben einigen Blocktypen wird auch die Verwendung von variablen Schrittweiten zur Code-Generierung nicht unterstützt.

## 3.4 Verteilte Simulation

Eine Möglichkeit zur Beschleunigung von Simulationsexperimenten ist die verteilte Simulation. Das Modell wird in mehrere Teile partitioniert und die einzelnen Teile werden auf unterschiedlichen Rechnern ausgeführt. Datenaustausch und Synchronisation finden in der Regel über ein Netzwerk statt. Die Beschleunigung der Gesamtsimulation hängt von verschiedenen Punkten ab. Die Anzahl der Synchronisationspunkte und die Menge der auszutauschenden Daten spielen dabei eine wichtige Rolle, insbesondere weil die Daten über ein Netzwerk versendet werden müssen. Außerdem beeinflusst die Parallelisierbarkeit des Modells die erreichbare Beschleunigung. Die parallele Ausführung kann nur dann genutzt werden, wenn sie nicht durch Abhängigkeiten zwischen den parallelen Teilen des Modells unterbunden wird.

Für die Simulation von Hardwaremodellen stehen für eine verteilte Simulation spezielle Compiler zur Verfügung [Nar98, LS00]. Sie sind in der Lage ein Modell geeignet zu partitionieren und integrieren die notwendigen Elemente zur Synchronisation und zum Datenaustausch. In der Praxis werden derartige Ansätze jedoch selten eingesetzt, da der Mehraufwand für Kommunikation und Synchronisation meist größer ist als der Gewinn durch die Parallelisierung.

Darüber hinaus stehen auch Rahmenwerke zur Verfügung, die losgelöst von einem speziellen Simulator Regeln und Grundstrukturen für eine verteilte Simulation definieren. Ein Beispiel dafür ist High Level Architecture (HLA) [55510]. Neben der grundsätzlichen Architektur eines Simulationsexperiments sind darin Methoden zur Synchronisation und zum Datenaustausch standardisiert. Dieses Konzept wird häufig zur Realisierung sehr großer Simulationsexperimente wie zum Beispiel Verkehrs- oder Schlachtfeldszenarien mit vielen heterogenen und komplexen Teilnehmern eingesetzt.

Eine weitere, offensichtliche Eigenschaft von verteilter Simulation ist, dass dazu mehrere Rechner benötigt werden. Dies ist nicht immer erwünscht und führt gegebenenfalls zu zusätzlichen Kosten.

### 3.4.1 Synchronisationspunkte und Garantien

Bei verteilter Simulation haben die Anzahl von Synchronisationspunkten und deren zeitlicher Abstand einen großen Einfluss auf die Geschwindigkeit der Gesamtsimulation. Bei wenigen Synchronisationspunkten mit großem Abstand können die einzelnen Teile der Simulation länger unabhängig voneinander parallel arbeiten. Dadurch verbessert sich das Verhältnis zwischen Gewinn durch parallele Ausführung und Mehraufwand für die Synchronisation.

Eine Methode zur Reduktion von Synchronisationspunkten sind *optimistische* Synchronisationsverfahren. Bei *konservativen* Synchronisationsverfahren werden kausale Abhängigkeiten immer eingehalten. Jeder der parallelen Teile des Modells führt den nächsten Simulationsschritt erst dann aus, wenn sichergestellt ist, dass keiner der anderen Modellteile diesen Schritt beeinflussen kann. *Optimistische* Verfahren erlauben Abweichungen von der kausalen Ordnung. Simulationsschritte können unter der Annahme ausgeführt werden, dass es keine Einflüsse aus anderen Modellteilen gibt. Sollte sich diese Annahme später als falsch herausstellen, wird der Zustand des Teilmodells vor den optimistisch ausgeführten Schritten wiederhergestellt und die darauf folgenden Schritte werden unter den neuen Bedingungen erneut ausgeführt. Diese Verfahren können demnach nur angewendet werden, wenn der Simulator das Rücksetzen auf einen früheren Zustand erlaubt. Bei HDL-Simulatoren ist dies in der Regel nicht möglich oder mit erheblichem Aufwand verbunden, da nicht nur der Zustand des Modells, sondern auch der Zustand des Simulators abgespeichert und wiederhergestellt werden muss.

Um bei der Verwendung konservativer Synchronisationsverfahren die Anzahl der Synchronisationspunkte zu reduzieren, wurde das Konzept der *Garantien* entwickelt. Es ist in [Meh94] ausführlich dargestellt und folgendermaßen definiert: Eine Garantie $G$ von einem Modellteil $LP_i$ an einen anderen Modellteil $LP_j$ ist die Zusicherung von $LP_i$ an $LP_j$, dass $LP_j$ während der restlichen Simulation keine Ereignisse mehr von $LP_i$ erhalten wird, deren Zeitstempel kleiner als $G$ ist. Hat ein Modellteil Garantien für alle Ereignisse, von denen er abhängig ist, bekommen, so kann er bis zum Zeitpunkt der kleinsten Garantie weiter simulieren, ohne auf die anderen Modellteile warten zu müssen.

## 3.5 Formale Semantik von HDLs und Simulink

Um einen Vergleich der Simulationssemantik von Hardwarebeschreibungssprachen und Simulink vornehmen zu können, müssen diese zunächst formal beschrieben werden. Im Folgenden werden existierende formale Beschreibungen vorgestellt und bezüglich ihrer Eignung im Kontext dieser Arbeit überprüft.

### 3.5.1 Formale Semantik von HDLs

In den Standards zu Hardwarebeschreibungssprachen ist die Simulationssemantik in der Regel in Form von beschreibendem Text gegeben. Um die Korrektheit von Modellen und Simulatoren untersuchen zu können, wurde diese Semantik in der Vergangenheit in verschiedenen Arbeiten formal erfasst. In [DJS93] wird eine Abbildung von VHDL-Modelle auf Petri-Netze vorgeschlagen. In [Tas93] wird ein neuer Formalismus zur Beschreibung von VHDL-Modellen eingeführt, und in [BFK94] wird eine weiterentwickelte Variante davon präsentiert.

Ähnliche Ansätze sind auch für SystemC zu finden. In [RHG+01] wird die Simulationssemantik von SystemC in Form von abstrakten Zustandsautomaten definiert, und in [Sal03] wird eine denotationale Semantik speziell für synchrone SystemC-Modelle entwickelt.

Darüber hinaus gibt es auch Ansätze zur Abbildung von HDLs auf den DEVS-Formalismus. In [CBFB03] wird eine Transformation von VHDL-Beschreibungen zu DEVS-Modellen vorgestellt. Ziel der Arbeit ist die einfache Erzeugung von Fehlermodellen aus VHDL-Beschreibungen und deren Simulation. Dazu wird jede VHDL-Instruktion auf ein atomares DEVS-Modell abgebildet. Alle atomaren Modelle aus den Instruktionen, die zu einem Prozess gehören, werden dann zu einem gekoppelten DEVS-Modell zusammengefasst. Die Prozessmodelle können schließlich zu einem Modell des Gesamtsystems zusammengekoppelt werden. Das Ergebnis ist eine Repräsentation des Gesamtmodells, die in einem generischen DEVS-Simulator simuliert werden kann.

Eine ähnliche Abbildung, allerdings für eine Untermenge von VHDL-AMS, die als sAMS-VHDL (simple Analog Mixed Signal VHDL) bezeichnet wird, ist in [MW05] beschrieben. Hier ist das Ziel die Simulation heterogener VHDL-AMS-Systeme in CD++ [Wai02]. CD++ ist ein Werkzeug zur Modellierung und Simulation von DEVS-Modellen. Seine Basis ist eine Bibliothek von C++-Klassen, von denen die Bestandteile des Modells abgeleitet werden. So wird zum Beispiel ein atomares DEVS-Modell abgeleitet von der Klasse *Atomic*. Dessen Eigenschaften können dann durch das Überladen von Methoden spezifiziert werden. Eine Überladung der Methode *externalFunction* beschreibt zum Beispiel die externe Zustandsübergangsfunktion. In [MW05] werden die einzelnen Bestandteile aus sAMS-VHDL auf diese Struktur übertragen und zu einem Gesamtmodell gekoppelt.

All diesen Ansätzen ist gemeinsam, dass sie für die in dieser Arbeit notwendigen Untersuchungen zu detailliert sind. Der Fokus in diesen Arbeiten liegt in der formalen Beschreibung der Simulatoren und in der formalen Verifikation von Modellen. Dazu wird das konkrete Verhalten der Modelle auf der Ebene einzelner Instruk-

tionen oder Prozesse analysiert, die dann zu Netzen zusammengekoppelt werden. Für eine korrekte Simulation oder die formale Verifikation bestimmter Eigenschaften ist eine Betrachtung auf dieser Ebene notwendig. Für ganze Hardwarekomponenten werden diese formalen Beschreibungen jedoch sehr umfangreich und ihre Analyse aufwendig. Für die Untersuchungen in der vorliegenden Arbeit ist dieser Detailgrad nicht geeignet. Der Vergleich mit anderen Simulationssemantiken wird dadurch lediglich erschwert, ohne dass die Betrachtung des konkreten Verhaltens auf Instruktionsebene einen Vorteil hätte. Hier ist eine einfache Formalisierung des abstrakten und generellen Zeitverhaltens von HDL-Komponenten, die allgemein die Entstehung von Ereignissen und die Zeitpunkte von Zustandsübergängen beschreibt, besser geeignet.

### 3.5.2 Formale Semantik von Simulink

Für Simulink gibt es keinen Standard, sondern lediglich die Beschreibung der Simulationssemantik durch den Hersteller Mathworks. Dennoch gibt es Ansätze zur formalen Beschreibung der Simulation und zur formalen Verifikation von Modellen in der Literatur. So wird zum Beispiel in [Tiw02] die Semantik von Stateflow-Modellen formal beschrieben. In [BC12] wird eine formale Semantik für Simulink-Modelle beschrieben, die sowohl kontinuierliche als auch diskrete Anteile beinhalten.

Diese Formalismen wurden zur formalen Verifikation entwickelt, und deshalb ist, ähnlich wie bei den Formalismen für Hardwarebeschreibungssprachen, der Detailgrad für die Untersuchungen in dieser Arbeit ungeeignet. Sie erfassen die Interaktion der einzelnen Blöcke untereinander und mit dem Solver sehr genau. Für den Vergleich mit der Semantik von HDLs wird aber nur eine Beschreibung des Zeitverhaltens eines einzelnen diskreten Blocks benötigt.

## 3.6 Zusammenfassung Stand der Technik

Alle derzeit existierenden Lösungen sind nicht in der Lage die gestellten Anforderungen vollständig zu erfüllen (s. Abbildung 3.1).

Die klassische Ko-Simulation bietet eine sehr allgemeine Lösung, die das Verhalten der HDL-Komponente exakt wiedergibt. Es ist jedoch auch ein sehr großer Mehraufwand erforderlich, um die Koordination und Kommunikation der beiden Simulatoren untereinander zu realisieren. Das vorgestellte Werkzeug zur Integration von Verilog in Simulink gibt das Verhalten der Hardwarekomponente nicht exakt wieder; eine weitere Untersuchung der Simulationsgeschwindigkeit wurde

| | Ko-Simulation | | | | Heterogene Umgebungen | |
|---|---|---|---|---|---|---|
| | HDL-Verifier | SystemC-Simulink | Verilog-Simulink | RT-nach-TLM | Ptolemy II, etc. | HDL-Erw. AMS |
| Simulations-geschwindigkeit | X | X | ? | O | O | O |
| Automatische Integration aller Modellteile | O | O | O | X | X | X |
| Exaktes Verhalten | O | O | X | X | O | O |
| Unterstützung synthetisierbare HDL-Modelle | O | O | O | O | ? | O |
| Keine Einschränkung in Simulink | O | O | O | ? | ? | X |
| Kontrolle in Simulink | O | O | O | ? | X | X |

Abbildung 3.1: Stand der Technik: erfüllte Anforderungen

deshalb nicht durchgeführt. Eine etwas andere Variante der Ko-Simulation ist die in Abschnitt 3.2 vorgestellte Methode zur Abstraktion eines RTL-Modells in eine TLM-Komponente. Sie bietet eine Transformation des ereignisgetriebenen RTL-Modells in eine Komponente, die über eine Methodenschnittstelle getrieben ist. Eine Integration in Simulink wäre denkbar, aber das Zeitverhalten der Komponente wird im abstrahierten Modell nicht exakt wiedergegeben.

Bei heterogenen Simulationsumgebungen liegt das wesentliche Problem in der Integration aller Teilmodelle in die neue Umgebung und der Integration des Gesamtmodells in Simulink. Diese Schritte erfordern manuelle Transformationen der Modelle und sind nicht automatisch anwendbar. Da diese Transformationen nicht vollständig erfasst sind, kann keine zuverlässige Aussage zu Einschränkungen bei der Modellierung gemacht werden. Lediglich die Übersetzung eines Simulink-Modells nach SystemC-AMS ist in der Literatur ausführlich beschrieben; dabei ergeben sich die Einschränkungen aus den Einschränkungen der zugrunde liegenden C-Code-Generierung.

Verteilte Simulation dient zwar auch der Beschleunigung von Simulation, wird jedoch im Bereich der eingebetteten Systeme nur selten eingesetzt. Die starken Abhängigkeiten in den Modellen führen dazu, dass eine geeignete Partitionierung der Modelle schwierig ist und der Mehraufwand für den Datenaustausch den Gewinn durch die Parallelisierung aufhebt.

Zur Formalisierung der Simulationssemantik von Hardwarebeschreibungssprachen und Simulink gibt es umfangreiche Literatur. Diese Ansätze wurden jedoch entwickelt um formale Analysen und Verifikation von Modelleigenschaften zu ermöglichen. Das Verhalten von Modellen muss dazu sehr detailliert abgebildet werden. Dieser hohe Detailgrad ist für die Untersuchungen in dieser Arbeit nicht sinnvoll. Hier soll ein Vergleich des abstrakten Zeitverhaltens vorgenommen werden, und dazu ist eine einfachere Formalisierung der erforderlichen Eigenschaften besser geeignet.

# 4 Problemstellung und eigener Ansatz

In Abschnitt 1.2 wurde die grundlegende Aufgabe dieser Arbeit motiviert und beschrieben. Ziel ist es, das Modell einer Hardwarekomponente in eine Simulink-Simulation zu integrieren. Das Modell der Hardwarekomponente liegt in Form von synthetisierbarem HDL-Code vor, das Umgebungsmodell als beliebiges Simulink-Modell. Die Gesamtsimulation soll aus Simulink heraus gesteuert werden können und die Integration und notwendige Transformationen des Modells sollen automatisch durchführbar sein. Außerdem soll die Simulationsgeschwindigkeit in der Gesamtsimulation möglichst hoch sein.

Die Untersuchung existierender Ansätze in diesem Problemfeld im vorangegangenen Kapitel hat ergeben, dass lediglich die Ko-Simulation in der Lage ist die Grundanforderungen zu erfüllen. Dabei ergibt sich jedoch ein hoher Mehraufwand für den Datenaustausch, die Synchronisation und die Kontextwechsel zwischen den beiden Simulatoren. Der vorliegende und häufig anzutreffende Anwendungsfall, die Simulation einer synthetisierbaren Hardwarebeschreibung in einem Simulink-Umgebungsmodell, bietet allerdings einige besondere Eigenschaften. Es stellt sich hier nun die Frage, ob nicht diese besonderen Eigenschaften ausgenutzt werden können, um eine effizientere Integration der HDL-Komponente in Simulink zu erreichen, den Mehraufwand für Datenaustausch, Synchronisation und Kontextwechsel zu verringern und somit die Simulationsgeschwindigkeit in der Gesamtsimulation zu verbessern.

Abbildung 4.1 zeigt beispielhaft zwei Zeitfunktionen, die eines Discrete-Time-Systems (DT) und die eines Discrete-Event-Systems (DE). Im Discrete-Time-System geschehen die Zustandsübergänge mit festem zeitlichem Abstand $c$, Im Discrete-Event-System geschehen die Zustandsübergänge immer dann, wenn ein Ereignis auftritt.

In heterogenen Simulationsumgebungen (Abschnitt 3.3) wird das Discrete-Time-System im Kontext des Discrete-Event-Systems ausgeführt. Dies ist schematisch in Abbildung 4.2 gezeigt. Im Wesentlichen werden dabei Ereignisse eingeführt, die einen festen zeitlichen Abstand haben und den logischen Zeitpunkten zugeordnet sind, an denen die Zustandsübergänge in der Discrete-Time-Simulation

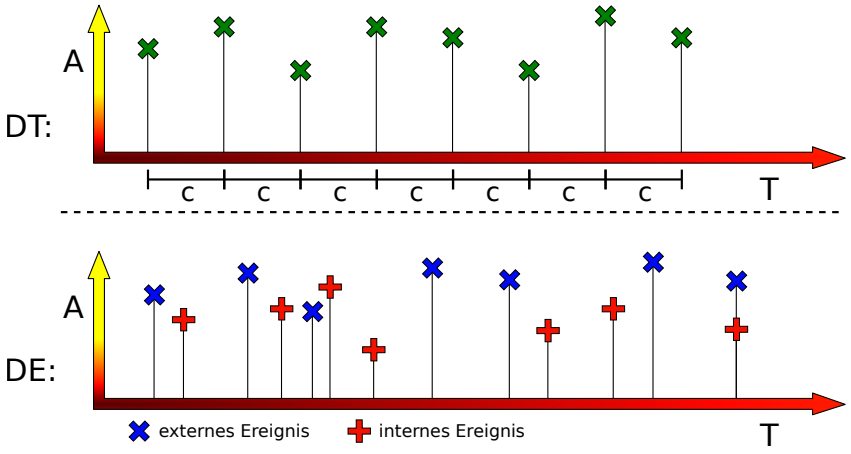



Abbildung 4.1: Zeitfunktionen in Discrete-Time (DT) und Discrete-Event (DE).

stattfinden würden. Das Discrete-Time-System kann dann wie ein Prozess im Discrete-Event-System behandelt werden.

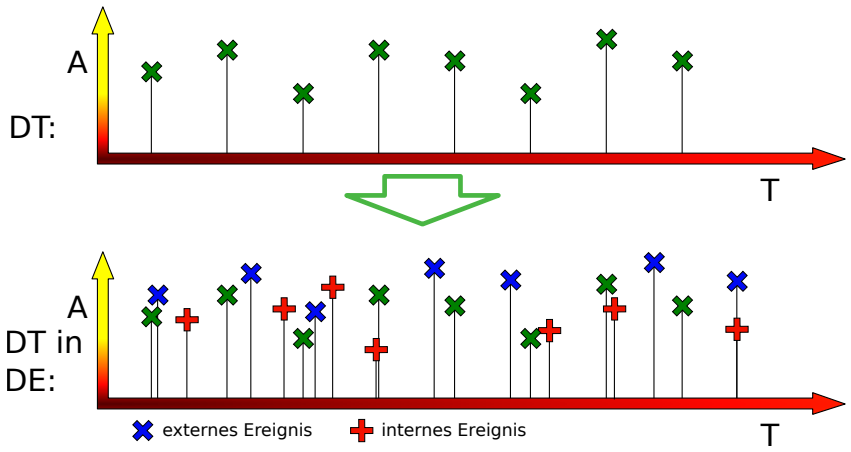



Abbildung 4.2: Zeitfunktionen und heterogene Simulation.

In dieser Arbeit soll der umgekehrte Fall realisiert werden. Das Discrete-Event-System soll im Kontext eines Discrete-Time-Simulators ausgeführt werden. Eine Möglichkeit ist, wie bereits erwähnt, die Ko-Simulation. Dabei wird die Simulation der HDL-Komponente in einem vollwertigen HDL-Simulator vorgenommen und die beiden Simulatoren werden untereinander synchronisiert (s. Abbildung 4.3).

Aus dem gegebenen Anwendungsfall ergeben sich allerdings einige spezielle Eigenschaften, die für die Simulation der HDL-Komponente gültig sind. Aufgrund dieser Eigenschaften kann automatisch ein optimiertes Simulationsmodell

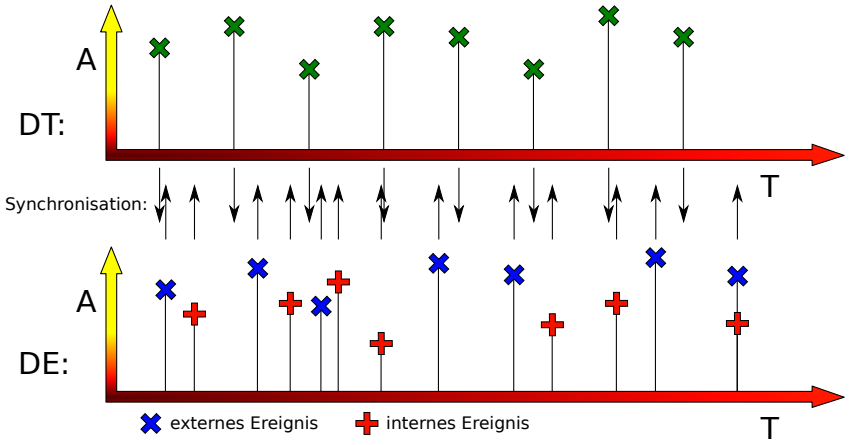



Abbildung 4.3: Zeitfunktionen und Ko-Simulation.

der HDL-Komponente erzeugt werden, welches in eine Simulink-Simulation eingebunden werden kann, ohne dass ein zusätzlicher Simulator benötigt wird. Der Mehraufwand, der bei einer echten Ko-Simulation auftritt, wird dadurch deutlich reduziert.

Die Eigenschaften ergeben sich aus den Vorbedingungen, dass es sich zum einen bei der HDL-Komponente um ein synthetisierbares HDL-Modell handelt, und dass zum anderen die Umgebung der Komponente in Simulink modelliert ist. Der erste Punkt, die Synthetisierbarkeit, führt dazu, dass die Implementierung der HDL-Komponente nicht den gesamten Umfang der HDL-Sprachelemente enthält, sondern nur Sprachelemente, die auch zur Synthese geeignet sind. Wie bereits in Abschnitt 2.3.4 beschrieben, bedeutet dies insbesondere, dass die Komponente keine Zeitannotationen enthalten darf, bzw. vorhandene Zeitannotationen ignoriert werden können, ohne die korrekte Funktion der Komponente zu beeinträchtigen. Die Tatsache, dass die Umgebung der Komponente in Simulink modelliert und simuliert wird, führt dazu, dass Wertänderungen an den Eingängen der Komponente und das Lesen der Komponentenausgänge nur zu bestimmten Zeitpunkten stattfindet, den Samplezeitpunkten in der Simulink-Simulation.

Diese beiden Gegebenheiten sind dafür verantwortlich, dass der logische Zeitpunkt jeder Aktivität in der HDL-Komponente mit den Samplezeitpunkten in der Simulink-Simulation übereinstimmt. Aktivitäten in der HDL-Komponente werden durch Ereignisse, also Wertänderungen, an ihren Eingängen ausgelöst. Diese Wertänderungen kommen aus Blöcken, die im Simulink-Modell vor der HDL-Komponente angeordnet sind. Sie können demnach nur zu den Zeitpunkten stattfinden, an denen die Ausgänge der Simulink-Blöcke aktualisiert werden, den Samplezeitpunkten (s. Abbildung 4.4). Ignoriert man in der HDL-Simulation die Zeitanno-

tationen, so sind alle Folgeereignisse, die aus einem Ereignis an einem der Komponenteneingänge resultieren, Deltaereignisse. Das bedeutet, dass nach einem Ereignis an einem der Komponenteneingänge eine Reihe von Deltazyklen folgen kann, aber kein *messbares* Voranschreiten der logischen Zeit. Abbildung 4.5 zeigt, dass sich das dynamische Verhalten der beiden Systeme stark ähnelt. Als Folge daraus kann der HDL-Simulator durch eine vereinfachte Variante ersetzt werden.

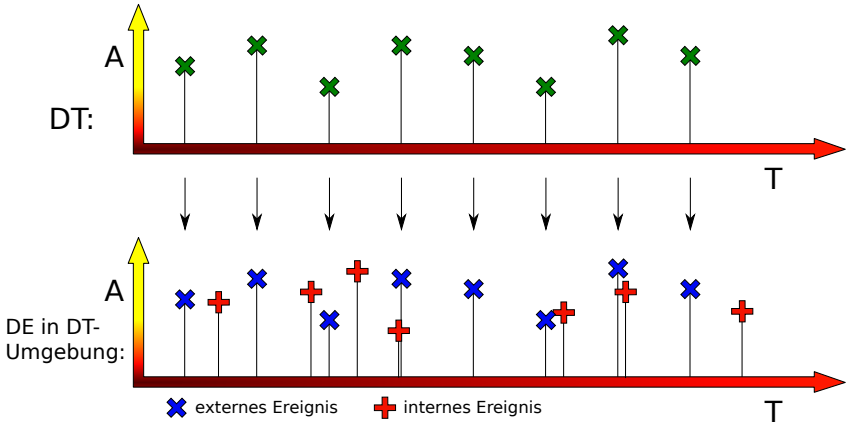



Abbildung 4.4: Zeitfunktion einer Discrete-Event-Komponente in einer Discrete-Time-Umgebung.

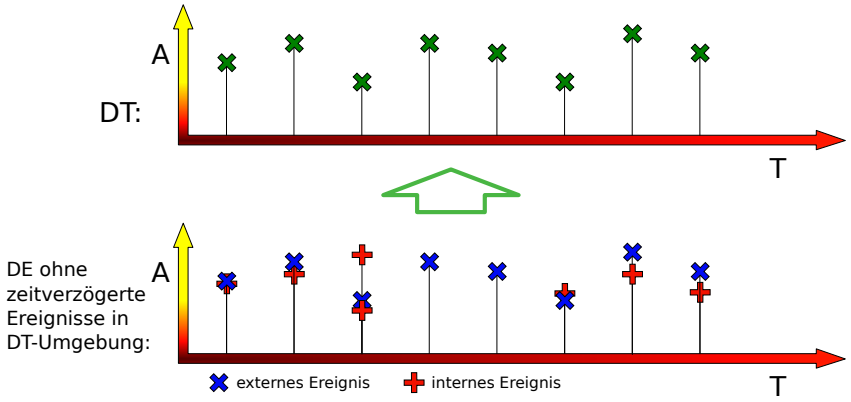



Abbildung 4.5: Zeitfunktion einer eingeschränkten Discrete-Event-Komponente in einer Discrete-Time-Umgebung.

Der vereinfachte Simulator benötigt keine eigene Zeitverwaltung und die Verwaltung der Ereignislisten ist erheblich einfacher. Deshalb kann das Modell der HDL-Komponente mittels Methoden aus der zyklusbasierten Hardwaresimulation automatisch in ein Simulationsmodell transformiert werden, in das die Aufgaben des Simulators direkt eingewoben sind. Ein separater Simulator ist deswegen nicht

mehr notwendig. Als Ergebnis entfällt der Aufwand für die Synchronisation der Simulatoren und der Aufwand für Kontextwechsel und das Wiederherstellen des HDL-Simulatorzustands kann deutlich reduziert werden.

Die Eigenschaft, dass es keinen eigenständigen Simulator für die Hardwarekomponente mehr gibt, ermöglicht dann eine weitere Optimierung. Bei der Simulation einer Hardwarekomponente in einer Systemumgebung treten häufig Phasen auf, in denen es keine Aktivität in der Hardwarekomponente gibt, zum Beispiel, weil die Komponente auf eine bestimmte Eingabe aus der Umgebung wartet. In der Systemsimulation ergeben sich aus diesen Phasen keine relevanten Erkenntnisse, ein gewöhnlicher HDL-Simulator muss sie dennoch vollständig simulieren. Der vereinfachte Simulator und insbesondere die Tatsache, dass es keine eigene Zeitverwaltung gibt, ermöglicht es, solche Phasen sehr effizient zu überspringen. So kann die Geschwindigkeit der Gesamtsimulation weiter erhöht werden.

## Wesentliche Beiträge der vorliegenden Arbeit

### Formalisierung der Simulationssemantik

Um das Verhalten von Simulink-Modellen und Hardwarekomponenten in der Simulation analysieren und vergleichen zu können, muss dieses Verhalten zunächst formal erfasst werden. Da existierende Formalisierungen von Simulink und Hardwarebeschreibungssprachen unter anderen Schwerpunkten entwickelt wurden, sind sie zu detailliert und für den hier erforderlichen Vergleich nicht geeignet. Deshalb wird eine neue, einfache Abbildung des abstrakten Simulations- und Zeitverhaltens von Simulink und HDLs auf die Formalismen zur Systemspezifikation *Discrete Time System Specification* (DTSS) und *Discrete Event System Specification* (DEVS) eingeführt.

### Abbildung des Anwendungsfalls auf die Formalismen

Der Anwendungsfall in dieser Arbeit ist die Simulation einer Hardwarekomponente, die in Form einer synthetisierbaren HDL-Beschreibung vorliegt, in einem Simulink-Umgebungsmodell. Die speziellen Eigenschaften und Anforderungen, die sich daraus ergeben, müssen erfasst und als Eigenschaften der formalen Modelle ausgedrückt werden. Daraus wird dann ein eingeschränktes DEVS-System definiert.

### Formale Transformation der Simulationssemantik

Anhand einer formalen Transformation des eingeschränkten DEVS-Systems wird gezeigt, dass dessen Verhalten dem eines DTSS-Systems entspricht. Folglich kann es im Kontext eines DTSS-Simulators simuliert werden.

### Erweiterung auf Takt-synchrone Hardwaremodelle

In der Praxis spielen Takt-synchrone Hardwarekomponenten eine große Rolle. Deshalb wird eine erweiterte Transformation für solche Hardwaremodelle entwickelt. Sie ermöglicht eine Simulation, bei der eine vorgegebene Taktrate unabhängig von der Schrittweite in Simulink realisiert werden kann.

### Konzept zur automatisierten Anwendung

Um die gestellten Anforderungen zu erfüllen, muss die in dieser Arbeit entwickelte Lösung automatisiert anwendbar sein. Deshalb wird die formal entwickelte Methode auf den eigentlichen Anwendungsfall übertragen, und es wird ein Konzept zur Automatisierung der notwendigen Transformationen beschrieben.

### Optimierung über stabile Zustände

Zur weiteren Steigerung der Simulationsgeschwindigkeit wird die Optimierung über stabile Zustände entwickelt, die durch die zuvor beschriebenen Transformationen ermöglicht wird. Neben der Beschreibung der Funktionsweise wird gezeigt, wie die dazu notwendigen Bedingungen formuliert und zu Garantien verknüpft werden können.

### Evaluation und Diskussion der entwickelten Methoden

Die entwickelten Methoden werden hinsichtlich des Simulationsverhaltens und der Simulationsgeschwindigkeit untersucht, und es wird evaluiert und diskutiert, ob sie die in Abschnitt 1.2 gestellten Anforderungen erfüllen.

# 5 Integration eines Discrete-Event-Modells in eine Discrete-Time-Simulation

Durch die Integration einer HDL-Komponente in eine Simulink-Umgebung ändert sich theoretisch betrachtet der experimentelle Rahmen für die Simulation der HDL-Komponente. Bestimmte Eigenschaften sind in der Simulation nicht mehr relevant, weil sie für den Beobachter, das umgebende Simulink-Modell, ohnehin nicht sichtbar sind. Die Auswirkungen dieses neuen experimentellen Rahmens werden nun in den bereits in Kapitel 2 vorgestellten formalen Modellen zur Systemspezifikation DEVS und DTSS analysiert. Dazu werden zum einen die Simulationssemantiken von VHDL als exemplarischem Vertreter für Hardwarebeschreibungssprachen und Simulink sowie zum anderen die besonderen Eigenschaften des vorliegenden Szenarios auf diese Formalismen abgebildet. Auf dieser Basis wird dann gezeigt, wie eine DEVS-Komponente in ein DTSS-System integriert werden kann.

## 5.1 Abbildung der Simulationsumgebungen auf die Formalen Modelle

Zunächst werden die Simulationsumgebungen in die Formalismen zur Systemspezifikation eingeordnet. Den Anfang macht Simulink gefolgt von VHDL.

### 5.1.1 DTSS und Simulink

Um den Bezug zum gegebenen Anwendungsszenario, der Simulation einer HDL-Komponente in einer Simulink-Umgebung, herzustellen, soll nun die Simulationssemantik von Simulink in den DTSS-Formalismus eingeordnet werden. In dieser Anwendung ist nur ein Teil von Simulink relevant. Die HDL-Komponente soll in Form eines S-Function-Blocks eingebunden werden. Da es sich bei der HDL-Komponente um ein digitales System handelt, enthält es außerdem nur diskrete

und keine kontinuierlichen Zustände. Es muss folglich lediglich die Simulationssemantik eines diskreten S-Function-Blocks betrachtet werden.

An dieser Stelle beschränkt sich die Betrachtung der Semantik zunächst auf die Simulation mit fester Schrittweite. Die Simulation mit variabler Schrittweite wird später in Abschnitt 5.3 behandelt.

Die Simulation mit fester Schrittweite entspricht exakt dem im DTSS-Formalismus beschriebenen Verhalten. Wie in Abschnitt 2.4.1 beschrieben, findet in jedem Simulationsschritt die Aktualisierung von Zustand und Ausgängen genau einmal statt, und bei der Simulation mit fester Schrittweite ist der zeitliche Abstand zwischen zwei aufeinanderfolgenden Aktualisierungsvorgängen immer gleich. Die Schrittweite in Simulink entspricht folglich der Konstante $c$. Die Callback-Funktionen `mdlOutputs()` und `mdlUpdate()` entsprechen der Ausgangsfunktion $\lambda$ und der Zustandsübergangsfunktion $\delta$ im DTSS-Formalismus.

Somit kann ein diskreter S-Funktion-Block in Simulink bei der Verwendung einer festen Schrittweite als DTSS-System spezifiziert werden.

### 5.1.2 DEVS und VHDL

Die Simulationssemantik von VHDL wurde bereits in Abschnitt 2.3.2 ausführlich beschrieben. Nun soll diese Semantik auf den DEVS-Formalismus abgebildet werden. Es gilt somit für die Bestandteile eines VHDL-Modells die jeweiligen Entsprechungen in DEVS zu finden. Da es sich bei VHDL um Discrete-Event-Simulation handelt, entspricht das dynamische Verhalten eines VHDL-Modells auch dem eines dynamischen DEVS-Systems. Unserem Anwendungsfall entsprechend betrachten wir kein vollständig geschlossenes VHDL-Modell, sondern eine VHDL-Komponente, also ein VHDL-Modell, das Ein- und Ausgangsports zu seiner Umgebung hat.

Widmen wir uns zunächst der Frage nach der Entstehung der unterschiedlichen Arten von Ereignissen. In VHDL entstehen Ereignisse durch bestimmte Wait-Anweisungen oder durch Wertänderungen auf Signalen und Eingangsports. Erstere sind die Ereignisse, die durch Wait-Anweisungen zum Warten für einen bestimmten Zeitraum[1] erzeugt werden. Da diese Anweisungen im Inneren der Komponente stehen und Teil der Verhaltensbeschreibung der Komponente selbst sind, müssen die entsprechenden Ereignisse den internen Ereignissen zugeordnet werden. Ebenfalls zu den internen Ereignissen zählen die Ereignisse, die durch Wertänderungen auf Signalen im Inneren der Komponente entstehen. Diese Ereignisse resultieren unmittelbar aus dem internen Verhalten der Komponente. Externe Ereignis-

[1] Im Folgenden auch mit Wait-for-Time-Anweisungen bezeichnet.

se sind diejenigen, die aus Wertänderungen an den Eingangsports entstehen. Ihre Entstehung steht in keinem direkten Zusammenhang mit dem internen Verhalten der Komponente. Insbesondere kann ihr Auftreten nicht auf Basis des Komponentenmodells vorhergesagt werden. Vielmehr werden sie aus der Umgebung in die Komponente eingebracht. Analog werden über die Ausgangsports Wertänderungen und Ereignisse von der Komponente an die Umgebung übermittelt. Damit ist die Abbildung der Mengen $X$ und $Y$ klar, sie entsprechen den Ein- und Ausgängen der VHDL-Komponente.

Das Verhalten, das in den Prozessen des VHDL-Modells implementiert ist, definiert sowohl die beiden Zustandsübergangsfunktionen $\delta_{ext}$ und $\delta_{int}$ als auch die Ausgangsfunktion $\lambda$. Eine Zustandsübergangsfunktion fasst alle Operationen zusammen, die aufgrund der einzelnen Ereignisse durchgeführt werden müssen. Dabei werden die Operationen, die durch externe Ereignisse ausgelöst werden, der externen Zustandsübergangsfunktion $\delta_{ext}$ zugeordnet. Operationen, die durch interne Ereignisse ausgelöst werden, sind $\delta_{int}$ zugeordnet. Im Quelltext eines VHDL-Modells gibt es oft keine vollständige Trennung dieser beiden Teile; die Ausführung einer einzelnen Anweisung kann unter Umständen sowohl durch ein externes als auch durch ein internes Ereignis ausgelöst werden. Logisch lässt sich die Ausführung einer Anweisung zu einem bestimmten Zeitpunkt der Simulation jedoch immer einem bestimmten Ereignis zuordnen, was für unsere Betrachtungen an dieser Stelle ausreichend ist. Operationen, die direkt zu Wertänderungen an Ausgängen führen, also Zuweisungen an Ausgangsports, werden der Ausgangsfunktion $\lambda$ zugeordnet.

Die Menge der Zustände $S$ ist in VHDL ebenfalls nicht explizit spezifiziert. Theoretisch ist die Menge der Zustände, die ein VHDL-Modell annehmen kann, eine Kombination aller Zustände, die seine Subelemente wie Prozesse, Signale und Variablen annehmen können. Zur formalen Betrachtung des Simulationsverhaltens sind jedoch nur die Zustände relevant, die ohne ein nächstes Ereignis nicht verlassen werden können. Gemeint sind damit die Zustände, die nach Abschluss aller Aktivitäten, die durch ein einzelnes Ereignis ausgelöst werden, erreicht sind. Zwischenzustände, die zum Beispiel eine Variable im Laufe dieser Aktivitäten annimmt, können aus der Zustandsmenge $S$ ausgenommen werden. Sie können vielmehr als Teil der Zustandsübergangsfunktionen betrachtet werden.

Bleibt noch die Zeitfortschrittsfunktion $ta$. Sie entspricht der zeitlichen Distanz bis zum nächsten internen Ereignis. Stellen wir uns dazu eine Ereignisliste vor, in der alle internen Ereignisse nach ihrem Zeitstempel geordnet eingetragen werden. Nach jedem Zustandsübergang kann der Wert von $ta(s)$ aus der Differenz aus dem Zeitstempel des ersten Elements in der Liste und dem aktuellen Zeitpunkt ermittelt werden. Ist die Liste leer, so ist $ta(s) = \infty$.

Zur Verdeutlichung des Zusammenhangs zwischen der Erzeugung von internen Ereignissen in VHDL und der Zeitfortschrittsfunktion stellen wir uns ein System vor, dass im Zeitpunkt $t_1$ in Zustand $s_1$ ist. $ta(s_1)$ ist gleich $\infty$, das heißt, es ist in diesem Moment kein zukünftiges internes Ereignis eingeplant. Aufgrund eines externen Ereignisses zum Zeitpunkt $t_2$ findet dann ein Zustandsübergang zum neuen Zustand $s_2$ statt. Im Laufe dieses Zustandsübergangs gibt es drei unterschiedliche Möglichkeiten zur Erzeugung neuer interner Ereignisse. Zum einen können im Zustandsübergang nicht-verzögerte Signalzuweisungen vorkommen. Ändert sich der Wert eines Signals, so wird ein neues Delta-Ereignis, ein internes Ereignis mit Zeitstempel des aktuellen Zeitpunkts $t_2$, zur Verarbeitung im nächsten Deltazyklus eingeplant. Die zweite Möglichkeit sind verzögerte Signalzuweisungen. Ebenso wie bei der dritten Möglichkeit, Wait-for-Time-Anweisungen, führt dies zu einem neuen internen Ereignis mit Zeitstempel $t_2+\tau$. Dabei ist $\tau$ die annotierte Verzögerungs-, bzw. Wartezeit mit $0 < \tau < \infty$ [2]. Nach dem Zustandsübergang gibt es folglich in der Liste der internen Ereignisse

1. ein erstes Ereignis mit Zeitstempel $t_2$,
2. ein erstes Ereignis mit Zeitstempel $t_2 + \tau$
3. oder kein Ereignis.

Folglich ergibt sich in Fall 1 für $ta(s_2)$ ein Wert von 0. In Fall 2 ist $ta(s_2) = \tau$ und in Fall 3 ist $ta(s_2) = \infty$.

Damit können wir nun beliebige VHDL-Komponenten und ihr dynamisches Verhalten im DEVS-Formalismus abbilden. Im folgenden Abschnitt werden auf dieser Basis die speziellen Eigenschaften für Modelle, die sich nach Kapitel 4 aus unserem Anwendungsfall ergeben, auf den DEVS-Formalismus übertragen.

## 5.2 Übergang von DEVS nach DTSS

Im vorangegangenen Abschnitt wurde gezeigt, wie die Simulationssemantiken von Simulink und VHDL auf die formalen Modelle DTSS und DEVS abgebildet werden können. Nachfolgend werden jetzt die wesentlichen Unterschiede zwischen den beiden formalen Modellen erläutert, bevor die Eigenschaften des HDL-Modells, die in Kapitel 4 beschrieben wurden, auf Eigenschaften des DEVS-Formalismus abgebildet werden. Schließlich wird gezeigt, wie sich diese Eigenschaften auf das dynamische Verhalten des DEVS-Systems auswirken und dass ein derart

[2] In der Theorie kann $\tau$ beliebige positive reelle Werte annehmen, in der Praxis gibt es jedoch Beschränkungen aufgrund der im Simulator verwendeten Zahlendarstellung.

eingeschränktes DEVS-System in seinem dynamischen Verhalten einem DTSS-System exakt gleicht.

### 5.2.1 Vergleich von DTSS und DEVS

Stellt man die Formalismen DTSS und DEVS nebeneinander, so fallen einige Gemeinsamkeiten, aber auch große Unterschiede auf. Gemeinsam ist beiden, dass es Mengen von Eingängen, Ausgängen und Zuständen gibt. Die wesentlichen Unterschiede finden sich in den Zustandsübergangsfunktionen und im dynamischen Verhalten.

Während die Zeitbasis eines dynamischen DTSS-Systems isomorph zu den natürlichen Zahlen ist, stellt die Menge der nicht-negativen reellen Zahlen inklusive unendlich die Zeitbasis eines DEVS-Systems dar. Dadurch unterscheiden sich die möglichen Zeitpunkte von Zustandsübergängen. In einem DTSS-System sind diese Zeitpunkte vorhersagbar. Gibt es einen Zustandsübergang bei Zeitpunkt $t_1$, so ist der nächste Zustandsübergang bei $t_1 + c$ und der vorhergegangene war bei $t_1 - c$. Demnach ist festgelegt, dass es in einem Zeitraum $[t_1, t_2)$ mit $t_2 - t_1 = c$ genau einen Zustandsübergang gibt, und die globale Zustandsübergangsfunktion des dynamischen Systems $\Delta$ kann wie in Gleichung 2.7 angegeben ausgedrückt werden.

Im Gegensatz dazu führt die reelle Zeitbasis eines DEVS-Systems dazu, dass die Zeitpunkte der Zustandsübergänge beliebig verteilt sein können. Befindet sich das System zu einem Zeitpunkt $t_1$ in Zustand $s \in S$, dann findet der nächste Zustandsübergang zum Zeitpunkt $t_1 + ta(s)$ oder wenn ein externes Ereignis eintritt statt. Da $ta(s)$ beliebige reelle Werte zwischen null und unendlich annehmen kann und die Zeitpunkte externer Ereignisse im Allgemeinen nicht vorhersagbar sind, können auch die Zeitpunkte von Zustandsübergängen nicht vorhergesagt werden. Ebenso ist die Anzahl von Ereignissen, und damit auch die Anzahl von Zustandsübergängen, in einem gegebenen Zeitintervall nicht festgelegt. In einem Intervall kann kein Ereignis stattfinden, während in einem anderen Intervall gleicher Größe viele Ereignisse stattfinden. Theoretisch können in einem Intervall auch unendlich viele Ereignisse stattfinden. In der Praxis ist das jedoch unerwünscht, da eine Simulation dieses Intervalls in endlicher Zeit keinen endgültigen Zustand erreichen könnte.

Damit das dynamische Verhalten eines DEVS-Systems dem eines DTSS-Systems gleicht, muss das Auftreten von Ereignissen eingeschränkt werden. Insbesondere müssen die Zeitpunkte von Ereignissen und Zustandsübergängen vorhersagbar sein.

### 5.2.2 Eingeschränktes DEVS

Wertänderungen an den Eingängen der HDL-Komponente entsprechen externen Ereignissen im DEVS-System. Diese treten nur zu bestimmten Zeitpunkten auf, den Samplezeitpunkten des Simulink-Modells. Im DTSS-Formalismus entsprechen diese Zeitpunkte der Zeitbasis $T_{DTSS} = c \bullet \mathbb{N}$. Ändern sich in einem Sampleschritt die Werte mehrerer Eingänge, so sind diese Änderungen und die daraus resultierenden Ereignisse echt gleichzeitig. Deshalb können solche Ereignisse ohne Beschränkung der Allgemeinheit als ein einzelnes Ereignis aufgefasst werden. Außerdem kann es passieren, dass sich in einem Zeitschritt der Wert keines der Eingänge ändern. Für diesen Fall kann ohne Beschränkung der Allgemeinheit ein spezielles, leeres Ereignis sowie eine Zustandsübergangsfunktion, die dieses Ereignis verarbeiten kann, angenommen werden. Somit kann folgende Einschränkung formuliert werden:

Seien $x_n, x_{n+1} \in X$ zwei aufeinanderfolgende externe Ereignisse und $t_{x_{n+1}}, t_{x_n} \in T_{DEVS}$ die Zeitpunkte ihres Auftretens, dann gilt:

$$\forall n \in \mathbb{N} : t_{x_{n+1}} - t_{x_n} = c. \tag{5.1}$$

Außerdem gibt es keine Zeitannotationen im Modell der HDL-Komponente. Das führt dazu, dass es im DEVS-System keine zeitverzögerten internen Ereignisse gibt, sondern lediglich Deltaereignisse. Alle Zustände des DEVS-Systems sind demnach entweder transitorische Zustände ($ta(s) = 0$) oder Zustände, die ohne ein externes Ereignis nicht verlassen werden können ($ta(s) = \infty$). Es gilt daher:

$$\forall s \in S : ta(s) \in \{0, \infty\}. \tag{5.2}$$

Aufgrund dieser Eigenschaften kann eine andere Formulierung der globalen Zustandsübergangsfunktion $\Delta$ gewählt werden.

#### Globale Zustandsübergangsfunktion $\Delta$

Gegeben sei ein Eingangssegment $\omega : [t_1, t_2) \longrightarrow X$ und ein initialer Zustand $q = (s, e_1)$. Dann können wieder drei Fälle unterschieden werden:

1. $t_1 = t_2$,
2. $t_1 = t_2 - c$,
3. alle anderen Möglichkeiten für $t_1$ und $t_2$.

**Fall 1:** Wenn $t_1 = t_2$ ist, so handelt es sich um ein leeres Segment, es gibt folglich auch kein Ereignis. Nach Gleichung 2.12 gilt:

$$\Delta(q, \omega[t_1, t_2)) = (s, e_1 + t_2 - t_1) = (s, e_1 + 0) = (s, e_1) = q. \tag{5.3}$$

**Fall 2:** Ist $t_1 = t_2 - c$, dann gibt es entsprechend obiger Eigenschaften genau ein externes Ereignis unmittelbar gefolgt von einer Reihe interner Ereignisse. Es wird somit entsprechend Gleichung 2.12 zunächst die externe Zustandsübergangsfunktion aufgerufen.

$$\Delta(q, \omega[t_1, t_2)) = \Delta\left(\left(\delta_{ext}\left((s, e_1 + t - t_1), \omega(t)\right), 0\right), \omega[t, t_2] \setminus \omega(t)\right) \tag{5.4}$$

Die Zeit, die zum Zeitpunkt des externen Ereignisses seit dem letzten Zustandsübergang vergangen ist, ist bekannt: $e_1 + t - t_1 = c$. Damit ist

$$\Delta(q, \omega[t_1, t_2)) = \Delta\left(\left(\delta_{ext}\left((s, c), \omega(t)\right), 0\right), \omega[t, t_2] \setminus \omega(t)\right). \tag{5.5}$$

Ist $ta\left(\delta_{ext}\left((s, c), \omega(t)\right)\right) = \infty$, so folgt nun kein internes Ereignis und der Zustand ändert sich nicht mehr. Andernfalls ist $ta\left(\delta_{ext}\left((s, c), \omega(t)\right)\right) = 0$, es folgt demnach unmittelbar und ohne Zeitverzögerung ein internes Ereignis. In dem Fall gilt:

$$\begin{aligned}
&\Delta(q, \omega[t_1, t_2)) \\
&= \Delta\left(\left(\delta_{int}\left(\delta_{ext}\left((s, c), \omega(t)\right)\right), 0\right), \omega\left[t + ta\left(\delta_{ext}\left((s, c), \omega(t)\right)\right) - 0, t_2\right) \setminus \omega(t)\right) \\
&= \Delta\left(\left(\delta_{int}\left(\delta_{ext}\left((s, c), \omega(t)\right)\right), 0\right), \omega[t + 0 - 0, t_2) \setminus \omega(t)\right) \\
&= \Delta\left(\left(\delta_{int}\left(\delta_{ext}\left((s, c), \omega(t)\right)\right), 0\right), \omega[t, t_2) \setminus \omega(t)\right)
\end{aligned} \tag{5.6}$$

Nun stellt sich erneut die Frage, ob $ta = \infty$ ist, dann gibt es keine weitere Zustandsänderung mehr. Ist $ta = 0$, dann gilt:

$$\begin{aligned}
\Delta(q, \omega[t_1, t_2)) = \Delta\Bigg(\Bigg(\delta_{int}\Big(\delta_{int}\Big(\delta_{ext}\left((s, c), \omega(t)\right)\Big)\Big), 0\Bigg), \\
\omega\left[t + ta\left(\delta_{int}\left(\delta_{ext}\left((s, c), \omega(t)\right)\right)\right) - 0, t_2\right) \setminus \omega(t)\Bigg) \\
= \Delta\left(\left(\delta_{int}\left(\delta_{int}\left(\delta_{ext}\left((s, c), \omega(t)\right)\right)\right), 0\right), \omega[t + 0 - 0, t_2) \setminus \omega(t)\right) \\
= \Delta\left(\left(\delta_{int}\left(\delta_{int}\left(\delta_{ext}\left((s, c), \omega(t)\right)\right)\right), 0\right), \omega[t, t_2) \setminus \omega(t)\right)
\end{aligned} \tag{5.7}$$

Die Reihe der internen Ereignisse setzt sich solange fort, bis ein Zustand $s'$ mit $ta(s') = \infty$ erreicht wird.

Wie bereits in Abschnitt 5.2.1 erläutert wurde, kann diese Reihe theoretisch unendlich lange fortgesetzt werden. In der Praxis ist dies jedoch nie der Fall, da für alle Zeiträume $[t_1, t_2)$ gilt, dass die Anzahl von Ereignissen in diesem Zeitraum endlich ist. Andernfalls kann eine Simulation dieses Zeitraums nicht in endlicher Zeit durchgeführt werden. Es gibt somit eine endliche Reihe von transitorischen Zuständen, an deren Ende ein Zustand mit $ta = \infty$ folgt. Die Definition eines transitorischen Zustands besagt, dass in diesem Zustand kein externes Ereignis auftreten kann. Außerdem ist der Wert von $e$ beim Verlassen eines solchen Zustands immer 0. Aus diesen Gründen kann die Reihe der Aufrufe der internen Zustandsübergangsfunktion an dieser Stelle ersetzt werden durch eine Funktion $\tilde{\delta}_{int} : S \longrightarrow S$, die in einer Schleife eine beliebige, endliche Reihe von Zustandsübergängen durchführt, solange bis ein nicht-transitorischer Zustand erreicht ist. Setzt man dies in die Gleichung für $\Delta$ ein, so erhält man

$$\Delta\left(q, \omega\left[t_1, t_2\right)\right) = \Delta\left(\left(\tilde{\delta}_{int}\left(\delta_{ext}\left(\left(s, c\right), \omega\left(t\right)\right)\right), 0\right), \omega\left[t + \infty - 0, t_2\right) \setminus \omega\left(t\right)\right) \tag{5.8}$$

Des Weiteren gilt, dass $t + \infty - 0$ immer größer ist als $t_2$, woraus folgt, dass $[t + \infty - 0, t_2)$ ein leeres Zeitintervall ist. Es gibt daher bis zum Zeitpunkt $t_2$ kein weiteres internes Ereignis und entsprechend der Voraussetzung in Gleichung 5.1 auch kein weiteres externes Ereignis. Um den endgültigen Zustand am Ende des Intervalls zu erhalten, muss lediglich noch der neue Wert für die vergangene Zeit seit dem letzten Zustandsübergang $e_2$ ermittelt werden. Zum Zeitpunkt $t$ ist der Wert für die vergangene Zeit $e_t = 0$, demnach ist:

$$e_2 = 0 + t_2 - t = t_2 - t. \tag{5.9}$$

An dieser Stelle kann auch der Wert für $t$ bestimmt werden. Wir wissen, dass es ein erstes externes Ereignis $x_0$ an Zeitpunkt $t_0$ gibt. $t_0$ ist dabei das erste Element der Zeitbasis des DTSS-Systems $T_{DTSS}$. Die Anzahl der externen Ereignisse, die vor einem bestimmten Zeitpunkt aufgetreten sind, ist:

$$n_{t)} = \begin{cases} 0 & \text{wenn } t \leq t_0 \\ \left\lceil \frac{t - t_0}{c} \right\rceil & \text{wenn } t > t_0 \end{cases} \tag{5.10}$$

Mit dieser Formel ist auch $n_{t_2)}$, die Anzahl der externen Ereignisse vor dem Zeitpunkt $t_2$, bestimmbar. Der Zeitpunkt des letzten Ereignisses vor Zeitpunkt $t_2$ ergibt sich so zu:

$$t = t_2 - e_2 = t_0 + n_{t_2)} \cdot c \tag{5.11}$$

Damit ist

$$\Delta\left(q, \omega\left[t_1, t_2\right)\right) = \left(\tilde{\delta}_{int}\left(\delta_{ext}\left(\left(s, c\right), \omega\left(t_0 + n_{t_2)} \cdot c\right)\right)\right), t_2 - t_0 + n_{t_2)} \cdot c\right) \tag{5.12}$$

**Fall 3:** Der dritte Fall soll alle verbleibenden Zeitintervalle abdecken:

$$t_1 \neq t_2 \wedge t_2 - t_1 \neq c. \tag{5.13}$$

Da bekannt ist, dass jegliche Aktivität im System zu periodischen Zeitpunkten mit Periode $c$ stattfindet, kann dieser Fall durch einen rekursiven Ausdruck vergleichbar mit dem DTSS-Formalismus auf die beiden zuvor beschriebenen Fälle zurückgeführt werden. Der neue Wert für $e_2$ kann ebenfalls über Gleichung 5.11 ermittelt werden.

$$\Delta\left(q, \omega\left[t_1, t_2\right)\right) = \left(\tilde{\delta}_{int}\left(\delta_{ext}\left(\Delta\left(q, \omega\left[t_1, t_2 - c\right)\right), \omega\left(t_0 + n_{t_2)} \cdot c\right)\right)\right), t_2 - t_0 + n_{t_2)} \cdot c\right) \tag{5.14}$$

Die drei Fälle Gleichung 5.3, Gleichung 5.12 und Gleichung 5.14 sind entsprechend zusammengefasst:

$$\begin{aligned}
&\Delta(q, \omega[t_1, t_2)) = \\
&(1) \quad q \\
&\qquad \text{wenn } t_1 = t_2, \\
&(2) \quad \left(\tilde{\delta}_{int}\left(\delta_{ext}\left(\left(s, c\right), \omega\left(t_0 + n_{t_2)} \cdot c\right)\right)\right), t_2 - t_0 + n_{t_2)} \cdot c\right) \\
&\qquad \text{wenn } t_1 = t_2 - c, \\
&(3) \quad \left(\tilde{\delta}_{int}\left(\delta_{ext}\left(\Delta\left(q, \omega\left[t_1, t_2 - c\right)\right), \omega\left(t_0 + n_{t_2)} \cdot c\right)\right)\right), t_2 - t_0 + n_{t_2)} \cdot c\right) \\
&\qquad \text{andernfalls.}
\end{aligned} \tag{5.15}$$

Abschließend können zwei weitere Vereinfachungen vorgenommen werden. Die Erste betrifft die vergangene Zeit seit dem letzten Zustandsübergang $e$ als Teil des Systemzustands. Ursprünglich ist das erste Argument der Abbildung $\delta_{ext}$ : $Q \times X \longrightarrow S$ aus $Q$, also ein Tupel $(s, e)$ bestehend aus einem Zustand $s$ und der vergangenen Zeit seit dem letzten Zustandsübergang $e$. Im vorliegenden Fall ist der Wert für $e$ jedoch bei jedem Aufruf von $\delta_{ext}$ gleich $c$. Deshalb muss $e$ nicht mehr als variabler Parameter in die externe Zustandsübergangsfunktion eingehen, und diese kann auf $\delta_{ext} : S \times X \longrightarrow S$ reduziert werden. Außerdem wird $e$ im vollständigen

DEVS-Formalismus bei der Bestimmung von internen Zustandsübergängen verwendet. Da für $ta(s)$ nach Gleichung 5.2 nur die Werte 0 oder $\infty$ annehmen kann, hat $e$ auch hier keine Bedeutung mehr. Im ersten Fall kann $e$ nie kleiner als $ta(s)$ sein, im zweiten Fall nie größer. Da $e$ folglich keinen Einfluss mehr auf das dynamische Verhalten des Systems hat, muss es auch nicht mehr als relevanter Bestandteil des Systemzustands betrachtet werden. Die globale Zustandsübergangsfunktion $\Delta$ des dynamischen Systems kann folglich auch auf eine Abbildung $\Delta : S \times \Omega \longrightarrow S$ reduziert werden.

Eine zweite Vereinfachung kann vorgenommen werden, weil die beiden Zustandsübergangsfunktionen immer in derselben Kombination aufgerufen werden. Sie können deshalb zusammengefasst werden.

$$\tilde{\delta}_{int}\left(\delta_{ext}\left((s,c),\omega\right)\right) = \delta_{ie}\left(s,\omega\right) \tag{5.16}$$

Somit ergibt sich schließlich für $\Delta$:

$$\begin{aligned}
&\Delta\left(s,\omega\left[t_1,t_2\right)\right) = \\
&(1)\quad s \\
&\qquad \text{wenn } t_1 = t_2 \\
&(2)\quad \delta_{ie}\left(s,\omega\left(t_0 + n_{t_2)}\cdot c\right)\right) \\
&\qquad \text{wenn } t_1 = t_2 - c \\
&(3)\quad \delta_{ie}\left(\Delta\left(s,\omega\left[t_1,t_2-c\right)\right),\omega\left(t_0 + n_{t_2)}\cdot c\right)\right) \\
&\qquad \text{andernfalls.}
\end{aligned} \tag{5.17}$$

Diese Definition entspricht exakt der Definition von $\Delta$ im DTSS-Formalismus in Gleichung 2.7.

### Zeitbasis

Als letzter Unterschied verbleibt noch die Zeitbasis. Im DEVS-Formalismus kann sie eine beliebige Teilmenge von $\mathbb{R}$ sein; im DTSS-Formalismus ist sie $c \bullet \mathbb{N}$, isomorph zu den natürlichen Zahlen. Die einzig relevanten Zeitpunkte im eingeschränkten DEVS-System sind die Zeitpunkte, an denen Aktivitäten, also Zustandsübergänge, stattfinden. Und diese Zeitpunkte entsprechen den Elementen der Zeitbasis des umgebenden DTSS-Systems. Dementsprechend kann auch die Zeitbasis des DEVS-Systems auf die DTSS-Zeitbasis eingeschränkt werden.

## 5.3 Simulation mit variablen Zeitschritten

Die Simulation mit variablen Zeitschritten in Simulink lässt sich im DTSS-Formalismus nicht vollständig beschreiben. Einige Eigenschaften von Discrete-Time-Systemen sind bei schrittweitenvariabler Simulation ebenfalls gegeben, wie zum Beispiel, dass die logische Zeit in diskreten Schritten voranschreitet. Andere jedoch nicht, insbesondere eben die konstante Größe der Zeitschritte. Dieser Fall kann besser als eine Sonderform von DEVS beschrieben werden.

Im Vergleich mit einem allgemeinen DEVS-System ist der wesentliche Unterschied, dass der Zeitpunkt des nächsten Zustandsübergangs aus dem aktuellen Zustand des Systems ermittelt werden kann. Die folgenden beiden Eigenschaften sind bei der Simulation in Simulink gegeben:

1. Das Simulink-Modell hat keine Eingänge, über die externe Ereignisse in die Simulation eingebracht werden.
2. Die Größe eines Zeitschritts ist ein diskreter Wert, der größer als null ist und kleiner als unendlich.

Eigenschaft 1 führt dazu, dass keine externe Zustandsübergangsfunktion benötigt wird. Damit ist auch die Rolle von $e$ – der vergangenen Zeit seit dem letzten Zustandsübergang – eine andere, da $e$ nicht als Parameter in die Zustandsübergangsfunktion eingeht. Zusammen mit Eigenschaft 2 ist außerdem die Zielmenge der Zeitfortschrittsfunktion *ta* eingeschränkt. Der Wert null würde hier einen Deltaschritt bedeuten, den es in Simulink nicht gibt. Der Wert unendlich hätte zur Folge, dass der aktuelle Zustand nur durch ein externes Ereignis verlassen werden kann. Die Simulation würde demnach in diesem Zustand stehen bleiben. Damit lässt sich ein entsprechend eingeschränktes Simulink-DEVS-System beschreiben als:

$$DEVS_{sim} : M_{sim} =< X_{sim}, S_{sim}, Y_{sim}, \delta_{int,sim}, \lambda_{sim}, ta_{sim} > \tag{5.18}$$

Dabei gilt:

- $X_{sim} = \left\{(p, v)|p \in InPorts, v \in X_{p,sim}\right\}$ ist die Menge der Eingangsports und ihrer Werte (hier leer).
- $Y_{sim} = \left\{(p, v)|p \in OutPorts, v \in Y_{p,sim}\right\}$ ist die Menge der Ausgangsports und ihrer Werte.
- $S_{sim}$ ist die Menge der Zustände.
- $\delta_{int,sim} : S_{sim} \longrightarrow S_{sim}$ ist die interne Zustandsübergangsfunktion.
- $\lambda_{sim} : S_{sim} \longrightarrow Y_{sim}$ ist die Ausgangsfunktion.
- $ta_{sim} : S_{sim} \longrightarrow \mathbb{R}^+$ ist die Zeitfortschrittsfunktion.

Die globale Zustandsübergangsfunktion eines dynamischen $\text{DEVS}_{sim}$-Systems kann dann wie folgt formuliert werden:

Sei $\omega : \langle t_1, t_2]$ ein Eingangssegment, $s$ der Zustand zu Zeitpunkt $t_1$ und $e$ die zu Zeitpunkt $t_1$ seit dem letzten Zustandsübergang vergangene Zeit. Dann ist:

$$\begin{aligned}
&\Delta\left((s,e),\omega\right) = \\
&(1)\quad (s, e + t_2 - t_1) && \text{wenn } e + t_2 - t_1 < ta_{sim}(s) \\
&(2)\quad \left(\delta_{int,sim}(s), 0\right) && \text{wenn } e + t_2 - t_1 = ta_{sim}(s) \\
&(3)\quad \Delta\left(\left(\delta_{int,sim}(s), 0\right), \omega\left(t_1 - e + ta_{sim}(s), t_2\right]\right) && \text{andernfalls}
\end{aligned} \tag{5.19}$$

Zustandsübergänge im Simulink-Teil des Gesamtmodells geschehen also durch Aufruf von $\delta_{int,sim}(s)$. Wie bereits in Abschnitt 5.2.2 beschrieben, sind diese Zustandsübergänge verantwortlich für das Auftreten externer Ereignisse in der HDL-Komponente.

In einem DTSS-System sind die Zeitpunkte der Zustandsübergänge stets statisch bestimmbar. Das ist hier nicht der Fall, aber zu jedem Zeitpunkt $t$ sind der Zeitpunkt $t_{last-ev}$ des jeweils letzten Zustandsübergangs vor $t$ und der Zeitpunkt $t_{next-ev}$ des nächsten Zustandsübergangs bekannt.

$$\begin{aligned}
t_{last-ev} &= t - e \\
t_{next-ev} &= t - e + ta_{sim}(s)
\end{aligned} \tag{5.20}$$

Mit diesen Eigenschaften können die Transformationen aus Gleichung 5.4 bis Gleichung 5.8 analog durchgeführt werden. Es kann folglich auch hier für die Simulation der DEVS-Komponente ein geschlossener Ausdruck formuliert werden, der einem Zeitschritt in der Simulink-Simulation entspricht.

## 5.4 Zusammenfassung

Die Analyse des gegebenen Anwendungsfalls hat ergeben, dass durch die Art der gegebenen Modelle und ihre Anordnung in der Simulation bestimmte Eigenschaften für diese Modelle in jedem Fall sichergestellt sind. Diese Eigenschaften wurden auf die Formalismen DTSS und DEVS abgebildet und so konnte gezeigt werden, dass das dynamische Verhalten eines DEVS-Systems, in dem diese Eigenschaften gelten, genau dem Verhalten des umgebenden DTSS-Systems entspricht.

Das führt in der Praxis dazu, dass für die Simulation einer DEVS-Komponente in einer DTSS-Umgebung kein vollwertiger Diskrete-Event-Simulator benötigt wird.

Das Zeitverhalten der eingeschränkten DEVS-Komponente erlaubt die Ausführung der Simulation direkt im Kontext des DTSS-Simulators. Somit ist der zusätzliche Aufwand, der bei der Verwendung von zwei vollwertigen Simulatoren für die Synchronisation und Kontextwechsel anfällt, nicht mehr erforderlich. Im Ergebnis ist deshalb eine schnellere Simulation des Gesamtsystems zu erwarten.

# 6 Takt-synchrone Hardwarekomponenten

In den bisherigen Betrachtungen wurden alle Eingänge einer Hardwarekomponente als Dateneingänge betrachtet. In der Praxis gibt es jedoch häufig einen Eingang mit besonderer Bedeutung: den Takteingang.

Ein großer Teil der digitalen Hardwarekomponenten arbeitet synchron zu einem Taktsignal. Dementsprechend sind auch die synthetisierbaren HDL-Modelle solcher Komponenten Takt-synchrone Modelle, sie haben also einen Takteingang. Die besondere Eigenschaft des Taktsignals ist, dass sein Wert mit einer vorgegebenen Frequenz zwischen 0 und 1 wechselt. Mit einem gegebenen Anfangswert können demnach sowohl der Wert des Taktsignals für jeden beliebigen Zeitpunkt als auch die Zeitpunkte der Wertänderungen berechnet werden. Das Verhalten des Taktsignals ist somit unabhängig vom Rest des Systems vorhersagbar.

In Simulink bringt das Taktsignal eine zusätzliche Beschränkung mit sich. Wird ein Taktsignal in der Simulink-Simulation erzeugt, so muss die Samplerate das Zweifache der gewünschten Taktrate (oder ganzzahlige vielfache davon) betragen, damit beide Taktflanken zum richtigen Zeitpunkt stattfinden können.

## 6.1 Skalierung der Taktfrequenz

Die Abhängigkeit der Samplerate in Simulink von der Taktrate der Hardwarekomponente ist nicht immer erwünscht. In einem Simulink-Modell zur Verarbeitung von Audiosignalen wird zum Beispiel die Samplerate oft entsprechend der Abtastrate des Audiosignals gewählt, die im kHz-Bereich liegt. Wird nun eine Hardwarekomponente, etwa ein DSP, eingebunden um einen Teil der Signalverarbeitung zu übernehmen, so ist dessen Taktfrequenz in der Regel deutlich höher, nämlich im MHz-Bereich. Eine Erhöhung der Samplerate entsprechend der Taktrate ist in einem solchen Fall ungünstig, da die Simulationsgeschwindigkeit dadurch geringer wird. Darüber hinaus kann eine Änderung der Samplerate auch zu Änderungen im Verhalten des Simulink-Modells führen. Um dies zu verhindern, kann ein Skalierungsfaktor zwischen der Samplerate in Simulink und der Taktrate der Hardware-

komponente verwendet werden. Das bedeutet, dass in einem Simulink-Zeitschritt mehrere Hardwaretakte ausgeführt werden.

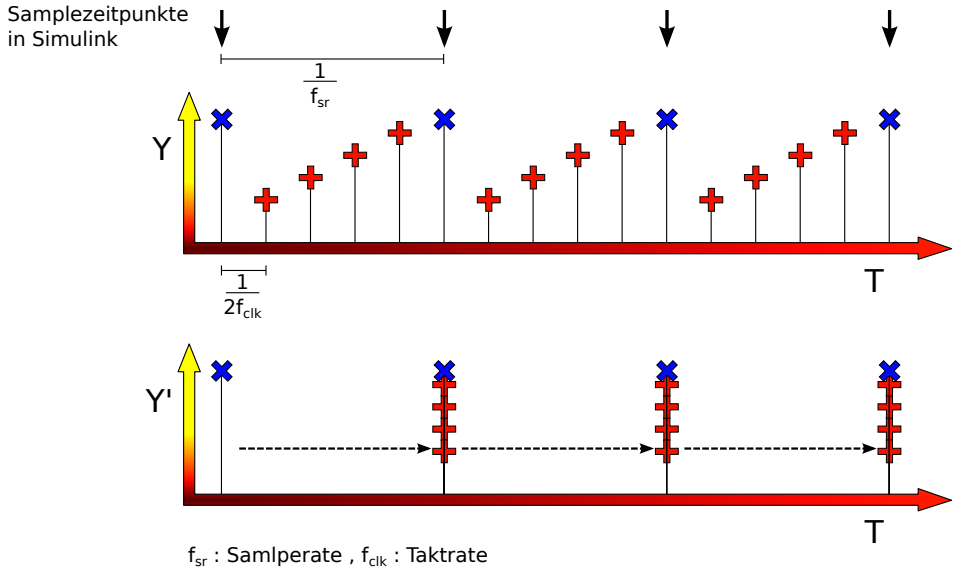



Abbildung 6.1: Ausgangstrajektorie mit Skalierung der Taktfrequenz.

In Abbildung 6.1 ist die Trajektorie eines Ausgangs der HDL-Komponente angedeutet. Der Wert am Ausgang ändert sich bei jeder Taktflanke. Das Simulink-Modell liest diesen Ausgang jedoch nur zu den Samplezeitpunkten. Deshalb kann die Verarbeitung der Taktflanken, die sich im Zeitintervall zwischen zwei Samplezeitpunkten befinden, an das Ende des Intervalls verschoben werden.

Im Folgenden wird zunächst davon ausgegangen, dass es sich bei einem Taktsignal um ein symmetrisches Taktsignal handelt. Das heißt, der zeitliche Abstand zwischen einer positiven Flanke und der folgenden negativen Flanke und der Abstand zwischen einer negativen Flanke und der folgenden positiven Flanke sind gleich. Eine Betrachtung asymmetrischer Taktsignale folgt später.

## 6.2 Formale Betrachtung der Frequenzskalierung

Ähnlich wie in Abschnitt 5.2.2 kann auch hier die Tatsache, dass es vorhersagbare Ereignisse mit gleichbleibendem zeitlichem Abstand gibt, ausgenutzt werden. Dadurch geht die vergangene Zeit seit dem letzten Zustandsübergang $e$ nicht mehr als variabler Parameter in die Zustandsübergangsfunktion ein. Ersetzt man den Parameter durch eine Konstante, so ist die reale Verweildauer für das Ergebnis der Funktion nicht relevant.

Damit die Eigenschaft der periodischen, vorhersagbaren Ereignisse gegeben ist, muss jedoch eine der folgenden beiden Bedingungen erfüllt sein. Entweder muss die Sampleschrittweite in Simulink ein ganzzahliges Vielfaches der Länge eines halben Taktzyklus der Hardwarekomponente[1] sein, oder es darf in der Hardwarekomponente keine Sensitivität auf externe Ereignisse außer dem Taktereignis geben. Im ersten Fall ist sichergestellt, dass alle externen Ereignisse mit einem Taktereignis zusammenfallen. Im zweiten Fall kann es zwar externe Ereignisse auch zu anderen Zeitpunkten geben, diese können aber ignoriert werden, wenn sichergestellt ist, dass sie keine Aktivitäten in der Hardwarekomponente auslösen können. asymmetrisch

Daher sind die in Abschnitt 5.2.2 vorgenommen Transformationen auch in diesem Fall gültig und die globale Zustandsübergangsfunktion aus Gleichung 5.17 kann als Ausgangspunkt dienen. Im Folgenden wird gezeigt, dass das Verhalten in diesem speziellen Fall über mehrere Taktschritte hinweg zusammengefasst werden kann.

Folgende Eigenschaften können für eine getaktete HDL-Komponente in einer Simulink-Umgebung angegeben werden:

- $f_s$ ist die Samplerate und $d = \frac{1}{f_s}$ die Schrittweite in Simulink,
- $f_{clk}$ ist die Taktrate und $c = \frac{1}{2 \cdot f_{clk}}$ die Schrittweite der Taktereignisse,
- $x_c$ und $x_{\bar{c}}$ sind die Taktereignisse,
  - das erste Taktereignis ist ein $x_c$-Ereignis zum Zeitpunkt $t_0$,
  - zu allen Zeitpunkten $t_0, t_0 + 2c, t_0 + 4c, ...$ tritt ein $x_c$-Ereignis auf,
  - zu allen Zeitpunkten $t_0 + c, t_0 + 3c, t_0 + 5c, ...$ tritt ein $x_{\bar{c}}$-Ereignis auf,

In diesem System soll nun der Zustandsübergang der HDL-Komponente über einen Zeitschritt in der Simulink-Simulation beschrieben werden. Seien also:

- $(t_1, t_2]$ ein Zeitschritt in der Simulink-Simulation mit
  - $t_2 - t_1 = d$ und
  - $t_2$ ist ein Samplezeitpunkt, also: $t_2 = t_0 + u \cdot d, u \in \mathbb{N}$,
- $\omega\,(t_1, t_2]$ ein Eingangssegment,
- $s_1 \in S$ der Zustand zum Zeitpunkt $t_1$ und
- $s_2$ der finale Zustand zum Zeitpunkt $t_2$.

[1] Die Länge eines halben Taktzyklus entspricht dem zeitlichen Abstand zwischen zwei aufeinanderfolgenden Taktereignissen (bei symmetrischem Takt).

### 6.2.1 Sampleschrittweite ist ganzzahliges Vielfaches der Schrittweite der Taktereignisse

Betrachten wir zunächst den ersten Fall: Die Sampleschrittweite in Simulink ist ein ganzzahliges Vielfaches des zeitlichen Abstands zwischen zwei aufeinanderfolgenden Taktereignissen. Demnach gilt:

$$d = v \cdot c, v \in \mathbb{N}^+ \tag{6.1}$$

Da die Zustandsübergangsfunktion in Gleichung 5.17 für rechts offene Intervalle $[t_a, t_b)$ definiert ist, muss der angegebene Zeitschritt zunächst auf ein solches Intervall abgebildet werden.

Aus $t_2 = u \cdot d \wedge t_1 = t_2 - d \wedge d = v \cdot c$ folgt, dass es jeweils bei $t_1$ und $t_2$ ein Taktereignis als Teil des Eingangsereignisses $\omega(t_1)$, bzw. $\omega(t_2)$, auftritt. Im Zeitraum dazwischen treten die Ereignisse $x_c$ und $x_{\bar{c}}$ abwechselnd jeweils einzeln auf. Der zeitliche Abstand zwischen zwei Ereignissen beträgt $c$. Demnach gibt es bei $t_1 + c$ das erste Ereignis nach $t_1$ und nach $t_2$ kein Ereignis vor $t_2 + c$. Damit gilt:

$$\begin{aligned} &\omega(t_1, t_2] \Leftrightarrow \omega[t_1 + c, t_2 + c) \\ &\Rightarrow \Delta(s_1, \omega(t_1, t_2]) = \Delta(s_1, \omega[t_1 + c, t_2 + c)) \end{aligned} \tag{6.2}$$

Zur besseren Anschaulichkeit wird das Zeitintervall aufgeteilt in den Bereich vor $t_2$ und den Bereich ab $t_2$. Sei dabei $s_2' \in S$ der Zustand unmittelbar vor $t_2$:

$$\begin{aligned} s_2' &= \Delta(s_1, \omega[t_1 + c, t_2)) \\ s_2 &= \Delta\left(s_2', \omega[t_2, t_2 + c)\right) \end{aligned} \tag{6.3}$$

Zur Bestimmung von $s_2'$ muss nun zunächst die Anzahl der Taktereignisse in $[t_1 + c, t_2)$ bestimmt werden. Über Gleichung 5.10 kann die Anzahl von Ereignissen vor einem bestimmten Zeitpunkt bestimmt werden. Die gesuchte Anzahl $n_{[t_1+c,t_2)}$ ist die Differenz der Anzahl der Ereignisse vor $t_2$ und der Anzahl der Ereignisse vor $t_1 + c$:

$$n_{[t_1+c,t_2)} = n_{t_2)} - n_{t_1+c)} \tag{6.4}$$

An dieser Stelle können drei Fälle unterschieden werden:

1. $n_{[t_1+c,t_2)} = 0$,
2. $n_{[t_1+c,t_2)} \neq 0$ und gerade,
3. $n_{[t_1+c,t_2)}$ ungerade.

**Fall 1:** Ist $n_{[t_1+c,t_2)} = 0$, dann muss $t_1 + c = t_2$ sein, weil Gleichung 6.1 gilt und Taktereignisse mit Abstand $c$ auftreten. Es gibt demnach kein Ereignis und es ist nichts weiter zu tun.

$$s_2' = s_1 \tag{6.5}$$

**Fall 2:** Ist $n_{[t_1+c,t_2)} \neq 0$ und gerade, so muss zuerst bestimmt werden, ob das erste Ereignis $x_c$ oder $x_{\bar{c}}$ ist. Ist $n_{t_1+c)}$ gerade, so ist das erste Ereignis in $[t_1 + c, t_2)$ $x_c$, andernfalls $x_{\bar{c}}$. Im Folgenden kann ohne Beschränkung der Allgemeinheit angenommen werden, dass das erste Ereignis $x_c$ ist. Im umgekehrten Fall ist das Vorgehen gleich, es müssen lediglich alle $x_c$ und $x_{\bar{c}}$ getauscht werden.

Ist das erste Ereignis $x_c$, die Anzahl der Ereignisse gerade und das Auftreten der Ereignisse immer abwechselnd ein $x_c$ und ein $x_{\bar{c}}$, so ist das letzte Ereignis im Intervall ein $x_{\bar{c}}$-Ereignis. Da es entsprechend der Voraussetzungen ein Taktereignis bei $t_2$ gibt, findet das letzte Ereignis in $[t_1 + c, t_2)$ zum Zeitpunkt $t_2 - c$ statt. Es gibt daher ein $x_{\bar{c}}$-Ereignis bei $t_2 - c$. Davor gibt es ein $x_c$-Ereignis bei $t_2 - 2c$. Also ist:

$$\begin{aligned} s_2' &= \delta_{ie}\left(\Delta\left(s_1, \omega\left[(t_1 + c), t_2 - c)\right)\right), x_{\bar{c}}\right) \\ &= \delta_{ie}\left(\delta_{ie}\left(\Delta\left(s_1, \omega\left[(t_1 + c), t_2 - 2c)\right)\right), x_c\right), x_{\bar{c}}\right). \end{aligned} \tag{6.6}$$

Da $n_{[t_1+c,t_2)}$ gerade ist, lassen sich genau $\frac{n_{[t_1+c,t_2)}}{2}$ dieser Paare aus $x_c$ und $x_{\bar{c}}$ bilden. Diese paarweisen Aufrufe von $\delta_{ie}$ können dann zusammengefasst werden als:

$$\delta_{ie}\left(\delta_{ie}\left(\Delta\left(s_1, \omega\left[(t_1 + c), t_2 - 2c)\right)\right), x_c\right), x_{\bar{c}}\right) = \delta_{clk}\left(\Delta\left(s_1, \omega\left[(t_1 + c), t_2 - 2c)\right)\right)\right) \tag{6.7}$$

Damit kann die Rekursion bis zum Beginn des Intervalls fortgesetzt werden und die sich wiederholenden Aufrufe können als Verkettung dargestellt werden.

$$\begin{aligned} s_2' &= \delta_{clk}\left(\Delta\left(s_1, \omega\left[(t_1 + c), t_2 - 2c)\right)\right)\right) \\ &= (\delta_{clk} \circ \delta_{clk})\left(\Delta\left(s_1, \omega\left[(t_1 + c), t_2 - 4c)\right)\right)\right) \\ &= (\delta_{clk} \circ \delta_{clk} \circ \delta_{clk})\left(\Delta\left(s_1, \omega\left[(t_1 + c), t_2 - 6c)\right)\right)\right) \\ &\vdots \\ &= \left(\delta_{clk}^{n_{[t_1+c,t_2)}/2}\right)(s_1) \end{aligned} \tag{6.8}$$

**Fall 3:** Ist $n_{[t_1+c,t_2)}$ ungerade, so gibt es in $[t_{1c}, t_2 - c)$ eine gerade Anzahl an Taktereignissen. Dieser Teil kann daher auf einen der vorherigen beiden Fälle zurückgeführt werden. Es verbleibt dann lediglich ein einzelnes Taktereignis am Ende des Intervalls. Unter der Annahme, dass das erste Ereignis im gesamten Intervall $x_c$ war, ist das zuletzt verbleibende Ereignis wieder $x_c$. So gilt in diesem Fall:

$$\begin{aligned} s_2' &= \delta_{ie}\left(\Delta\left(s_1, \omega\left[t_1 + c, t_2 - c\right)\right), x_c\right) \\ &= \delta_{ie}\left(\left(\delta_{clk}^{n_{[t_1+c,t_2-c)}/2}\right)(s_1), x_c\right) \end{aligned} \tag{6.9}$$

Als Ergebnis steht eine Berechnungsvorschrift zur Bestimmung von $s_2'$ zur Verfügung. Nun verbleibt nur noch die Verarbeitung des Zeitraums $[t_2, t_2 + c)$ zur Bestimmung von $s_2$.

$$\begin{aligned} s_2 &= \Delta\left(s_2', \omega\left[t_2, t_2 + c\right)\right) \\ &= \delta_{ie}\left(s_2', \omega\left(t_2 + c - c\right)\right) \\ &= \delta_{ie}\left(s_2', \omega\left(t_2\right)\right) \end{aligned} \tag{6.10}$$

Somit kann der Zustandsübergang von $t_1$ nach $t_2$ bestimmt werden. Dabei haben die tatsächlichen Zeitpunkte der dazwischenliegenden Zustandsübergänge und die Zeitdauer, die das System in den zwischenzeitlichen Zuständen verbringt, keinen Einfluss auf das Ergebnis.

### 6.2.2 Beliebige Sampleschrittweite

Um dieses Vorgehen auf beliebige Intervalle $(t_1, t_2]$ zu erweitern, muss zunächst eine Einschränkung gemacht werden: in der HDL-Komponente darf es keine nicht Takt-synchrone Sensitivität auf externe Ereignisse geben. Dadurch wäre die Vorhersagbarkeit der Zeitpunkte von Zustandsübergängen nicht mehr gegeben. Da sich die Werte der Eingänge zu beliebigen Zeitpunkten $t_1$ ändern können, entstehen auch die entsprechenden Ereignisse zu beliebigen Zeitpunkten. Gibt es im Modell jedoch keine Prozesse, die auf diese Ereignisse sensitiv sind, so können sie ignoriert werden.

Um dies in den formalen Modellen zu berücksichtigen, wird die Menge der Eingänge $X$ aufgeteilt in den Takteingang $X_{clk}$ und die übrigen Eingänge $X_{val}$. Folglich kann auch das Eingangssegment $\omega : (t_1, t_2] \longrightarrow X$ aufgeteilt werden in

$\omega_{clk} : (t_1, t_2] \longrightarrow X_{clk}$ und $\omega_{val} : (t_1, t_2] \longrightarrow X_{val}$, so dass

$$\begin{aligned} \omega(t) &= (\omega_{clk}(t), \omega_{val}(t)) \\ \Delta(s, \omega\langle t_a, t_b\rangle) &= \Delta\Big(s, (\omega_{clk}\langle t_a, t_b\rangle, \omega_{val}\langle t_a, t_b\rangle)\Big) \end{aligned} \tag{6.11}$$

$\omega_{clk}$ beinhaltet die Taktereignisse. $\omega_{val}$ beinhaltet die Werte aller anderen Eingänge, aber keine relevanten Ereignisse. Seien $s \in S$ ein initialer Zustand und $(\omega_{clk}\langle t_a, t_b\rangle, \omega_{val}\langle t_a, t_b\rangle)$ ein Eingangssegment, wobei $\omega_{clk}\langle t_a, t_b\rangle = \varnothing$, dann gilt:

$$\begin{aligned} \Delta(s, \omega\langle t_a, t_b\rangle) &= \Delta\Big(s, (\omega_{clk}\langle t_a, t_b\rangle, \omega_{val}\langle t_a, t_b\rangle)\Big) \\ &= \Delta\Big(s, (\varnothing, \omega_{val}\langle t_a, t_b\rangle)\Big) \\ &= s \end{aligned} \tag{6.12}$$

Die Werte der Eingänge in $X_{val}$ ändern sich darüber hinaus nur an den Samplezeitpunkten, also gilt:

$$\forall t \in [t_1, t_2) : \omega_{val}(t) = \omega_{val}(t_1) \tag{6.13}$$

Nun muss zunächst das Zeitintervall $(t_1, t_2]$ auf die Form $[t_a, t_b)$ gebracht werden. Dazu muss der Zeitpunkt $t_{1c}$ des ersten Taktereignisses nach $t_1$, sowie der Zeitpunkt $t_{2c}$ des ersten Taktereignisses nach $t_2$, bestimmt werden. Die Anzahl der Taktereignisse vor $t_1$ kann analog zu Gleichung 5.10 ermittelt werden. Wenn es darüber hinaus ein Taktereignis bei $t_1$ gibt, so ist $t_1 - t_0$ ganzzahlig durch $c$ teilbar.

$$n_{t_1]} = \begin{cases} n_{t_1)} + 1 & \text{wenn } (t_1 - t_0)\, mod\, c = 0 \\ n_{t_1)} & \text{andernfalls.} \end{cases} \tag{6.14}$$

Daraus ergibt sich für den Zeitpunkt des nächsten Taktereignisses:

$$t_{1c} = \Big(n_{t_1]} + 1\Big) \cdot c \tag{6.15}$$

Die Bestimmung von $t_{2c}$ ist analog dazu. Damit kann das Intervall $(t_1, t_2]$ aufgeteilt werden in $(t_1, t_{1c})$, $[t_{1c}, t_2)$ und $[t_2, t_2]$. Da außerdem bekannt ist, dass es in $(t_2, t_{2c})$ keine Taktereignisse gibt, gilt:

$$\omega_{clk}[t_2, t_2] \Leftrightarrow \omega_{clk}[t_2, t_{2c}) \tag{6.16}$$

Ebenso ist bekannt, dass es in $(t_1, t_{1c})$ keine Taktereignisse gibt:

$$\begin{aligned} \Delta(s1, \omega(t_1, t_{1c})) &= \Delta\Big(s1, (\omega_{clk}(t_1, t_{1c}), \omega_{val}(t_1, t_{1c}))\Big) \\ &= \Delta\Big(s1, (\varnothing, \omega_{val}(t_1, t_{1c}))\Big) \\ &= s_1 \end{aligned} \tag{6.17}$$

Demnach ist

$$\begin{aligned} s_2 &= \Delta\left(s_1, \omega\left(t_1, t_2\right]\right) \\ &= \Delta\left(s_2', \omega\left[t_2, t_{2c}\right)\right) \\ &= \Delta\left(\Delta\left(s_1, \omega\left[t_{1c}, t_2\right)\right), \omega\left[t_2, t_{2c}\right)\right) \end{aligned} \tag{6.18}$$

Im Weiteren ist das Vorgehen analog zu dem im vorhergehenden Abschnitt. Zunächst wird der Zustand $s_2'$ unmittelbar vor $t_2$ bestimmt. Die Anzahl $n_{[t_{1c},t_2)}$ der Taktereignisse im Zeitintervall $[t_{1c}, t_2)$ kann wie in Gleichung 6.4 ermittelt werden, und es werden die gleichen drei Fälle unterschieden.

**Fall 1:** Ist $n_{[t_{1c},t_2)} = 0$, dann gibt es kein Taktereignis und es ist nichts weiter zu tun.

$$\begin{aligned} s_2' &= \Delta\left(s_1, \left(\varnothing, \omega_{val}\left(t_1\right)\right)\right) \\ &= s_1 \end{aligned} \tag{6.19}$$

**Fall 2:** Ist $n_{[t_{1c},t_2)} \neq 0$ und gerade, so können auch hier Paare von Taktereignissen gebildet werden, so dass

$$\begin{aligned} s_2' &= \delta_{ie}\left(\delta_{ie}\left(\Delta\left(s_1, \left(\omega_{clk}\left[t_{1c}, t_2 - 2c\right), \omega_{val}\left(t_1\right)\right)\right), \left(x_c, \omega_{val}\left(t_1\right)\right)\right), \left(x_{\bar{c}}, \omega_{val}\left(t_1\right)\right)\right) \\ &= \delta_{clk}\left(\Delta\left(s_1, \left(\omega_{clk}\left[t_{1c}, t_2 - 2c\right), \omega_{val}\left(t_1\right)\right)\right), \omega_{val}\left(t_1\right)\right) \\ &= \left(\delta_{clk}^{n_{[t_{1c},t_2)}/2}\right)\left(s_1, \omega_{val}\left(t_1\right)\right) \end{aligned} \tag{6.20}$$

**Fall 3:** Ist $n_{[t_{1c},t_2)}$ ungerade, so wird auch in diesem Fall das letzte Ereignis separat behandelt, und der Rest des Intervalls kann auf einen der beiden vorhergehenden Fälle zurückgeführt werden.

$$s_2' = \delta_{ie}\left(\Delta\left(s_1, \left(\omega_{clk}\left[t_{1c}, t_2 - c\right), \omega_{val}\left(t_1\right)\right)\right), \left(x_c, \omega_{val}\left(t_1\right)\right)\right) \tag{6.21}$$

**Zeitpunkt $t_2$:** Basierend auf der Berechnungsvorschrift für den Zustand $s'_2$ unmittelbar vor Zeitpunkt $t_2$, kann nun $s_2$ selbst bestimmt werden.

$$s_2 = \Delta\left(s'_2, \left(\omega_{clk}\left[t_2, t_{2c}\right), \omega_{val}\left[t_2, t_{2c}\right)\right)\right) \tag{6.22}$$

Dazu wird zunächst ermittelt, ob bei $t_2$ ein Taktereignis stattfindet, und wenn ja, welches.

Ist $t_2 - t_0$ ganzzahlig durch $c$ teilbar, dann gibt es ein Taktereignis. Ist $t_2 - t_0$ ebenfalls ganzzahlig durch $2c$ teilbar, so handelt es sich um ein $x_c$-Ereignis. Somit kann $s_2$ wie folgt ermittelt werden:

$$s_2 = \begin{cases} s'_2 & \text{wenn } (t_2 - t_0)\, mod\, c \neq 0 \\ \delta_{ie}\left(s'_2, (c_x, \omega_{val}(t_2))\right) & \text{wenn } (t_2 - t_0)\, mod\, 2c = 0 \\ \delta_{ie}\left(s'_2, (c_{\bar{x}}, \omega_{val}(t_2))\right) & \text{andernfalls} \end{cases} \tag{6.23}$$

Als Ergebnis kann mit Gleichung 6.19, Gleichung 6.20, Gleichung 6.21 und Gleichung 6.23 auch in diesem Fall eine Vorschrift angegeben werden, über die der Zustandsübergang in der HDL-Komponente, der einem beliebigen Simulink-Zeitschritt entspricht, ohne explizite Zwischenschritte berechnet werden kann.

### 6.2.3 Asymmetrisches Taktsignal

Die bisherigen Betrachtungen der Frequenzskalierung gingen von der Annahme aus, dass es sich um symmetrische Taktsignale handelt. In Simulationen auf Systemebene ist die Betrachtung asymmetrischer Taktsignale in der Regel nicht notwendig, da sich dadurch nur eine kleine Veränderung des Zeitverhaltens innerhalb eines Taktzyklus ergibt. Die wesentlichen nach außen sichtbaren Aktivitäten finden jedoch bei vielen Hardwarekomponenten nur zu einer Taktflanke statt, so dass eine explizite Simulation des asymmetrischen Takts keine zusätzlichen, auf Systemebene interessanten Ergebnisse zeigt. Dennoch wird an dieser Stelle erläutert, wie die beschriebene Vorgehensweise auch bei asymmetrischen Taktsignalen angewendet werden kann.

Der Tastgrad, das Verhältnis zwischen Pulsdauer und Periodendauer, bestimmt bei asymmetrischen Taktsignalen den exakten Zeitpunkt der Taktflanken. Ein Tastgrad von 50 % entspricht einem symmetrischen Takt, bei einem kleineren Wert verringert sich der Abstand zwischen einer positiven Flanke und der folgenden negativen Flanke und der Abstand zwischen der negativen Flanke und der folgenden positiven Flanke vergrößert sich. Die grundlegende Eigenschaft eines Taktsignals,

die Vorhersagbarkeit seines Verhaltens, bleibt jedoch erhalten. Sind Frequenz, Anfangszustand und Tastgrad bekannt, dann können die Werte des Taktsignals für jeden beliebigen Zeitpunkt und damit auch die Zeitpunkte von Wertänderungen, also Ereignissen, berechnet werden. Des Weiteren ist nach wie vor sichergestellt, dass in einem Intervall $[t_1, t_2)$ mit $t_2 - t_1 = 2c$ genau zwei Taktereignisse auftreten, ein $x_c$- und ein $x_{\bar{c}}$-Ereignis. Deshalb kann der grundlegende Ansatz, wie er zuvor beschrieben wurde, erhalten bleiben. Lediglich die Berechnung der Anzahl von Ereignissen bis zu einem bestimmten Zeitpunkt und die Berechnung der Zeitpunkte von Ereignissen muss leicht angepasst werden.

Seien nun

- $c_1$ der Abstand von einem $x_c$-Ereignis zum folgenden $x_{\bar{c}}$-Ereignis,
- $c_2$ der Abstand von einem $x_{\bar{c}}$-Ereignis zum folgenden $x_c$-Ereignis und
- $c_1 + c_2 = 2c$.

Der Abstand zwischen zwei aufeinanderfolgenden $x_c$-Ereignissen oder zwei aufeinanderfolgenden $x_{\bar{c}}$-Ereignissen ist immer $2c$. Teilt man die gesamte Reihe der Taktereignisse auf in die Reihe der $x_c$-Ereignisse und die Reihe der $x_{\bar{c}}$-Ereignisse, dann lässt sich die Anzahl von Taktereignissen in einem Zeitintervall mit den bekannten Formeln aus Gleichung 5.10 und Gleichung 6.4 bestimmen. Dazu wird die Anzahl von $x_c$- und ein $x_{\bar{c}}$-Ereignissen getrennt berechnet und dann addiert. Bei der Berechnung der Zeitpunkte von Taktereignissen muss berücksichtigt werden, dass der Abstand zwischen zwei Taktereignissen nicht $c$ ist, sondern $c_1$ oder $c_2$. Findet zum Zeitpunkt $t_1$ ein $x_c$-Ereignis statt, so ist das letzte Taktereignis davor ein $x_{\bar{c}}$-Ereignis zum Zeitpunkt $t_1 - c_2$ und das nächste Taktereignis danach ein $x_{\bar{c}}$-Ereignis zum Zeitpunkt $t_1 + c_1$. Ebenso gilt für ein $x_{\bar{c}}$-Ereignis zum Zeitpunkt $t_2$, dass das letzte Taktereignis davor ein $x_c$-Ereignis zum Zeitpunkt $t_2 - c_1$ ist und das nächste danach ein $x_c$-Ereignis zum Zeitpunkt $t_2 + c_2$. Mit Berücksichtigung dieser Änderungen kann analog zum Vorgehen in den vorherigen Abschnitten eine Berechnungsvorschrift für den zusammengefassten Zustandsübergang zwischen zwei beliebigen Zeitpunkten entwickelt werden.

### 6.2.4 Mehrere Taktfrequenzen

Es gibt digitale Hardwarekomponenten, die über mehrere Takteingänge verfügen. Ein Grund dafür kann sein, dass zum Beispiel ein Prozessor zusammen mit zusätzlichen Schaltungsteilen in einer Komponente zusammengefasst wird. Die Taktfrequenz des Prozessors ist häufig vom Anbieter vorgeschrieben, während die anderen Teile für diese Frequenz nicht unbedingt geeignet sind. Auch wenn dies bisher nicht berücksichtigt wurde, so bleibt die Grundannahme, dass die Zeitpunkte von

Taktereignissen berechenbar sind, bestehen. In diesem Fall kann der größte gemeinsame Teiler der Schrittweiten der einzelnen Taktsignale als Grundschrittweite für Taktereignisse angenommen werden.

Seien

- $m$ die Anzahl an Taktsignalen,
- $c_1, c_2, ..., c_m$ die Schrittweiten der jeweiligen Taktsignale,
- $x_{c1}, x_{c2}, ..., x_{cm}$ und $x_{\bar{c}1}, x_{\bar{c}2}, ..., x_{\bar{c}m}$ die zugehörigen Taktereignisse und
- $c = GGT(c_1, c_2, ..., c_m)$ die Basisschrittweite der Taktereignisse.

Dann kann berechnet werden, dass alle $\frac{c_1}{c}$ Schritte immer abwechselnd ein $x_{c1}$- oder ein $x_{\bar{c}1}$-Ereignis auftritt. Außerdem tritt alle $\frac{c_2}{c}$ Schritte abwechselnd ein $x_{c2}$- oder ein $x_{\bar{c}2}$-Ereignis auf. Somit kann in jedem Schritt ermittelt werden, welche Taktereignisse verarbeitet werden müssen. In der Zustandsübergangsfunktion muss dann lediglich berücksichtigt werden, dass nicht bei jedem Aufruf genau ein Taktereignis stattfindet. Es können auch mehrere gleichzeitige Taktereignisse oder kein Taktereignis in einem Zeitschritt auftreten.

### 6.2.5 Variable Skalierung der Taktfrequenz

Im vorherigen Kapitel konnten variable Schrittweiten in der Simulink-Simulation vernachlässigt werden, da auch dann Ereignisse nur an Samplezeitpunkten auftreten können. Bei getakteten HDL-Komponenten muss jedoch auch die Skalierung zwischen Samplerate und Taktrate dynamisch gehalten werden. Bei variablen Schrittweiten ändert sich die Länge des Zeitschritts zwischen zwei Samplezeitpunkten. Da die Taktrate aber konstant bleiben soll, muss die Anzahl der Taktereignisse in einem Zeitschritt variabel und in Abhängigkeit von der Länge des Zeitschritts ermittelt werden.

Im vorangegangenen Abschnitt wurde gezeigt, wie eine Skalierung zwischen Samplerate und Taktfrequenz mit einer beliebigen Schrittweite durchgeführt werden kann. Dasselbe Vorgehen kann auch zur Unterstützung von variablen Sampleraten verwendet werden. Es ist lediglich zu beachten, dass neben dem initialen Zustand und den Werten der Eingänge zum vorangegangenen Samplezeitpunkt auch der Wert der logischen Zeit zum vorangegangenen Samplezeitpunkt benötigt wird, um die Anzahl der Taktereignisse bestimmen zu können.

## 6.3 Zusammenfassung

In diesem Kapitel wurde das Konzept erweitert, um den besonderen Anforderungen bei der Verwendung von Takt-synchronen Hardwarekomponenten gerecht zu werden. Die Regelmäßigkeit und Vorhersagbarkeit der Taktereignisse erlaubt es, diese von der Simulink-Umgebung zu trennen und separat zu erzeugen. Dadurch ist es möglich, die Samplefrequenz des Simulink-Modells unabhängig von der Taktfrequenz der Hardwarekomponente zu wählen. Eine Einschränkung muss jedoch bei der Skalierung der Frequenzen beachtet werden. Wenn die Samplezeitpunkte nicht mit dem Zeitpunkt eines Taktereignisses zusammenfallen, dann muss sichergestellt werden, dass es in der Hardwarekomponente keine Sensitivität auf ein externes Ereignis außer den Taktereignissen gibt. Sonst kann ein korrektes Verhalten nicht garantiert werden.

Abgesehen von dieser Einschränkung können beliebige Faktoren zur Skalierung zwischen Samplefrequenz und Taktfrequenz verwendet werden. Neben festen Faktoren können auf diese Weise auch variable Samplefrequenzen und eine variable Skalierung zwischen den Frequenzen realisiert werden.

# 7 Integration eines VHDL Modells in eine Simulink Simulation

Nachdem in den vorigen Kapiteln anhand theoretischer Modelle gezeigt wurde, wie ein DEVS-System in einer DTSS-Umgebung ausgeführt werden kann, wird dieses Konzept nun auf den realen Anwendungsfall übertragen: die Integration eines HDL-Modells in eine Simulink-Simulation.

Die praktische Umsetzung des Konzepts erzeugt in drei Schritten aus einem HDL-Modell den Quelltext eines Simulink-S-Function-Block. Dieser kann dann mit dem Simulink-Compiler übersetzt und in einem Simulink-Modell instanziiert werden. Im ersten Schritt, der internen Ablaufplanung, werden die Prozesse im HDL-Modell und deren Sensitivitäten analysiert. Basierend auf dieser Analyse werden die einzelnen Prozesse neu angeordnet und mit zusätzlicher Funktionalität versehen, die die Prozesse entsprechend ihrer Aktivierung ausführt. Ergebnis dieser Modelltransformation ist eine einzelne Funktion, die die gesamte durch ein externes Ereignis ausgelöste Aktivität inklusive aller anschließenden internen Ereignisse in einem Aufruf ausführt. Im zweiten Schritt muss die generierte Funktion nach C++ übersetzt werden. Ist das ursprüngliche Modell in SystemC gegeben, kann dieser Schritt entfallen. Dadurch erhalten wir eine C++-Funktion, deren Aufruf dem Verhalten der HDL-Komponente in einem Zeitschritt entspricht. Schließlich wird diese Funktion in einen Wrapper integriert, der das S-Function-Interface implementiert. Der Wrapper liest in jedem Zeitschritt die Eingänge des Blocks, ruft die erzeugte C++-Funktion auf und schreibt deren Ergebnisse auf die Ausgänge des Blocks.

## 7.1 Interne Ablaufplanung

In Abschnitt 5.2.2 wurde gezeigt, wie der Zustandsübergang einer synthetisierbaren HDL-Komponente, der einem Zeitschritt in Simulink entspricht, in einem geschlossenen Ausdruck formuliert werden kann (Gleichung 5.16). Es wird zunächst das externe Ereignis verarbeitet, dann die daraus resultierenden Deltaereignisse. Im Folgenden wird eine interne Ablaufplanung beschrieben, die diesen Ablauf reali-

siert. Das Ergebnis ist ein transformiertes Modell der HDL-Komponente, das die Simulation einzelner Zeitschritte ermöglicht, ohne dass die Infrastruktur, die eigentlich der HDL-Simulator zur Verfügung stellt, erforderlich ist. Dazu können Techniken verwendet werden, die aus dem Bereich der statischen und Taktzyklus-basierten Simulation bekannt sind. Das Konzept ist unter anderem in [NMS+04] beschrieben. In der Taktzyklus-basierten Simulation ist das Ziel die Erzeugung eines Softwareobjekts, das das Verhalten einer Hardwarekomponente in Taktzyklusschritten wiedergeben kann. Dies ist unserem Ziel, der schrittweisen Simulation, sehr ähnlich.

Im Wesentlichen werden die Prozesse des Modells durch Funktionen ersetzt und diese Funktionen werden dann in der Reihenfolge aufgerufen, in der ein Simulator die Prozesse ausführen würde. Das Grundkonzept ist in Abbildung 7.1 skizziert und in [Gö] ist die Realisierung in Form eines Beispiels dargestellt. Die Analyse des Datenflusses ergibt, dass Prozess A auf Signal S1 schreibt und Prozess B auf S1 sensitiv ist und auf Signal S2 schreibt. Prozess C ist wiederum auf Signal S2 sensitiv. Bei der Ablaufplanung werden die Prozesse A, B und C durch Funktionen ersetzt und die Signale S1 und S2 durch Variablen. Der Ablauf beginnt mit der Ausführung von Funktion A. Ändert sich in deren Verlauf der Wert von Variable S1, so wird im Anschluss Funktion B ausgeführt. Ändert sich nun bei deren Ausführung der Wert von S2, so wird dann Funktion C ausgeführt.

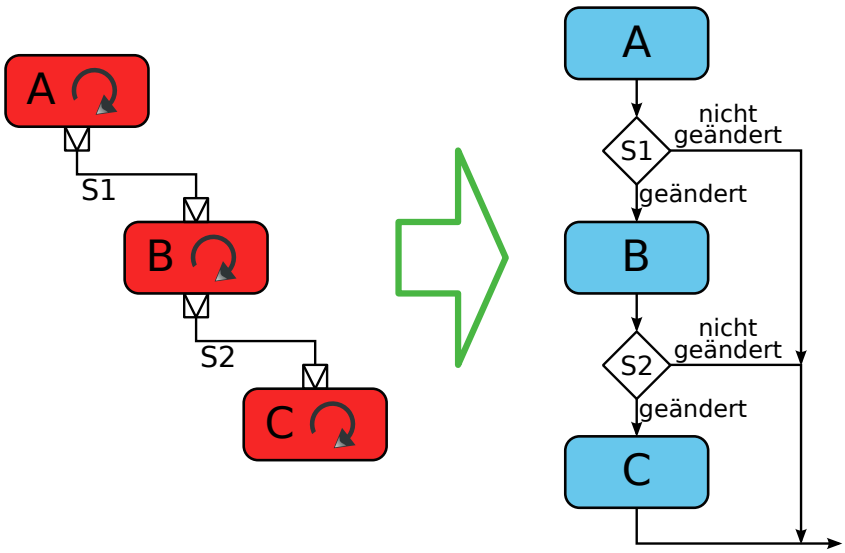



Abbildung 7.1: Konzept der internen Ablaufplanung.

Zunächst werden also die Sensitivitäten der einzelnen Prozesse und der Daten-

fluss im Modell analysiert[1]. Dann werden die Prozesse zu Funktionen umgebaut und alle Signale durch Variablen ersetzt. Dabei ist die Deltazyklussemantik und die Aktivitätssemantik der Signale zu berücksichtigen [VHD11b]. Wird in einem Deltazyklus sowohl schreibend als auch lesend zugegriffen, so muss beachtet werden, dass lesende Zugriffe noch den vorangegangenen Wert - vor dem Schreibzugriff - sehen. Außerdem wird nicht bei jedem Schreibzugriff ein Ereignis ausgelöst, sondern nur dann, wenn sich der Wert des Signals ändert. Deshalb werden in der Regel zwei Variablen benötigt, um ein Signal zu ersetzen. Die Prozessfunktionen werden dann dem Datenfluss im Modell entsprechend angeordnet. Begonnen wird mit den Funktionen, die als Prozess sensitiv auf Eingangssignale, also externe Ereignisse, waren. Aus dem zuvor analysierten Datenfluss lässt sich ersehen, welche Signale in diesen Prozessen beschrieben werden. Hat sich der Wert einer der Variablen, die als Ersatz für diese Signale eingeführt wurden, geändert, so werden die Funktionen aufgerufen, die jene Prozesse ersetzen, die auf das Signal sensitiv waren. Haben sich die Werte mehrerer Variablen geändert oder sind mehrere Prozesse auf dasselbe Ereignis sensitiv, so werden die entsprechenden Funktionen in einer bei der Transformation beliebig festgelegten Reihenfolge aufgerufen. Dies entspricht der Semantik der Discrete-Event-Simulation, da auch dort die Ausführungsreihenfolge der Prozesse innerhalb eines Deltazyklus nicht definiert ist. Die Datenflussanalyse kann nun erneut dazu genutzt werden, um die Liste der Signale zu ermitteln, die sich geändert haben können. Anschließend werden erneut die darauf sensitiven Prozesse gesucht und die entsprechenden Funktionsaufrufe angeordnet. Dieses Schema wird fortgesetzt bis die Liste der zuletzt geschriebenen Signale leer ist.

Gibt es zyklische Abhängigkeiten im Datenfluss, so kann eine leere Signalliste nie erreicht werden. Dann wird die weitere Ablaufplanung in Form einer Schleife realisiert, die endet, wenn sich in einem Durchgang kein relevanter Variablenwert mehr geändert hat. Dass diese Schleife terminiert, ist genau dann sichergestellt, wenn auch die zyklischen Aktivierungen in der ursprünglichen HDL-Simulation terminieren. Endlosschleifen können folglich nur dann auftreten, wenn sie auch im Originalmodell vorhanden waren.

Der Ablaufplan und die Aufrufe der Prozessfunktionen können so in einer einzelnen Funktion zusammengefasst werden. Diese Funktion bekommt die aktuellen Werte der Eingänge sowie die aktuellen externen Ereignisse als Eingangsparameter. Ein Aufruf der Funktion führt alle Aktivitäten aus, die in der HDL-Komponente aufgrund der externen Ereignisse ausgelöst würden. Neben den direkt angestoßenen Prozessen schließt das insbesondere auch alle daraus resultierenden internen Ereignisse ein. Die Ergebniswerte für die Ausgangssignale und die entsprechenden

[1] Nebenläufige Signalzuweisungen außerhalb von Prozessen (concurrent signal assignment) werden hier und in allen folgenden Schritten wie die dazu äquivalenten Prozesse behandelt.

Ausgangsereignisse werden schließlich über Ausgangsparameter an die Umgebung zurückgegeben.

In der Literatur sind diverse weitere Schritte zur Optimierung solcher Ablaufplanungen und zur Minimierung von Variablenzuweisungen und -vergleichen zu finden. So wird zum Beispiel in den Abschnitten 73 und 74 in [NMS+04] erläutert, wie durch die Umsortierung von Prozessen das Puffern von Signalwerten in bestimmten Fällen, und damit auch der Aufwand für das Kopieren von Variablen, entfallen kann. In [GN00] wird erläutert, wie bei Taktzyklus-basierter Simulation durch Umordnen von Prozessen deren mehrfache Evaluation in einem Taktschritt vermieden werden kann. Auf die weitere Betrachtung und Umsetzung solcher Optimierungsschritte wird an dieser Stelle verzichtet, weil die Implementierung einer effizienten Ablaufplanung nicht im Vordergrund dieser Arbeit steht. Da andere Werkzeuge, wie zum Beispiel [Car], Modelltransformationen mit vergleichbarem Ergebnis bieten, ist es jedoch ohne Weiteres möglich, die hier gewählte Transformation in zukünftigen Arbeiten zu ersetzen.

Das Ergebnis der Transformation ist eine einzelne, monolithische Funktion (im Folgenden auch *step-Funktion* genannt), die der in Gleichung 5.16 dargestellten Zustandsübergangsfunktion entspricht. Mit einem Aufruf dieser Funktion kann die vollständige Reaktion der HDL-Komponente auf ein einzelnes oder mehrere gleichzeitige externe Ereignisse simuliert werden. Damit kann die step-Funktion zur Simulation eines Zeitschritts verwendet werden. Prinzipiell entspricht die interne Ablaufplanung der Implementierung eines einfachen Discrete-Event-Simulators. Aber es werden kein eigener Zustand und keine Zeitverwaltung für den Simulator benötigt. Der Zustand des Systems beruht einzig auf den Werten der Variablen im Modell und die Verwaltung der Simulationszeit kann vollständig in die Umgebung ausgelagert werden.

## 7.2 Übersetzung nach C++

Das Ergebnis der zuvor beschriebenen Transformation enthält keine der Elemente mehr, die für HDLs spezifisch sind. Prozesse, explizite Parallelität und Signale wurden durch Funktionen, eine feste Ablaufplanung und Variablen ersetzt. Es bleibt lediglich die step-Funktion, eine Funktion im Sinne einer klassischen Programmiersprache. Deshalb ist es möglich, die HDL-Funktion in eine C++-Funktion zu übersetzen. Liegt das ursprüngliche Modell in SystemC vor, so ist diese Übersetzung nicht erforderlich, da es sich ja schon um C++ handelt. Bei VHDL oder Verilog stehen Werkzeuge zur automatischen Übersetzung zur Verfügung.

Zur Übersetzung einer VHDL-Funktion nach C++ wird in dieser Arbeit ein

Werkzeug zur automatischen Transformation von VHDL nach SystemC verwendet [GON12]. Es ist in der Lage synthetisierbare VHDL-Modelle in funktional äquivalente SystemC-Modelle zu transformieren. Da dieses Werkzeug ein eigenes C++-Datentypsystem mitbringt, das den Datentypen in VHDL entspricht, und das zu übersetzende Modell keine Prozesse oder Signale enthält, gibt es im Übersetzungsergebnis auch keine Abhängigkeiten zu SystemC. Man erhält als Ergebnis eine reine C++-Funktion, die äquivalent zu der zuvor erzeugten VHDL-Funktion ist.

In dieser Arbeit werden lediglich Modelle in VHDL betrachtet, es sei jedoch erwähnt, dass ähnliche Werkzeuge auch für Verilog verfügbar sind [Ver].

## 7.3 Integration in Simulink

Bis jetzt haben wir eine C++-Funktion, die einen Zeitschritt der HDL-Komponente simulieren kann. Diese Funktion soll nun in eine Simulink-S-Function integriert und damit zu einem Simulink-Block werden. Dazu wird ein Wrapper verwendet, der das S-Function-Interface implementiert und in jedem Zeitschritt die step-Funktion aufruft. Die Aufgaben dieses Wrappers und deren Ablauf sind in Abbildung 7.2 skizziert. In der Initialisierungsphase der Simulation kommen dazu lediglich noch die üblichen Schritte zur Initialisierung einer S-Function: das Erzeugen und Registrieren von Ein- und Ausgängen und deren Datentypen.

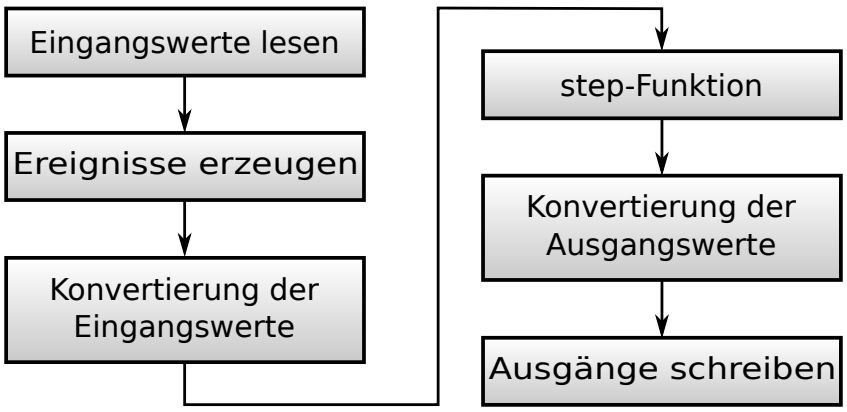



Abbildung 7.2: Aufgaben des S-Function-Wrappers.

Wie in Abschnitt 2.4.2 beschrieben bietet das S-Function-Interface eine Callback-Funktion namens `mdlOutputs()`, die in jedem Zeitschritt der Simulation einmal aufgerufen wird. In dieser Funktion wird der in Abbildung 7.2 dargestellte Ablauf implementiert.

Zunächst werden die Werte der Eingänge gelesen. Dann werden diese Werte mit den Werten aus dem letzten Zeitschritt verglichen. Für alle Eingänge deren Wert sich geändert hat, wird eine Markierung gesetzt, die einem externen Ereignis in der Discrete-Event-Simulation entspricht. Bevor die Eingangswerte an die Hardwarekomponente übergeben werden können, müssen diese noch in passende Datentypen konvertiert werden. Dies ist notwendig, da in Simulink in der Regel andere Datentypen eingesetzt werden als in HDL-Modellen. Dann wird die step-Funktion mit den Ereignismarkierungen und den Eingangswerten als Parameter aufgerufen. Nach dem Aufruf der Funktion enthalten die Ausgangsparameter die neuen Werte für die Ausgänge der Komponente sowie Markierungen für die Werte, die sich in diesem Aufruf geändert haben. Veränderte Ausgangswerte werden schließlich zu Simulink-Datentypen konvertiert und auf die entsprechenden Ausgänge des Simulink-Blocks geschrieben.

### 7.3.1 Konvertierung der Datentypen

Während in Simulink in der Regel abstrakte numerische Datentypen verwendet werden, werden in HDL-Modellen bitakkurate numerische oder bitorientierte logische Datentypen eingesetzt. Deshalb ist an der Grenze zwischen dem Simulink-Modell und der Hardwarekomponente eine geeignete Konvertierung der Datentypen notwendig. In dieser Arbeit wurde eine Abbildung der Datentypen verwendet, wie sie auch bestehenden Werkzeugen zur Ko-Simulation üblich ist. Wahrheitswerte in Simulink (boolean) werden auf Logik- oder Bitwerte in HDLs (std_logic, bit) abgebildet. Bei numerischen Daten werden diese zunächst in Simulink auf Fixkomma-Datentypen mit festgelegter Bitbreite übertragen. Diese können dann auf entsprechend breite Logik- oder Bitvektoren (std_logic_vector, bit_vector) abgebildet werden. In einigen Fällen ist es auch möglich, numerische Datentypen entsprechend ihres Wahrheitswerts ($x = 0 \rightarrow falsch$, $x \neq 0 \rightarrow wahr$) zu interpretieren. Außerdem ist es möglich Vektoren von Wahrheitswerten auf Logik- oder Bitvektoren abzubilden. Eine Abbildung von abstrakten numerischen Datentypen wie Fließkommazahlen anhand ihres numerischen Wertes könnte zukünftig ergänzt werden. Die Verwendung von Fließkommazahlen ist jedoch in synthetisierbaren Hardwarebeschreibungen nicht erlaubt und eine Abbildung auf Fixkomma-Datentypen ist sehr stark von der jeweiligen Anwendung abhängig. Deshalb ist es nicht ohne Weiteres eine allgemeingültige Abbildung für beliebige Systeme zu definieren.

### 7.3.2 Direct-Feedthrough

In der Simulation von HDL-Modellen führen aufeinanderfolgende Deltazyklen nicht zu einem messbaren Voranschreiten der logischen Zeit. In der Simulink-Simulation werden keine Deltazyklen, sondern nur diskrete Zeitpunkte unterschieden. Deshalb muss die gesamte step-Funktion in der Theorie einem einzigen logischen Zeitpunkt zugeordnet werden, und damit auch einem einzigen Zeitschritt in der Simulink-Simulation. Dieses Verhalten wird durch das in Abbildung 7.2 skizzierte Verhalten realisiert. Ändern sich die Eingangswerte, so geschehen daraus resultierende Änderungen der Ausgänge unmittelbar im selben Zeitschritt – zum selben logischen Zeitpunkt.

In Simulink wird ein solches Verhalten, bei dem die aktuellen Eingangswerte zur Berechnung der Ausgangswerte im selben Zeitschritt verwendet werden, als *Direct-Feedthrough* bezeichnet.

Auch wenn dieses Verhalten nach der Theorie korrekt ist, entspricht es nicht immer der Erwartungshaltung eines Entwicklers. Insbesondere ein Hardwareentwickler empfindet eine Reihe von Deltazyklen nicht unbedingt als *zum gleichen Zeitpunkt*. Er versteht eher nur den ersten Deltazyklus zum Zeitpunkt $t$ als wirklich zu Zeitpunkt $t$ gehörig. Alle daran anschließenden Deltazyklen finden *kurz nach* $t$ statt. In einigen Fällen ist es erwünscht, eine solche Zeitverzögerung auch in der Simulink-Simulation sichtbar zu machen. Im Sinne der Simulink-Simulation bedeutet das, dass Änderungen an Ausgängen der HDL-Komponente, die nicht im ersten Deltazyklus stattfinden, erst im nächsten Zeitschritt für nachgeschaltete Simulink-Blöcke sichtbar sind.

Dem ersten Deltazyklus kommt eine besondere Rolle zu. Die Deltazyklussemantik der Signale in HDLs sorgt jedoch dafür, dass im ersten Deltazyklus eines Zeitschritts keine Wertänderung eines Ausgangs auftreten kann. Ändert sich der Wert eines Eingangs, so kann die Wertänderung eines Ausgangs als Reaktion darauf frühestens im nächsten Deltazyklus am Ausgang beobachtet werden.

Dies lässt sich auch anhand des DEVS-Formalismus erklären. Dort ist festgelegt, dass die Ausgangsfunktion $\Lambda$ nur bei internen Ereignissen ausgelöst wird. In unserem Fall entstehen interne Ereignisse jedoch nur als Reaktion auf externe Ereignisse, also nicht echt gleichzeitig mit den externen Ereignissen.

Insgesamt führt das dazu, dass der Aufruf der step-Funktion vollständig in den nächsten Zeitschritt verschoben werden kann. Zu beachten ist lediglich, dass dabei die Eingangswerte aus dem vorherigen Zeitschritt verwendet werden. Da die Eingangswerte im S-Funktion-Wrapper in jedem Fall zwischengespeichert werden müssen, um Wertänderungen zu ermitteln und die entsprechenden Ereignis-Markierungen zu erzeugen, entsteht dadurch kein zusätzlicher Aufwand. Die Rei-

henfolge der Verarbeitungsschritte muss lediglich entsprechend Abbildung 7.3 geändert werden, um die Zeitverzögerung in Simulink sichtbar zu machen.

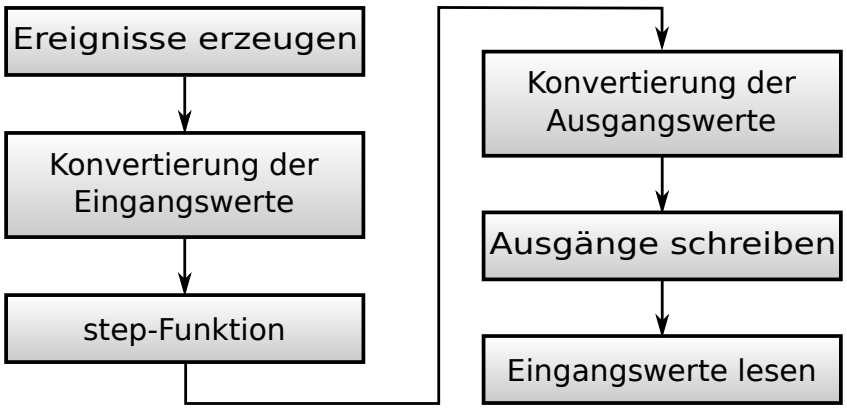



Abbildung 7.3: S-Function-Wrapper ohne Direct Feedthrough.

## 7.4 Erzeugung eines Taktsignals und Skalierung der Frequenz

In Kapitel 6 wurde bereits die besondere Rolle des Takteingangs bei Hardwarekomponenten geschildert und es wurde gezeigt, wie eine Skalierung zwischen der Samplerate in Simulink und der Taktrate der Hardwarekomponente realisiert werden kann. In der Praxis kann dieses Konzept sehr einfach durch eine Schleife im S-Funktion-Wrapper umgesetzt werden.

Zunächst muss dazu das Taktsignal vom Wrapper selbst erzeugt werden. Statt eines Takteingangs für den Simulink-Block, wird im S-Function-Wrapper eine boolesche Variable angelegt. Für jede Taktflanke muss dann der Wert dieser Variablen invertiert und die Ereignismarkierung für die Wertänderung erzeugt werden. Anstelle des Werts des Takteingangs wird der Wert dieser Variablen an die step-Funktion übergeben.

Wird das gleiche Ratenverhältnis wie bei der Taktgenerierung in Simulink verwendet, die Samplerate entspricht somit dem Doppelten der Taktrate, dann muss der Wert der Taktvariablen in jedem Zeitschritt einmal invertiert werden. Ebenso wird die step-Funktion in jedem Zeitschritt einmal aufgerufen, also zyklisch und abwechselnd mit den Werten 0 und 1 für den Takteingang. Dieses Verhalten entspricht exakt dem, welches bei einer Generierung des Taktsignals im umgebenden Simulink-Modell auftreten würde. Bei einem anderen Verhältnis zwischen Taktrate

und Samplerate muss die Anzahl und Art der virtuellen Taktereignisse wie in Abschnitt 6.2 dargestellt aus der Größe des Zeitschritts ermittelt werden. Dazu wird eine weitere Variable $t_{clk}$ benötigt, die einen Zeitpunkt speichern kann. Zu beachten ist, dass der Zeitpunkt des letzten Samplehits bei der Berechnung des Zustandsübergangs nur verwendet wird, um daraus die Zeitpunkte von Taktereignissen zu ermitteln. In der Praxis können direkt die Zeitpunkte von Taktereignissen verwendet werden. Die Art des Taktereignisses, positive oder negative Flanke, muss nicht explizit ermittelt werden, da in der Taktvariablen die Art des zuletzt verarbeiteten Taktereignisses bereits gespeichert ist.

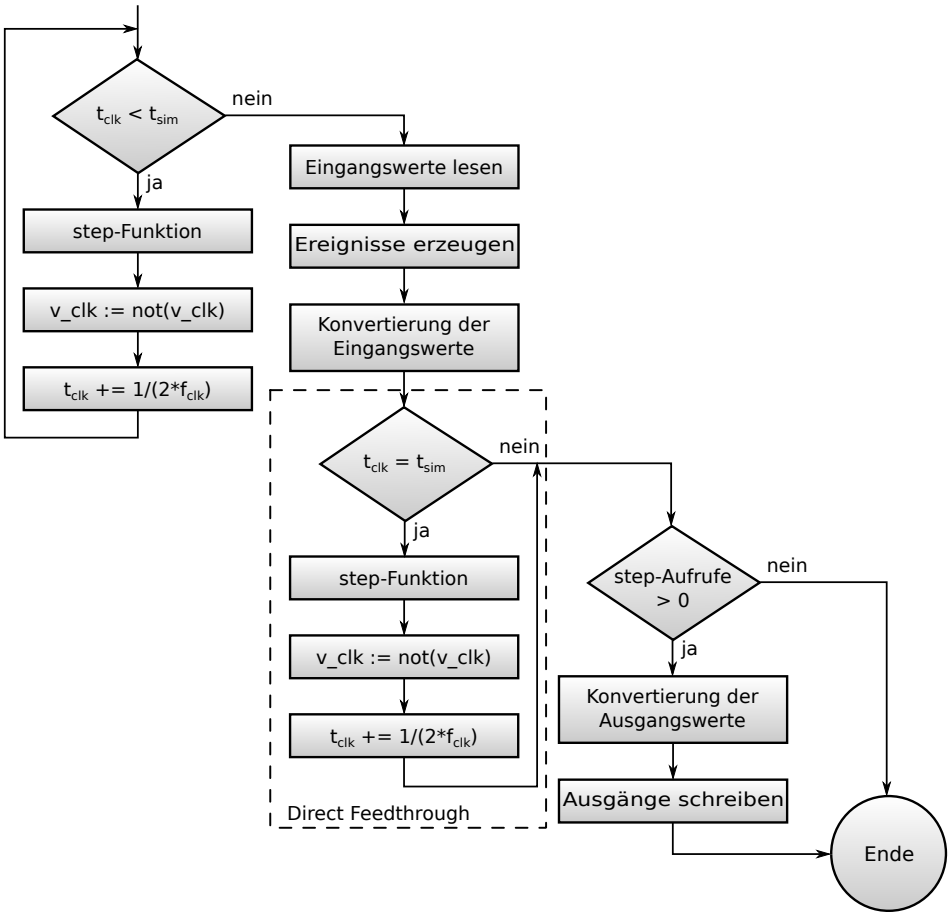



Abbildung 7.4: S-Function-Wrapper mit Skalierung der Taktrate.

Zu Beginn der Simulation müssen beide Variablen initialisiert werden, die Taktvariable *v_clk* mit dem initialen Wert des Taktsignals und $t_{clk}$ mit $t_0$. Danach kann in jedem Zeitschritt der in Abbildung 7.4 dargestellte Ablauf durchgeführt werden. Ist der Wert von $t_{clk}$ kleiner als die aktuelle Simulationszeit in Simulink $t_{sim}$, dann muss mindestens ein Taktereignis verarbeitet werden. Also wird die step-Funktion

aufgerufen. Danach wird der Wert von *v_clk* invertiert und der Wert von $t_{clk}$ wird um einen halben Taktschritt erhöht und somit auf den Zeitpunkt des nächsten Taktereignisses gesetzt. Diese drei Schritte werden solange wiederholt, bis $t_{clk}$ nicht mehr kleiner als $t_{sim}$ ist. Dann werden die aktuellen Eingangswerte gelesen, die entsprechenden Ereignismarkierungen erzeugt und die Eingangswerte konvertiert. Nun verbleibt noch die Verarbeitung eines möglichen Taktereignisses zur aktuellen Simulationszeit. Da $t_{clk}$ immer in halben Taktschritten erhöht wird, genügt hier ein Vergleich zwischen $t_{clk}$ und $t_{sim}$. Gibt es dieses Ereignis, so wird erneut die step-Funktion aufgerufen, *v_clk* invertiert und $t_{clk}$ erhöht. Zuletzt werden die Werte der Ausgänge konvertiert und auf die Ausgangssignale des Simulink-Blocks geschrieben. Da alle Variablen ihre Werte bis zum nächsten Samplehit behalten, ist bei diesem Vorgehen direkt sichergestellt, dass zu Beginn des Ablaufs $t_{clk}$ den Zeitpunkt des ersten Taktereignisses ($t_{1c}$) nach dem letzten Samplezeitpunkt ($t_1$) enthält. Die Variablen der Eingangswerte enthalten die Werte der Eingänge des letzten Samplezeitpunkts ($\omega_{val}(t_1)$).

Soll wie in Abschnitt 7.3.2 die Deltaverzögerung von Ausgangswerten in der Simulink-Simulation sichtbar gemacht werden, entfällt die Verarbeitung des letzten Taktereignisses bei $t_{clk} = t_{sim}$. Dieses Ereignis wird dann erst zum nächsten Samplezeitpunkt verarbeitet.

Diese Form des S-Function-Wrappers kann für beliebige Verhältnisse zwischen Samplerate und Taktrate eingesetzt werden. Insbesondere wird auch die Verwendung von variablen Sampleraten in Simulink unterstützt. Um ein korrektes Simulationsergebnis zu erhalten, muss allerdings vorab sichergestellt werden, dass die in Abschnitt 6.2 angegebenen Bedingungen erfüllt sind. Wenn die Sampleschrittweite in Simulink kein ganzzahliges Vielfaches der Schrittweite der Taktereignisse ist, dann muss sichergestellt werden, dass es in der HDL-Komponente keine Sensitivität auf externe Ereignisse außer den Taktereignissen gibt.

## 7.5 Automatische Transformation von Modellen

Eine der grundlegenden Anforderungen für das Integrationskonzept ist die automatische Anwendbarkeit. Das Erzeugen eines Simulink-Blocks aus einer HDL-Komponente soll ohne manuelle Schritte durch den Entwickler durchführbar sein. Nachdem in den vorherigen Abschnitten die Funktionsweise und Umsetzung des Konzepts beschrieben wurde, wird nun erläutert, wie die einzelnen Schritte automatisiert werden können.

Als Basis für die automatische Transformation des Hardwaremodells kommt ein Werkzeug zur regelbasierten Transformation von Entwurfsbeschreibungen zum

Einsatz [OGR06].

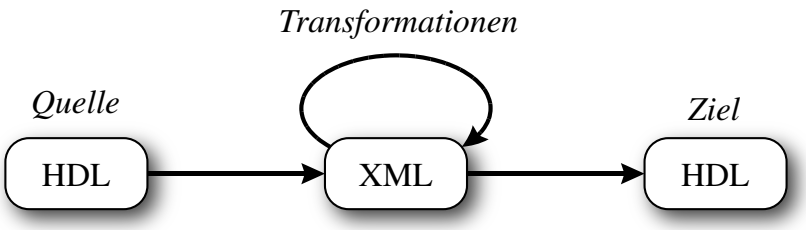



Abbildung 7.5: XML-basierte Transformation von Entwurfsbeschreibungen.

Das Werkzeug ist in der Lage HDL-Beschreibungen einzulesen und in eine XML-Datenstruktur zu überführen. Dann können zuvor definierte Transformationsregeln angewendet werden, und das Ergebnis der Transformation kann wieder in Form einer HDL-Beschreibung ausgegeben werden (Abbildung 7.5). Die XML-Datenstruktur entspricht einem abstrakten Syntaxbaum, angereichert mit einigen semantischen Informationen. Die Transformationsregeln werden über eine spezielle Programmiersprache formuliert. Diese kann automatisch in XSLT-Programme [Wor07] übersetzt werden. Außerdem wird durch diese Sprache sichergestellt, dass das Transformationsergebnis syntaktisch korrekt ist. Die Anwendung von Transformationsregeln auf ein HDL-Modell kann durch eine umfangreiche Kommandozeilenschnittstelle in Stapelverarbeitungsdateien automatisiert werden.

### 7.5.1 Prüfen der Vorbedingungen

Um die Transformation der HDL-Komponente vernehmen zu können, müssen bestimmte Bedingungen erfüllt sein. Es muss sich um ein synthetisierbares HDL-Modell handeln. Die eigentlich relevante Eigenschaft ist, dass keine zeitverzögerten Ereignisse auftreten, beziehungsweise, dass die zeitverzögerten Ereignisse als Ereignisse mit Deltaverzögerung interpretiert werden können. Dies ist notwendig, damit die statische Ablaufplanung angewendet werden kann. Eine zweite Bedingung muss erfüllt sein, wenn bei getakteten HDL-Modellen eine beliebige Skalierung zwischen Taktrate und Samplerate unterstützt werden soll. Dazu darf es in der HDL-Komponente keine Sensitivität auf externe Ereignisse außer dem Taktereignis geben.

In Abschnitt 5.1.2 wurde beschrieben wie unterschiedliche Arten von Ereignissen in VHDL entstehen. Zeitverzögerte Ereignisse entstehen durch verzögerte Signalzuweisungen und durch Wait-for-Time-Anweisungen. Beide werden zu eindeutigen Elementen im XML-Syntaxbaum. Somit kann eine einfache Regel implementiert werden, die nach diesen Elementen im Syntaxbaum sucht. Werden ent-

sprechende Elemente gefunden, kann eine Meldung und die dazugehörige Stelle im Quelltext des Modells ausgegeben werden. Der Entwickler muss dann entscheiden, ob die annotierten Verzögerungen ignoriert werden können und die Verzögerung somit als Deltaverzögerung zu interpretieren ist. Für gewöhnlich ist dies der Fall, da andernfalls auch die Logiksynthese nicht ohne Weiteres möglich ist.

Sollen bei getakteten HDL-Komponenten beliebige Skalierungen zwischen Taktrate und Samplerate oder variable Sampleraten unterstützt werden, so muss sichergestellt werden, dass es keine Prozesse gibt, die auf andere Eingänge als den Takteingang sensitiv sind. Dazu müssen die Sensitivitätslisten aller Prozesse im HDL-Modell untersucht werden. Darin wird dann geprüft, ob einer der Eingänge außer dem Takteingang darin verwendet wird. Dabei müssen auch Eingänge auf tieferen Hierarchieebenen berücksichtigt werden, wenn sie direkt, das heißt ohne Verwendung von Signalen, mit einem der Eingänge auf oberster Ebene verbunden sind. Wird eine solche Sensitivität gefunden, so können keine beliebigen Sampleraten in Simulink verwendet werden. Ein korrektes Simulationsergebnis kann dann nicht automatisch sichergestellt werden. Auch an dieser Stelle kann jedoch gegebenenfalls der Entwickler im Einzelfall entscheiden, ob Prozessaktivierungen asynchron zum Takt für das Simulationsverhalten relevant sind.

### 7.5.2 Interne Ablaufplanung

Sind alle notwendigen Bedingungen erfüllt, so kann die HDL-Komponente nach dem in Abschnitt 7.1 beschriebenen Vorgehen transformiert werden. Eine entsprechende Transformationsregel zur Automatisierung dieses Schritts wurde prototypisch implementiert und im Rahmen der Evaluation des Konzepts erfolgreich bei verschiedenen Modellen angewendet.

Die Transformationsregel ist als Abfolge von mehreren Schritten implementiert. Zunächst werden die sogenannten *Generate Statements* evaluiert. Sie dienen der parametrisierbaren Erzeugung von Prozessen oder Subkomponenten und werden in diesem Schritt entsprechend der gewählten Parameter durch statische Instanzierungen ersetzt. Danach wird die Hierarchie im Modell aufgelöst. Im Ergebnis gibt es nur noch ein Modul, das alle Prozesse und Signale enthält. Im nächsten Schritt werden Daten- und Kontrollfluss analysiert. Das heißt, es wird eine Liste aller Prozesse erstellt, in der festgehalten wird, auf welche Signale der Prozess sensitiv ist, welche Signale in seinem Inneren gelesen und welche beschrieben werden. Dann werden die Prozesse zu Funktionen und die Signale zu Variablen transformiert. Schließlich werden die Ergebnisse der Kontroll- und Datenflussanalyse verwendet, um diese Funktionen anzuordnen wie in Abschnitt 7.1 beschrieben und die notwendigen Kontrollstrukturen für deren Aufruf zu erzeugen.

Im Folgenden werden einzelne Ausschnitte aus der Implementierung der Transformationsregel zur internen Ablaufplanung gezeigt. Die darin verwendete Variable `scheduling_help` enthält eine Baumstruktur mit allen Informationen, die mittels Kontroll- und Datenflussanalyse aus dem Modell extrahiert wurden. Das sind unter anderem Listen von Ein- und Ausgängen, sowie eine Liste der Prozesse. Zu jedem Prozess ist annotiert, auf welche Ereignisse er sensitiv ist und auf welche Signale im Prozess geschrieben wird.

Listing 7.1 zeigt einen Ausschnitt, der die Verarbeitung der externen Ereignisse realisiert. Hier wird zunächst für alle Prozesse (Zeile 2f) geprüft, ob sie sensitiv auf ein Eingangssignal sind (Zeile 6 bis Zeile 14). Wenn ja, dann werden Anweisungen erzeugt, die den Prozess ausführen falls ein entsprechendes Ereignis, das heißt eine Änderung des Werts des Signals, vorliegt.

```
1  <!-- schedule port processes -->
2  <for-each>
3    <select>$scheduling_help/info/process_info/process</select>
4    <variable>
5      <qname>schedule</qname>
6      <for-each>
7        <select>./sensitivity/IDENTIFIER</select>
8        <if>
9          <test>$list_of_in_ports/IDENTIFIER[@text=current()/@text]</test>
10         <value-of>
11           <select>'yes'</select>
12         </value-of>
13       </if>
14     </for-each>
15   </variable>
16   <if>
17     <test>contains($schedule,'yes')</test>
18     <statement>
19       <!-- generate code to run process -->
20       ...
21     </statement>
22   </if>
23 </for-each>
```

Listing 7.1: Transformationsregel: Externe Ereignisse.

In Listing 7.2 wird in der Variablen `list_of_written_signals` eine Liste aller Signale erzeugt, auf die bei der vorhergehenden Ausführung von Prozessen potentiell geschrieben wurde. Das sind die Signale, die in den Prozessen, die auf externe Ereignisse sensitiv sind, als Ziel von Signalzuweisungen auftreten. In Zeile 42ff wird diese Liste dann an das Template `generate_process_schedule_recursion`

übergeben, das entsprechend der internen Ereignisse den Ablauf der folgenden Prozesse bearbeitet. Ausschnitte daraus sind in Listing 7.3 und Listing 7.4 abgebildet.

```
24 <!-- generate list of written signals -->
25 <variable>
26   <qname>list_of_written_signals</qname>
27   <for-each>
28     <select>$list_of_in_ports/*</select>
29     <for-each>
30       <select>$scheduling_help/info/process_info/
31                process[sensitivity/IDENTIFIER[@text = current()/@text]]
32       </select>
33       <for-each>
34         <select>./signal_assign_targets/IDENTIFIER</select>
35           <copy-of>
36           <select>.</select>
37         </copy-of>
38       </for-each>
39     </for-each>
40   </for-each>
41 </variable>
42 <call-template>
43   <qname>generate_process_schedule_recursion</qname>
44   <with-param>
45     <qname>list_of_signals</qname>
46     <copy-of>
47       <select>$list_of_written_signals/*</select>
48     </copy-of>
49   </with-param>
50 </call-template>
```

Listing 7.2: Transformationsregel: Liste beschriebener Signale.

Listing 7.3 zeigt die Erzeugung der Ablaufplanung für die Prozesse, die auf eines der Signale aus der zuvor generierten Liste sensitiv sind. Dies ist vergleichbar mit Listing 7.1, jedoch hier für interne Ereignisse.

In Listing 7.4 wird dann eine neue Liste von Signalen erzeugt, die die Signale enthält, auf die im vorhergehenden Schritt in einem der Prozesse geschrieben wurde. Anschließend wird das Template `generate_process_schedule_recursion` in Zeile 107ff rekursiv mit der neuen Signalliste aufgerufen. Diese Rekursion bearbeitet die Reihe der internen Ereignisse und wird solange fortgesetzt, bis die neue Liste der geschriebenen Signale leer ist. Das ist der Fall, wenn entweder keiner der zuletzt eingeplanten Prozesse Signalzuweisungen enthält oder keine Prozesse auf die zuletzt geschriebenen Signale sensitiv sind. Enthält das Modell zyklische Ab-

```
<!-- run scheduled processes -->
<for-each>
  <select>$scheduling_help/info/process_info/process</select>
  <variable>
    <qname>is_sens_to_signal</qname>
    <for-each>
      <select>./sensitivity/IDENTIFIER</select>
      <if>
        <test>$list_of_signals/IDENTIFIER[@text=current()/@text]</test>
        <value-of>
          <select>'y'</select>
        </value-of>
      </if>
    </for-each>
  </variable>
  <if>
    <test>contains($is_sens_to_signal,'y')</test>
    <statement>
      <!-- schedule process -->
      ...
    </statement>
  </if>
</for-each>
```

Listing 7.3: Transformationsregel: Interne Ereignisse.

hängigkeiten im Datenfluss, so kann das dazu führen, dass die Abbruchbedingung der Rekursion nie erfüllt werden kann. Dieser Sonderfall wird an anderer Stelle bearbeitet. Statt rein sequenziell wird die Ablaufplanung dann in Form einer Schleife erzeugt.

### 7.5.3 S-Function-Wrapper

Eine automatische Generierung des S-Function-Wrappers ist ebenfalls ohne Probleme möglich, da alle dazu benötigten Informationen in der XML-Datenstruktur des Transformationswerkzeugs vorhanden sind. Benötigt werden lediglich Informationen über die Schnittstelle: Anzahl, Namen und Datentypen von Ein- und Ausgängen. Die Methoden zur Konversion der Datentypen zwischen Simulink und Hardwarekomponente können generisch für die gängigen Datentypen bereitgestellt werden.

Lediglich für die Unterstützung der Skalierung der Taktfrequenz werden zusätzliche Eingaben vom Entwickler benötigt. Dazu muss der Name des Takteingangs und der gewünschte Skalierungsfaktor explizit angegeben werden. Der Skalierungsfaktor kann alternativ auch in Form eines Parameters an den S-Function-Block im Simulink-Modell angegeben werden.

Alle weiteren Instruktionen sind eine Anwendung der Simulink-S-Function-Programmierschnittstelle.

## 7.6 Zusammenfassung

In diesem Kapitel wurde das zuvor theoretisch erfasste Konzept auf die praktische Anwendung, die Integration einer HDL-Komponente in ein Simulink-Modell, übertragen. Am Beispiel von VHDL wurde gezeigt, wie die notwendigen Transformationen der Komponente realisiert werden können. Dies geschieht in drei Schritten, der internen Ablaufplanung, der Übersetzung nach C++ und der Generierung des S-Function-Wrappers. Außerdem wurde erläutert, wie diese Schritte und auch die gegebenenfalls notwendige Prüfung von Vorbedingungen auf Basis eines Werkzeugs zur regelbasierten Transformation von Hardwaremodellen automatisiert werden können. Die tatsächliche praktische Anwendbarkeit des Konzepts und der Einfluss auf die Simulation eines Gesamtsystems werden nun anhand von Beispielmodellen überprüft.

```
74 <!-- generate new signal list -->
75 <variable>
76   <qname>new_signal_list</qname>
77   <for-each>
78     <select>$scheduling_help/info/process_info/process</select>
79     <variable>
80       <qname>is_sensitive_to_written_signal</qname>
81       <for-each>
82         <select>./sensitivity/IDENTIFIER</select>
83         <choose>
84           <when>
85             <test>$list_of_signals/IDENTIFIER[@text=current()/@text]
86             </test>
87             <value-of>
88               <select>'y'</select>
89             </value-of>
90           </when>
91         </choose>
92       </for-each>
93     </variable>
94     <if>
95       <test>contains($is_sensitive_to_written_signal,'y')</test>
96       <for-each>
97         <select>./signal_assign_targets/IDENTIFIER</select>
98         <copy-of>
99           <select>.</select>
100        </copy-of>
101      </for-each>
102    </if>
103  </for-each>
104 </variable>
105 <if>
106   <test>$new_signal_list/*</test>
107   <call-template>
108     <qname>generate_process_schedule_recursion</qname>
109     <with-param>
110       <qname>list_of_signals</qname>
111       <copy-of>
112         <select>$new_signal_list/*</select>
113       </copy-of>
114     </with-param>
115   </call-template>
116 </if>
```

Listing 7.4: Transformationsregel: Rekursion interne Ereignisse.

# 8 Evaluation der Simulationsgeschwindigkeit

Das in Kapitel 7 vorgestellte Verfahren zur Transformation der HDL-Komponente und zur Integration in Simulink wurde an einigen Beispielen durchgeführt. Im Folgenden werden die Ergebnisse der Simulation diskutiert, um die Anwendbarkeit des Verfahrens und die Simulationsgeschwindigkeit zu bewerten. Dazu werden neben dem in dieser Arbeit vorgestellten Konzept zwei Ko-Simulations-Werkzeuge für die Integration der HDL-Komponenten in ihre Simulink-Umgebungen verwendet. Die einzelnen Simulationsanordnungen werden dann bezüglich ihrer Simulationsgeschwindigkeit und der Simulationsergebnisse miteinander verglichen.

Neben den Simulationen in Simulink werden auch Simulationen der einzelnen Varianten der HDL-Komponente in reinen HDL-Umgebungen durchgeführt. Diese dienen der Bewertung des Einflusses der Modelltransformation auf die Simulationsgeschwindigkeit der HDL-Komponente selbst. Besonders der Einfluss der Übersetzung von VHDL nach SystemC/C++ ist relevant. Das hier eingesetzte Werkzeug wurde mit starkem Fokus auf der Lesbarkeit des Ergebnismodells entwickelt und nicht zur Erzeugung besonders schneller Modelle. Deshalb ist die Simulation der SystemC/C++-Modelle langsamer als die Simulation der VHDL-Modelle. Dies ist jedoch kein grundsätzliches Problem der in dieser Arbeit entwickelten Lösung, sondern lediglich eine Eigenschaft des eingesetzten Werkzeugs. Andere Werkzeuge, wie zum Beispiel [Car] oder auch einige VHDL-Simulatoren zeigen, dass es möglich ist aus VHDL-Modellen äquivalente C++-Modelle zu erzeugen, deren Simulationsgeschwindigkeit nicht langsamer ist als eine VHDL-Simulation. Die Bewertung der Simulationsgeschwindigkeit bei der optimierten Integration der HDL-Komponente in eine Simulink-Umgebung wird deshalb im Folgenden in erster Linie anhand des Vergleichs mit der Ko-Simulation von SystemC-Modellen vorgenommen. Da zu deren Erzeugung dasselbe Übersetzungswerkzeug eingesetzt wurde, kommt dessen Einfluss darin in gleichem Maße zum Tragen.

## 8.1 Szenario

Ausgangspunkt für die Bewertung sind verschiedene VHDL-Modelle, die jeweils in einer Simulink-Umgebung simuliert werden sollen. Zur Integration der VHDL-Komponenten in die Simulink-Simulation werden vier verschiedene Varianten realisiert:

**OPT:** Hierbei handelt es sich um das in dieser Arbeit entwickelte Verfahren. Das Modell der Hardwarekomponente wird nach dem in Kapitel 7 beschriebenen Vorgehen transformiert und mit einem S-Function-Wrapper umgeben. Die so entstandene S-Function wird dann im Simulink-Modell der Umgebung instanziiert.

**SC-KoSim:** Eine Methode, die ein Werkzeug zur Integration von SystemC-Modellen in Simulink verwendet (s. Abschnitt 3.1.2, [HON08]), kombiniert mit dem bereits in Abschnitt 7.2 erwähnten Werkzeug zur Übersetzung von VHDL nach SystemC [GON12]. Zur Simulation des SystemC-Modells wird dabei der von Accellera frei verfügbare SystemC-Simulator [Sys] eingesetzt. Die Integration in Simulink wird hier ebenfalls über eine S-Function realisiert, die den Datenaustausch und deren Konversion, sowie die Synchronisation der Simulatoren übernimmt. Diese Methode repräsentiert eine ganze Klasse von Methoden, die auf dem Accellera SystemC-Simulator basieren und das S-Function-Interface für die Integration in Simulink verwenden. Abgesehen von Details in der Implementierung ist ihre Funktionsweise gleich.

**HdlVerif-SC:** Die HDL-Verifier Toolbox von Mathworks (s. Abschnitt 3.1.1) wird in Kombination mit Mentor ModelSim zur Simulation des SystemC-Modells der Hardwarekomponente verwendet. Das SystemC-Modell wurde mit dem bereits in Abschnitt 7.2 erwähnten Werkzeug zur Übersetzung von VHDL nach SystemC aus dem VHDL-Modell erzeugt. Die Integration geschieht über einen speziellen Ko-Simulationsblock, der Synchronisation und Datenaustausch zwischen Simulink und ModelSim realisiert.

Die HDL-Verilator Toolbox ist die von Mathworks angebotene Methode zur Integration von HDL-Modellen in Simulink und stellt damit den herstellerseitig empfohlenen Weg dar. Sie ist schwergewichtiger als die SC-KoSim-Methode, da hierbei kommerzielle HDL-Simulatoren mit großem Funktionsumfang für die Simulation der HDL-Komponente eingesetzt werden.

**HdlVerif-VHD:** Die HDL-Verifier Toolbox von Mathworks in Kombination mit Mentor ModelSim zur Simulation des VHDL-Modells der Hardwarekomponente. Hier wird das ursprüngliche VHDL-Modell verwendet. Wie bereits erwähnt ist die VHDL-Simulation schneller als die C++- oder SystemC-Ver-

sionen des Modells. Dies ist kein generelles Problem des Ansatzes, sondern lediglich ein Effekt, der durch das verwendete Werkzeug zur Übersetzung von VHDL nach C++ beziehungsweise SystemC auftritt. In Abschnitt 8.2.4 wird darauf nochmal näher eingegangen. Zur Bewertung der Simulationsgeschwindigkeit und des Mehraufwands für die Ko-Simulation ist diese Variante somit nicht geeignet. Deshalb werden die Ergebnisse beim Vergleich der Geschwindigkeit etwas abgesetzt dargestellt.

Anhand von Messungen der Simulationsgeschwindigkeit in den einzelnen Szenarien soll gezeigt werden, dass der in dieser Arbeit vorgestellte Ansatz verglichen mit herkömmlichen Methoden der Ko-Simulation eine höhere Geschwindigkeit bei der Simulation des Gesamtsystems bietet. Das Verhalten der HDL-Komponente bleibt aus Sicht der Simulink-Umgebung gleich. Um dies zu zeigen, werden die Werte an den Ein- und Ausgängen der HDL-Komponenten in jedem Zeitschritt aufgezeichnet und als Text in Dateien gespeichert. Diese Dateien können dann mit dem Werkzeug *diff* [GNU], einem Programm zum textbasierten Vergleich von Dateien, auf Gleichheit überprüft werden. Die Aufzeichnungen der HdlVerif-VHD-Methode dienen als Referenzwerte. Da bei der Aufzeichnung der Werte große Datenmengen entstehen, wurde sie während der eigentlichen Messung der Simulationsdauer deaktiviert, um Einflüsse von Dateioperationen zu vermeiden.

Um die Einflüsse der Modelltransformation bewerten zu können, werden außerdem Messungen der Simulationsgeschwindigkeit in reinen HDL-Simulationen durchgeführt. Neben dem ursprünglichen VHDL-Modell werden das automatisch übersetzte SystemC-Modell, das SystemC-Modell in einem VHDL-Wrapper und das optimierte C++-Modell untersucht. Das SystemC-Modell mit VHDL-Wrapper dient dabei zur Abschätzung des Mehraufwands bei der Ko-Simulation mehrerer Hardwarebeschreibungssprachen. Das optimierte C++-Modell ist dazu in einen SystemC-Wrapper eingebettet, der ähnlich wie der S-Function-Wrapper die Eingangswerte entgegennimmt, die step-Funktion aufruft und schließlich die Ausgangswerte auf Ausgangsports schreibt. Die verschiedenen Varianten der Hardwarekomponente werden jeweils in einer Verilog-Testumgebung in einem kommerziellen HDL-Simulator simuliert. Für alle Varianten eines Modells wird stets dieselbe Verilog-Testumgebung verwendet. Die Verwendung von Verilog bietet sich an dieser Stelle an, da diese Testumgebungen bereits existieren und nicht neu erstellt werden müssen.

**VHD:** VHDL-Modell der Hardwarekomponente in der Verilog-Testumgebung.

**SC:** SystemC-Modell der Hardwarekomponente in der Verilog-Testumgebung.

**OPT+SC-Wr:** Das nach dem in Kapitel 7 vorgestellten Verfahren transformierte Modell der Hardwarekomponente in einem SystemC-Wrapper, der dann in der Verilog-Testumgebung simuliert wird.

**SC+VHD-Wr:** SystemC-Modell der Hardwarekomponente in einem VHDL-Ko-Simulations-Wrapper, der dann in der Verilog-Testumgebung simuliert wird. So kann im Vergleich mit der SC-Variante die Auswirkung der Ko-Simulation mehrerer HDLs untersucht werden.

Neben der Messung der Simulationsgeschwindigkeit werden auch hier Aufzeichnungen der Ein- und Ausgangswerte der einzelnen Varianten verglichen, um die funktionale Äquivalenz der einzelnen Modelle zu zeigen.

### 8.1.1 Verwendete Modelle

Zur Evaluation kommen vier verschiedene Modelle zum Einsatz.

**GGT** Hierbei handelt es sich um eine Hardwarekomponente zur Berechnung des größten gemeinsamen Teilers zweier Zahlen. Die Komponente hat insgesamt vier Eingänge und zwei Ausgänge, besteht aus lediglich einem Prozess und hat einen Umfang von 70 Zeilen VHDL-Code. Die in Simulink modellierte Umgebung besteht aus einem Block zur Generierung des Taktsignals, einem Block zu Generierung des Reset-Signals, sowie zwei Ramp-Blöcken zur Stimulation der Werteingänge.

**DSP_1** Das Modell der Hardware beinhaltet einen konfigurierbaren Signalverarbeitungsprozessor (DSP) und einige periphere Komponenten. Es handelt sich um ein industrielles Design der Robert Bosch GmbH. Insgesamt ist in diesem Beispiel ein Filteralgorithmus implementiert. Das Modell umfasst etwa 2300 Zeilen VHDL-Code und besteht aus vierzehn unterschiedlichen Modulen sowie einem Paket mit Konstantendefinitionen. Es hat fünf Eingänge, zwei Ausgänge und besteht aus 47 Prozessen. Die Simulink-Testumgebung basiert auf der Aufzeichnung einer Referenzsimulation, die in einer VHDL-Testumgebung durchgeführt wurde. Die Aufzeichnung beinhaltet Zeitreihen mit Stimuluswerten für die Eingänge der HDL-Komponente und Erwartungswerten für ihre Ausgänge. Vor Beginn der Simulink-Simulation wird die Aufzeichnung in eine Matlab-Variable eingelesen. Zur Simulation werden die Zeitreihen dann entsprechend der Aufzeichnung abgespielt und die Ausgänge der HDL-Komponente in jedem Zeitschritt mit den erwarteten Werten verglichen.

**GFB** Das GFB-Modell implementiert eine Gammatone-Filterbank [Pop98]. Dabei handelt es sich um ein Signalverarbeitungssystem aus dem Bereich der Au-

diologie, das aus einer Anordnung mehrerer linearer Filter besteht. Die Filterbank besteht aus 30 Kanälen, die auf unterschiedliche Frequenzbereiche reagieren.

Das verwendete Modell beinhaltet 14 Prozesse und umfasst etwa 1600 Zeilen VHDL-Code. Es hat fünf Eingänge: Takt und Reset, ein *Stop*-Signal zur Unterbrechung der Verarbeitung, den eigentlichen Werteingang für das Audiosignal und einen *Take*-Eingang, der die Verarbeitung eines neuen Wertes startet. Der Werteingang erwartet ein 32-Bit-Signal, in dem ein Stereosignal mit je 16 Bit zusammengefasst ist. Über den *Take*-Eingang wird die Abtastrate des Eingangssignals festgelegt. Darüber hinaus hat das Modell 65 Ausgänge, je zwei Ausgänge pro Filterkanal (Real- und Imaginäranteil) und einige Kontrollausgänge.

Die vorgesehene Taktrate für die Hardwarekomponente beträgt 50 MHz, die Abtastrate für das Audiosignal 20 kHz. In der Simulink-Testumgebung werden dazu zwei Pulsgeneratoren verwendet, der erste mit einer Frequenz von 50 MHz und einer Pulsbreite von 50%, der zweite mit einer Frequenz von 20 kHz und einer Pulsbreite von 1%. Zur Generierung des Audiosignals dienen mehrere Sinusquellen. Außerdem kommt ein Step-Block zum Einsatz, der zu Beginn der Simulation ein Reset der Hardwarekomponente initiiert. Der *Stop*-Eingang wird nicht verwendet und mit einer konstanten Null belegt. Die Ausgänge der Hardwarekomponente werden über Anzeigeelemente ausgegeben.

**DSP_2** Dieses Modell ist ebenfalls ein industrielles Design der Robert Bosch GmbH. Bei der Hardwarekomponente handelt es sich um einen optimierten Signalverarbeitungsprozessor (DSP), der in dieser Form in verschiedenen Produkten der Robert Bosch GmbH eingesetzt wird. In diesem Beispiel ist der DSP Teil eines Drucksensors, der als System-in-Package gefertigt wird. Er übernimmt darin Teile der digitalen Signalverarbeitung.

Der DSP selbst ist in etwa 15000 Zeilen VHDL-Code implementiert, beinhaltet circa 1500 Prozesse und hat jeweils 31 Ein- und Ausgänge. Das Simulink-Modell stellt ein realistisches Modell des gesamten Drucksensor-SiP inklusive analogem Sensor und analoger Vorverarbeitung dar. Programmspeicher und Arbeitsspeicher des DSP sind ebenfalls Teil des Simulink-Modells. Das DSP-Programm reagiert auf verschiedene Interrupts, die in Simulink erzeugt werden. Entsprechend der Interrupts werden dann verschiedene Schritte des Signalverarbeitungsalgorithmus abgearbeitet.

In Tabelle 8.1 sind einige wesentliche Informationen zu den verwendeten Hardwarekomponenten nochmals zusammengefasst.

Tabelle 8.1: Strukturinformationen zu verwendeten Modellen

| | GGT | DSP_1 | GFB | DSP_2 |
|---|---|---|---|---|
| Anzahl Prozesse | 1 | 47 | 14 | ca. 1500 |
| Codezeilen (VHDL) | 70 | 2300 | 1600 | 15000 |
| Eingänge | 4 | 5 | 5 | 31 |
| Ausgänge | 2 | 2 | 65 | 31 |

### 8.1.2 Messumgebung

Für die Untersuchungen wurden MATLAB und Simulink in der Version R2012a mit HDL-Verifier Toolbox Version 4.0 verwendet. Als HDL-Simulator für die Ko-Simulation mit HDL-Verifier und die reinen HDL-Simulationen wurde Mentor ModelSim in Version 10.1b eingesetzt.

Alle Simulink-Simulationen wurden mit aktiviertem *Accelerator Mode* durchgeführt und die C++- und SystemC-Anteile wurden mit eingeschalteten Compiler-Optimierungen (O3) kompiliert.

Die Simulationen wurden auf zwei verschiedenen Rechner durchgeführt. Die Beispiele GGT, DSP_1 und GFB wurden auf einem Rechner mit folgender Spezifikation simuliert:

Prozessor: Intel Core 2 Duo E8400 (3GHz)

Arbeitsspeicher: 3,8 GB

Betriebssystem: openSUSE Linux 12.1 (32 Bit).

Die Simulationen des DSP_2-Beispiels wurden auf folgendem Rechner durchgeführt:

Prozessor: 2 * AMD Opteron DualCore 2222

Arbeitsspeicher: 8 GB

Betriebssystem: Red Hat Enterprise Linux 5 (64 Bit).

#### Messung der Simulationsgeschwindigkeit

Die Messung der Simulationsgeschwindigkeit in Simulink erfolgte, indem jeweils eine Reihe von Simulationsläufen durchgeführt wurde, bei denen der simulierte

Zeitraum gleich blieb. Die Simulation wurde über ein MATLAB-Skript gestartet und mit den MATLAB-Funktionen *tic* und *toc* wurde die Dauer jedes einzelnen Simulationslaufs in Sekunden gemessen[1]. Für jedes Modell wurden pro Integrationsmethode zehn Simulationsläufe durchgeführt, wobei davor je ein separater Simulationslauf durchgeführt wurde, um die Einflüsse durch das Laden des Modells und das Vorbereiten des *Accelerator Mode* zu eliminieren. Zum Vergleich der Simulationsdauer wurde der Durchschnitt der zehn Simulationsläufe verwendet. Zur Bewertung der statistischen Abweichungen ist zusätzlich zu diesen Durchschnittswerten die Standardabweichung angegeben. In der graphischen Darstellung markieren Fehlerindikatoren den jeweils größten und kleinsten Wert der Messreihen.

Zur Messung der Simulationsgeschwindigkeit in der reinen HDL-Simulation wurden ebenfalls je zehn gleiche Simulationsläufe durchgeführt und der Mittelwert daraus ermittelt. Dazu wurde der in ModelSim integrierte Profiler verwendet, der die Dauer eines Simulationslaufs in *Profiling Samples* ausgibt. Eine Umrechnung der *Profiling Samples* in Sekunden ist nicht möglich, zum Vergleich der ModelSim-Simulationen untereinander ist die Einheit jedoch ausreichend.

## 8.2 Einflussgrößen auf die Simulationsgeschwindigkeit

Bevor nun die eigentliche Messung der Simulationsgeschwindigkeit vorgenommen wird, werden nochmals zusammenfassend die wesentlichen Einflussgrößen dargestellt. Sie beeinflussen nicht nur die Simulationsgeschwindigkeit selbst, sondern auch die zu erwartende Verbesserung durch die Modelltransformation. Ein wesentlicher Punkt dabei ist, dass die vorgestellte Methode nicht die Simulation der beiden Modellanteile (Simulink und HDL) beschleunigen soll. Die Verbesserung in der Gesamtsimulation soll durch eine effizientere Zusammenarbeit an der Grenze zwischen den beiden Modellanteilen erreicht werden.

### 8.2.1 Anzahl Synchronisationspunkte/Samplerate

Der Gewinn an Simulationsgeschwindigkeit, der in einer reinen Simulink-Simulation durch eine kleinere Samplerate erreicht werden kann, spielt hier eine geringere Rolle. Er beruht darauf, dass das Modell weniger oft evaluiert werden muss. Dies

[1] Die Verwendung des in Simulink integrierten Profilers war nicht möglich, da die Dauer der Ko-Simulation mit ModelSim damit nicht korrekt ermittelt werden kann. Außerdem wird die Simulationsdauer stark durch den Profiler selbst beeinflusst.

gilt jedoch nicht für die Hardwarekomponente. Bei einer getakteten Hardwarekomponente muss stets die gleiche Anzahl an Taktzyklen simuliert werden, egal ob ein Zeitraum am Stück oder in mehreren kleinen Abschnitten simuliert wird.

Der Mehraufwand, der bei der Ko-Simulation anfällt, ist aber sehr wohl abhängig von der Anzahl der Synchronisationspunkte und damit von der Samplerate in Simulink. Der Aufwand für Synchronisation, Kontextwechsel und Datenaustausch ist bei jedem Synchronisationsvorgang in etwa gleich und weitgehend unabhängig von der Größe des Zeitschritts zwischen zwei Synchronisationspunkten. Die Verbesserung, die durch die effizientere Gestaltung der Synchronisationsvorgänge erreicht werden kann, ist dementsprechend bei vielen Synchronisationspunkten, also bei hohen Sampleraten, größer.

### 8.2.2 Datenaustausch

Der Aufwand für den Datenaustausch und die Konversion der Datentypen hängt neben der Samplerate auch stark von der Menge der auszutauschenden Daten ab. Hier spielt somit die Anzahl der Ein- und Ausgänge der HDL-Komponente und deren Bitbreite eine Rolle. Beschleunigt man den Datenaustausch, so kann demnach bei einer größeren Anzahl von Ein- und Ausgängen ein größerer Einfluss auf die Geschwindigkeit der Gesamtsimulation erwartet werden.

### 8.2.3 Komplexität der Modelle

Ein weiterer Punkt, der sich auf das Verbesserungspotenzial auswirkt, ist die Komplexität der Modelle. Die in dieser Arbeit entwickelte Methode reduziert den Rechenaufwand, der an der Schnittstelle zwischen den beiden Modellteilen entsteht. Der Aufwand für die Berechnung des Modells selbst ist davon nicht betroffen. Da der Mehraufwand für die Ko-Simulation außer in den zuvor genannten Punkten unabhängig von der Berechnungskomplexität des Modells selbst ist, ist der prozentuale Anteil dieses Mehraufwands am Gesamtaufwand bei sehr umfangreichen Modellen geringer. Deshalb kann bei Modellen mit geringerer Berechnungskomplexität eine größere relative Verbesserung der Simulationsgeschwindigkeit erwartet werden, als bei Modellen mit größerer Komplexität.

### 8.2.4 VHDL-nach-SystemC-Transformation

In den im Rahmen dieser Arbeit vorgenommenen Vergleichsuntersuchungen spielt auch das verwendete Werkzeug zur automatischen Generierung des SystemC-, be-

ziehungsweise C++-Codes aus dem ursprünglichen VHDL-Modell eine Rolle. Das Werkzeug legt bei der Transformation großen Wert auf die Lesbarkeit und Wiedererkennbarkeit des generierten Codes. Außerdem bleibt die volle Konfigurierbarkeit des ursprünglichen VHDL-Modells erhalten. Die Simulationsgeschwindigkeit des Transformationsergebnisses steht weniger im Vordergrund. Das führt dazu, dass die Simulation des generierten SystemC- und C++-Codes deutlich langsamer als die Simulation des ursprünglichen VHDL-Codes ist. Dies ist kein generelles Problem. Viele VHDL-Simulatoren erzeugen zur Simulation C/C++-basierte Zwischenmodelle, und es gibt andere Werkzeuge, die sehr schnelle, äquivalente C++-Programme aus VHDL-Modellen erzeugen können [Car]. Prinzipiell ist es kein Problem, die vorgestellte Methode auch mit einem solchen Werkzeug zu kombinieren. Zur Evaluation steht jedoch nur die in Abschnitt 7.2 beschriebene Lösung zur Verfügung. Der Einfluss, der durch die Modelltransformation entsteht, wird in der reinen HDL-Simulation deutlich. Zur Bewertung der Integrationsmethode in Simulink ist der Vergleich mit der SystemC-Ko-Simulation folglich besser geeignet. Aus diesem Grund werden die Ergebnisse der VHDL-Ko-Simulation etwas abgesetzt dargestellt.

## 8.3 Evaluation der Modelle

Nachdem der Messaufbau für die Evaluation der Simulationsgeschwindigkeit und die verwendeten Modelle beschrieben wurden, werden nun die Messergebnisse im Einzelnen dargestellt.

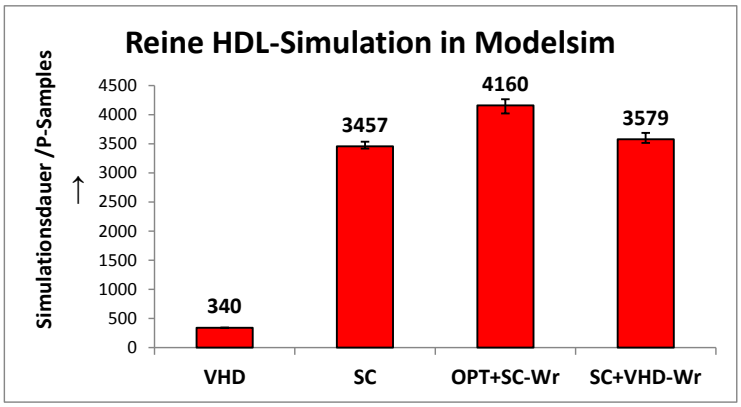
Zunächst wird die Simulation ohne Skalierung der Taktfrequenz untersucht. Die Generierung des Taktsignals ist Teil des Umgebungsmodells in Simulink. Somit ist die Samplefrequenz in Simulink auf das Doppelte der Taktfrequenz festgelegt, das heißt, jeder Zeitschritt in Simulink entspricht einem halben Taktschritt in der HDL-Komponente. Im Anschluss wird am Beispiel des GFB-Modells gezeigt, wie die Skalierung der Taktfrequenz eingesetzt werden kann, und es wird untersucht, wie sich das auf die Simulationsgeschwindigkeit auswirkt.

Die Benennung der einzelnen Varianten entspricht den in Abschnitt 8.1 eingeführten Bezeichnungen.

### 8.3.1 GGT

In Abbildung 8.1 sind zunächst die Ergebnisse der reinen HDL-Simulationen dargestellt. Es zeigt sich, dass die Simulation der VHDL-Variante deutlich schneller

ist als die SystemC-Varianten. Außerdem ist erkennbar, dass die OPT+SC-Wr-Variante langsamer ist als die SC-Variante. Es kann davon ausgegangen werden, dass der Mehraufwand durch den zusätzlichen SystemC-Wrapper entsteht, die Modelle selbst aber etwa gleich schnell sind. Die SC+VHD-Wr-Variante ist nur geringfügig langsamer als die SC-Variante. Das zeigt, dass die Ko-Simulation mehrerer Hardwarebeschreibungssprachen in ModelSim nur wenig Mehraufwand erfordert.

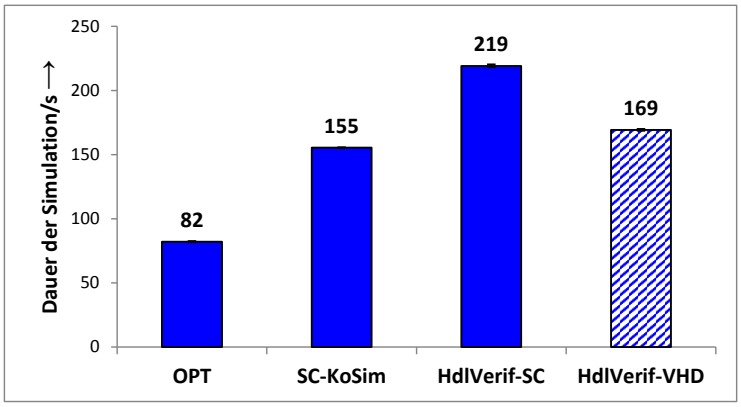



| | VHD | SC | OPT+SC-Wr | SC+VHD-Wr |
|---|---|---|---|---|
| Simulationsdauer | 340,3 PS | 3457 PS | 4159,7 PS | 3578,8 PS[2] |
| Standardabweichung | 2,100 PS | 38,722 PS | 78,711 PS | 65,029 PS |

Abbildung 8.1: Dauer der Simulation des GGT-Modells in Verilog-Umgebung.

Die Dauer der Simulationsläufe mit dem GGT-Modell in der Simulink-Testumgebung ist in Abbildung 8.2 als Diagramm dargestellt. Es handelt sich um ein sehr kleines Modell, so dass der zusätzliche Aufwand für die Ko-Simulation einen großen Anteil am Gesamtaufwand der Simulation hat. Die Ergebnisse zeigen, dass die OPT-Variante deutlich schneller ist als die anderen Varianten. Hier zeigt sich auch ein Vorteil der SC-KoSim-Variante gegenüber der HdlVerif-SC-Variante. Da der HDL-Simulator in der SC-KoSim-Variante deutlich kompakter ist als das in der HdlVerif-SC-Variante eingesetzte Simulationswerkzeug, können die Kontextwechsel zwischen den Simulatoren mit geringerem Mehraufwand durchgeführt werden. Die HdlVerif-VHD-Variante ist ebenfalls schneller als die HdlVerif-SC-Variante. Dies ist auf die schnellere Simulation des VHDL-Modells gegenüber dem SystemC-Modell zurückzuführen.

[2]PS: Profiling-Samples (Ausgabeeinheit des ModelSim Profilers)




| | OPT | SC-KoSim | HdlVerif-SC | HdlVerif-VHD |
|---|---|---|---|---|
| Simulationsdauer | 82,1 s | 155,4 s | 219,1 s | 169,2 s |
| Standardabweichung | 0,265 s | 0,198 s | 0,671 s | 0,334 s |

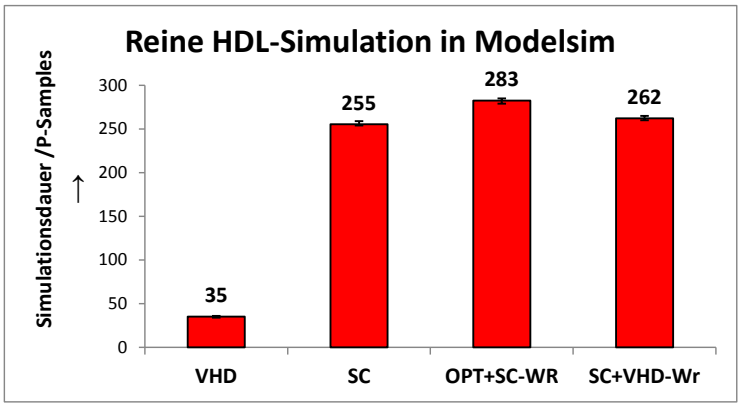
Abbildung 8.2: Dauer der Simulation des GGT-Modells in Simulink-Umgebung.

Beim Vergleich der aufgezeichneten Ein- und Ausgangswerte konnten sowohl bei den reinen HDL-Simulationen, als auch bei den Simulationsläufen in der Simulink-Umgebung keinerlei Unterschiede festgestellt werden.

### 8.3.2 DSP_1

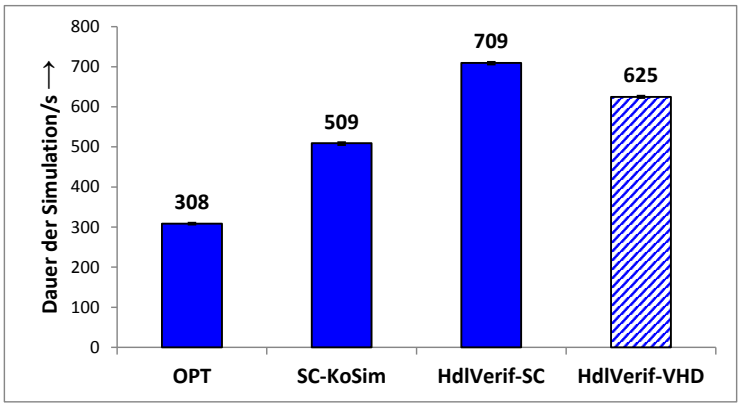
Das DSP_1-Modell bestätigt die Erkenntnisse aus der Evaluation des GGT-Modells. Auch in diesem Fall betrachten wir zunächst die Ergebnisse der reinen HDL-Simulation (Abbildung 8.3). Es zeigt sich erneut, dass die Simulation mit VHDL-Komponente deutlich schneller ist als die übrigen. Außerdem sind die Varianten mit zusätzlichen Wrappern etwas langsamer als die SC-Variante. Zieht man vom Ergebnis der OPT+SC-Wr-Variante den geschätzten Mehraufwand für den SystemC-Wrapper ab, so zeigt sich, dass das transformierte C++-Modell und das SystemC-Modell etwa gleich schnell sind.

Abbildung 8.4 zeigt die Ergebnisse der Simulation in Simulink. Es ist ebenfalls zu erkennen, dass die optimierte Variante einen deutlichen Geschwindigkeitsvorteil gegenüber allen anderen bietet. Auch in diesem Beispiel ist die SC-KoSim-Variante schneller als die Varianten mit HDL-Verifier, und die HdlVerif-VHD-Variante ist aufgrund der schnelleren Simulation des VHDL-Modells schneller als die HdlVerif-SC-Variante.




| | VHD | SC | OPT+SC-Wr | SC+VHD-Wr |
|---|---|---|---|---|
| Simulationsdauer | 35,3 PS | 255,4 PS | 282,6 PS | 262,3 PS |
| Standardabweichung | 0,781 PS | 1,497 PS | 1,685 PS | 1,418 PS |

Abbildung 8.3: Dauer der Simulation des DSP_1-Modells in Verilog-Umgebung.




| | OPT | SC-KoSim | HdlVerif-SC | HdlVerif-VHD |
|---|---|---|---|---|
| Simulationsdauer | 308,4 s | 509,3 s | 709,4 s | 624,8 s |
| Standardabweichung | 1,194 s | 1,914 s | 2,100 s | 1,749 s |

Abbildung 8.4: Dauer der Simulation des DSP_1-Modells in Simulink-Umgebung.

Die Aufzeichnungen der Ein- und Ausgangswerte haben auch in diesem Fall keinerlei Abweichungen zwischen den einzelnen Varianten gezeigt.

### 8.3.3 GFB

Das GFB-Modell ist ebenfalls ein mittelgroßes Modell. Eine Besonderheit ist, dass es im Verhältnis zur Größe des Modells viele Ausgänge gibt, das heißt, dass es eine sehr umfangreiche Schnittstelle zur Umgebung gibt. Dementsprechend ist auch der Aufwand für den Austausch von Daten und deren Konvertierung hoch.

Für dieses Modell liegt keine Verilog-Testumgebung vor, deshalb werden die reinen HDL-Simulationen in einer VHDL-Umgebung durchgeführt. Die Simulation des VHDL-Modells ist somit keine Ko-Simulation zweier HDLs. Um den Einfluss der VHDL-SystemC-Ko-Simulation in den anderen Varianten bewerten zu können, wird zusätzlich eine Simulation des VHDL-Modells in einem SystemC-Wrapper durchgeführt. Dieser Wrapper hat keinerlei eigene Funktionalität, er leitet lediglich die Ein- und Ausgänge der Komponente weiter. Die Varianten in der reinen HDL-Simulation sind somit:

**VHD:** Das VHDL-Modell der Hardwarekomponente in einer VHDL-Testumgebung.

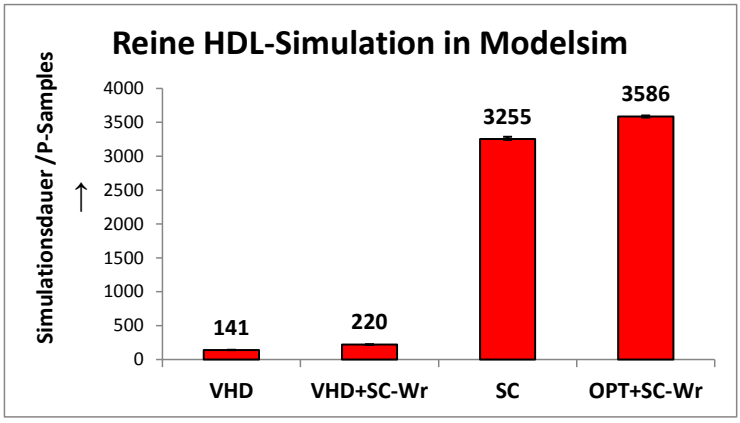
**VHD+SC-Wr:** Das VHDL-Modell der Hardwarekomponente in einem SystemC-Ko-Simulation-Wrapper, der in einer VHDL-Testumgebung simuliert wird.

**SC:** Das SystemC-Modell der Hardwarekomponente in einer VHDL-Testumgebung.

**OPT+SC-Wr:** Das nach dem in Kapitel 7 vorgestellten Verfahren transformierte Modell der Hardwarekomponente in einem SystemC-Wrapper, der dann in einer VHDL-Testumgebung simuliert wird.

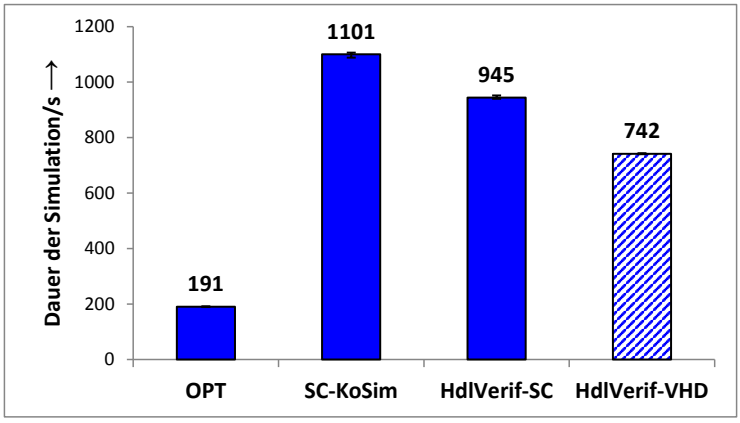
Abbildung 8.5 zeigt die Ergebnisse dieser Simulationsläufe. Auch in diesem Beispiel ist zu erkennen, dass das SystemC-Modell und das optimierte Modell deutlich langsamer sind als die VHDL-Simulation. Der Einfluss der VHDL-SystemC-Ko-Simulation ist in der VHD+SC-Wr-Variante zu erkennen, er spielt jedoch offensichtlich eine wesentlich geringere Rolle als die Sprachtransformation von VHDL nach SystemC, bzw. C++. Ebenfalls ist auch hier zu sehen, dass die optimierte Hardwarekomponente und das SystemC-Modell etwa gleich schnell sind.

Die Simulation in der Simulink-Umgebung (Abbildung 8.6) zeigt, dass die Optimierung bei diesem Beispiel eine große Wirkung hat. Gegenüber der HdlVerif-SC-Variante ergibt sich eine Beschleunigung um Faktor 5, gegenüber der SC-KoSim-Variante sogar 5,8. Die umfangreiche Schnittstelle der Hardwarekomponente erlaubt somit eine deutliche Verbesserung der Simulationsgeschwindigkeit




| | VHD | VHD+SC-Wr | SC | OPT+SC-Wr |
|---|---|---|---|---|
| Simulationsdauer | 141 PS | 220 PS | 3255 PS | 3586 PS |
| Standardabweichung | 1,789 PS | 1,721 PS | 16,88 PS | 10,48 PS |

Abbildung 8.5: Dauer der Simulation des GFB-Modells in VHDL-Umgebung.



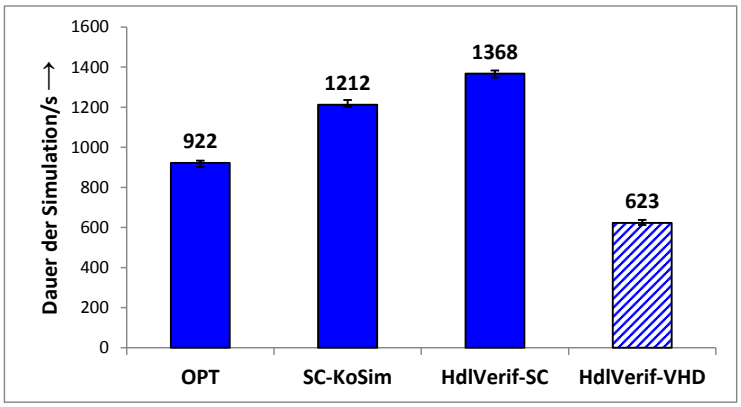

| | OPT | SC-KoSim | HdlVerif-SC | HdlVerif-VHD |
|---|---|---|---|---|
| Simulationsdauer | 190,6 s | 1100,6 s | 944,7 s | 741,6 s |
| Standardabweichung | 0,664 s | 5,211 s | 3,805 s | 1,071 s |

Abbildung 8.6: Dauer der Simulation des GFB-Modells in Simulink-Umgebung.

durch die effizientere Integration und den dadurch schnelleren Datenaustausch. In diesem Beispiel ist außerdem die SC-KoSim-Variante langsamer als die HdlVerif-SC-Variante. Die HdlVerif-VHD-Variante ist wie in den vorherigen Beispielen aufgrund des schnelleren VHDL-Modells geringfügig schneller als die HdlVerif-SC-Variante.

### 8.3.4 DSP_2

Das DSP_2-Modell ist ein großes Modell. Sowohl die Hardwarekomponente als auch das Umgebungsmodell in Simulink sind äußerst umfangreich. Demnach ist auch der Aufwand für die eigentliche Simulation der Modelle hoch und der relative Anteil des Mehraufwands für die Ko-Simulation am Gesamtaufwand gering. Demnach ist auch die zu erwartende Beschleunigung der Simulation geringer als in den anderen Beispielen.

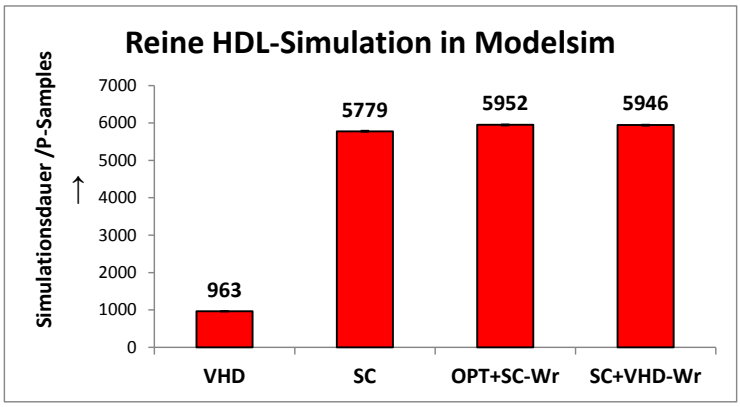



| | OPT | SC-KoSim | HdlVerif-SC | HdlVerif-VHD |
|---|---|---|---|---|
| Simulationsdauer | 922,3 s | 1211,8 s | 1367,9 s | 622,7 s |
| Standardabweichung | 12,30 s | 14,08 s | 15,20 s | 10,84 s |

Abbildung 8.7: Dauer der Simulation des DSP_2-Modells in Simulink-Umgebung.

In Abbildung 8.7 sind die Laufzeiten der einzelnen Varianten in der Simulink-Simulation dargestellt. Auch hier zeigt sich, dass die Simulationsgeschwindigkeit der OPT-Variante schneller ist als die der SC-KoSim- und der HdlVerif-SC-Variante. Die Beschleunigung gegenüber der SC-KoSim-Variante beträgt etwa 25 %, gegenüber der HdlVerif-SC-Variante etwa 33 %. Die HdlVerif-VHD-Variante ist bei diesem Modell die schnellste. Der Vorteil des deutlich schnelleren VHDL-Modells wirkt sich besonders deutlich aus.

Die Messungen, die in einer reinen HDL-Simulation durchgeführt wurden, bestätigen die Ergebnisse der anderen Modelle (Abbildung 8.8). Das VHDL-Modell simuliert deutlich schneller als die anderen Varianten. Die drei C++-basierten Modelle sind etwa gleich schnell. In der OPT+SC-Wr- und der SC+VHD-Wr-Variante sind im Vergleich mit der SC-Variante lediglich leichte Unterschiede aufgrund der zusätzlichen Wrapper zu beobachten.




| | VHD | SC | OPT+SC-Wr | SC+VHD-Wr |
|---|---|---|---|---|
| Simulationsdauer | 962,7 PS | 5778,5 PS | 5951,5 PS | 5946 PS |
| Standardabweichung | 5,658 PS | 10,817 PS | 14,193 PS | 13,405 PS |

Abbildung 8.8: Dauer der Simulation des DSP_2-Modells in Verilog-Umgebung.

### 8.3.5 Zusammenfassung

Die Evaluation der Beispiele hat gezeigt, dass die optimierte Integration der HDL-Komponente in Simulink in allen Fällen zu einer schnelleren Simulation führt als die Integration des SystemC-Modells mit herkömmlichen Methoden zur Ko-Simulation. Die Geschwindigkeitssteigerung ist bei kleinen und mittleren Modellen am größten (Faktor 1,8 bis 5). Das umfangreichste Beispiel (DSP_2) erlaubt lediglich eine Verbesserung um 25 %. Der Einfluss der Modelltransformationen wurde in Simulationen in einer HDL-Umgebung untersucht. Dabei konnte beobachtet werden, dass das SystemC-Modell und das für die Simulink-Integration optimierte C++-Modell etwa gleich schnell sind. Die Simulation des ursprünglichen VHDL-Modells ist deutlich schneller. Die Ko-Simulation dieses Modells in der Simulink-Umgebung ist deshalb nicht geeignet, um die Methode zur Integration in Simulink zu bewerten. Der Vergleich der aufgezeichneten Ein- und Ausgangs-

werte der HDL-Komponente hat bei allen Modellen gezeigt, dass es zwischen den einzelnen Modellvarianten keinerlei Unterschiede im in Simulink beobachtbaren Verhalten gibt. Die Werte aller Ports haben zu jedem Zeitpunkt mit der Referenzsimulation übereingestimmt.

## 8.4 Evaluation der Frequenzskalierung

In den Untersuchungen im vorangegangenen Abschnitt wurde das Taktsignal für die Hardwarekomponenten in den jeweiligen Simulink-Umgebungsmodellen erzeugt. Damit ist auch die Samplefrequenz in der Simulink-Simulation festgelegt: sie muss das Doppelte der für die Hardwarekomponente vorgesehenen Taktfrequenz betragen. Nur dann können die beiden Taktflanken korrekt erzeugt werden. In diesem Abschnitt soll nun das in Kapitel 6 beschriebene Konzept verwendet werden, um die Erzeugung des Taktsignals in den S-Function-Block zu verschieben und eine Skalierung zwischen Samplefrequenz und Taktfrequenz zu ermöglichen.

Die Untersuchungen finden anhand des GFB-Beispiels statt. Es bietet die geeigneten Voraussetzungen, um eine Skalierung der Frequenzen sinnvoll einzusetzen. Wie in Abschnitt 8.1.1 beschrieben, beträgt die gewünschte Abtastrate für das Audiosignal $20\,kHz$ und die Taktrate für die Hardwarekomponente $50\,MHz$. Wenn das Taktsignal nicht mehr in der Simulink-Umgebung erzeugt wird, kann die Samplefrequenz reduziert werden. Die Bedingung, die eine HDL-Komponente erfüllen muss, um für die Skalierung geeignet zu sein, ist ebenfalls gegeben: Kein Prozess ist sensitiv auf ein externes Ereignis außer den Taktereignissen.

In der Simulink-Umgebung wird die Samplefrequenz nun durch das *Take*-Signal begrenzt. Es soll eine Frequenz von $20\,kHz$ und eine Pulsbreite von 1 % haben. Das bedeutet, dass im Abstand von $50\,\mu s$ eine positive Signalflanke erzeugt werden muss und die negative Flanke jeweils $500\,ns$ danach stattfindet. Bei Verwendung einer festen Samplefrequenz ist der zweite Wert entscheidend, die kleinste mögliche Frequenz ist $2\,MHz$. Wird in Simulink eine variable Samplefrequenz verwendet, so kann die Simulation in abwechselnden Schritten von $500\,ns$ und $49500\,ns$ durchgeführt werden. So bleibt das Verhalten des Modells insgesamt erhalten.

Die Skalierung der Taktfrequenz erzeugt lediglich einen Unterschied im Zeitverhalten an den Ausgängen der Hardwarekomponente. Nach der positiven Flanke des *Take*-Signals übernimmt die Hardwarekomponente den aktuellen Wert des Eingangs für das Audiosignal. Dann werden nacheinander die Ergebniswerte für die einzelnen Kanäle berechnet. Diese werden nicht alle gleichzeitig auf die Ausgangsports geschrieben, sondern sobald sie verfügbar sind. Da neue Werte an den Aus-

gängen der Hardwarekomponente nur zu Samplezeitpunkten in Simulink sichtbar werden und die Samplefrequenz in diesem Fall nicht an die Aktivität der Ausgangssignale angepasst ist, kann es bei einer kleineren Samplefrequenz dazu führen, dass Änderungen der Ausgangswerte zu einem späteren Zeitpunkt in der Simulink-Simulation erscheinen, als dies bei der taktgenauen Simulation der Fall wäre. Aus Sicht der Gesamtsimulation ist dieses Verhalten nicht problematisch, da die Ausgangswerte mit einer Frequenz von 20 $kHz$ weiterverarbeitet werden. Die nachgelagerten Blöcke in Simulink, die mit dieser Frequenz arbeiten, sehen die neuen Ausgangswerte in allen Fällen gleichzeitig.

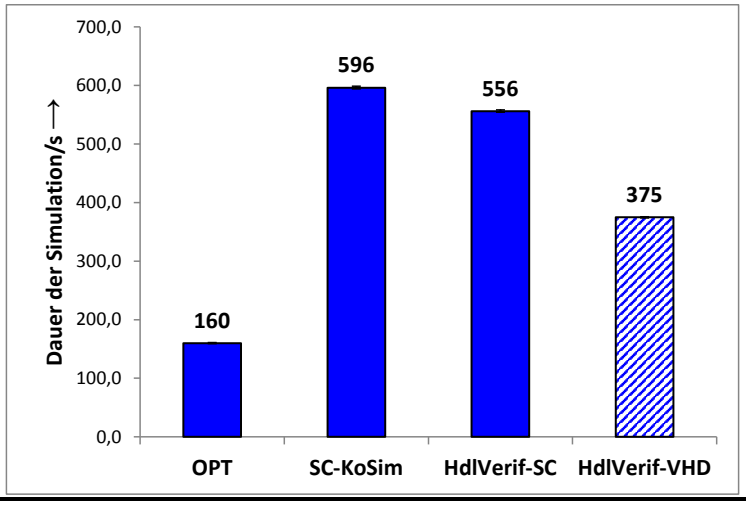
Auch bezüglich der Vergleichbarkeit und der Bewertung der unterschiedlichen Methoden zur Integration der Hardwarekomponente in Simulink spielt dieses Verhalten keine Rolle, da es bei allen Integrationsmethoden gleich ist.

Um den Einfluss der Samplefrequenz bei konstanter Taktfrequenz auf die Simulationsgeschwindigkeit zu untersuchen, werden nun Messungen mit unterschiedlichen Samplefrequenzen durchgeführt. In Abschnitt 8.3.3 wurde bereits eine Messreihe mit Schrittweite 1 durchgeführt. Das entspricht einer Frequenz von 100 $MHz$. Im Folgenden werden nun Messreihen mit den Schrittweiten 2, 5 und 50, also Samplefrequenzen von 50 $MHz$, 20 $MHz$ und 2 $MHz$, durchgeführt. Außerdem wird die Simulationsgeschwindigkeit bei Verwendung einer variablen Schrittweite gemessen. Alle Messungen werden wie in Abschnitt 8.1.2 angegeben durchgeführt.

In Abbildung 8.9 ist zunächst die Dauer der Simulationsläufe mit einer Schrittweite von 2 dargestellt. Die OPT-Variante ist wie auch in Abschnitt 8.3.3 schneller als die anderen Varianten. Der Abstand ist jedoch deutlich kleiner geworden. Während die Simulationsdauer bei der OPT-Variante im Vergleich mit den Ergebnissen aus Abbildung 8.6 lediglich um etwa 15 % geringer wurde, ist sie in der HdlVerif-SC-Variante um etwa 40 % reduziert, in der SC-KoSim-Variante sogar um etwa 45 %. Daraus ist deutlich zu sehen, dass der Aufwand für Synchronisation, Kontextwechsel und Datenaustausch je Synchronisationspunkt bei der optimierten Integrationsmethode deutlich geringer ist. Hier ist das Einsparpotential durch die geringere Anzahl an Synchronisationspunkten deutlich kleiner.

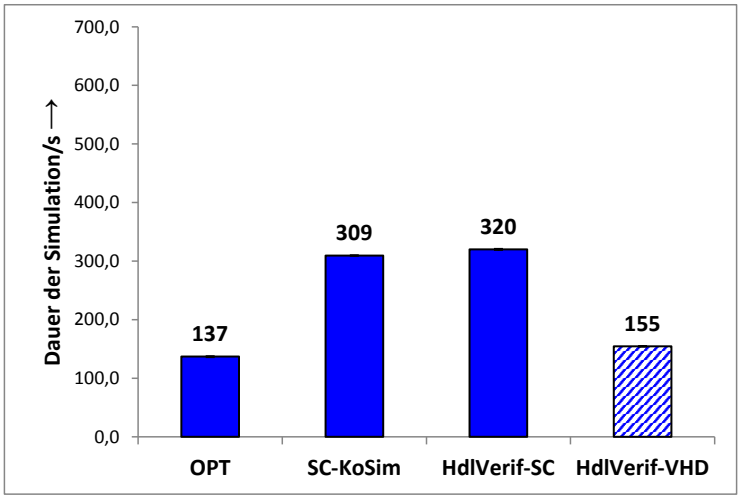
Bei der Simulation mit Schrittweite 5 (Abbildung 8.10) setzt sich dieser Trend fort. Der Abstand zwischen der OPT-Variante wird nochmals geringer, sie ist jedoch nach wie vor schneller als die übrigen Varianten.

Abbildung 8.11 zeigt die Ergebnisse der Simulation mit Schrittweite 50. Der Abstand zwischen der OPT-Variante und den Varianten SC-KoSimund HdlVerif-SChat sich weiter verringert. Der Vorteil beträgt nur noch etwa 20 %. Außerdem ist die HV_VH-Variante nochmals deutlich schneller als die OPT-Variante. Bei dieser Schrittweite dominiert der geringere Aufwand zur Simulation des VHDL-Modells den höheren Aufwand für die Ko-Simulation.



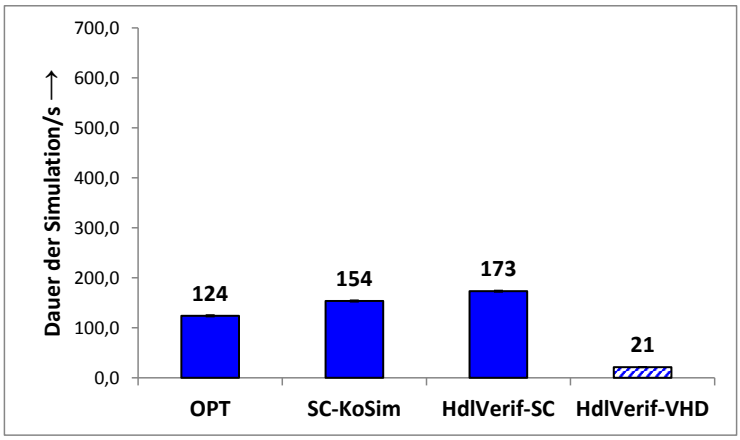

| | OPT | SC-KoSim | HdlVerif-SC | HdlVerif-VHD |
|---|---|---|---|---|
| Simulationsdauer | 159,9 s | 596,1 s | 556,0 s | 374,9 s |
| Standardabweichung | 0,540 s | 1,414 s | 1,084 s | 0,573 s |

Abbildung 8.9: Dauer der Simulation des GFB-Modells in Simulink-Umgebung mit Schrittweite 2.



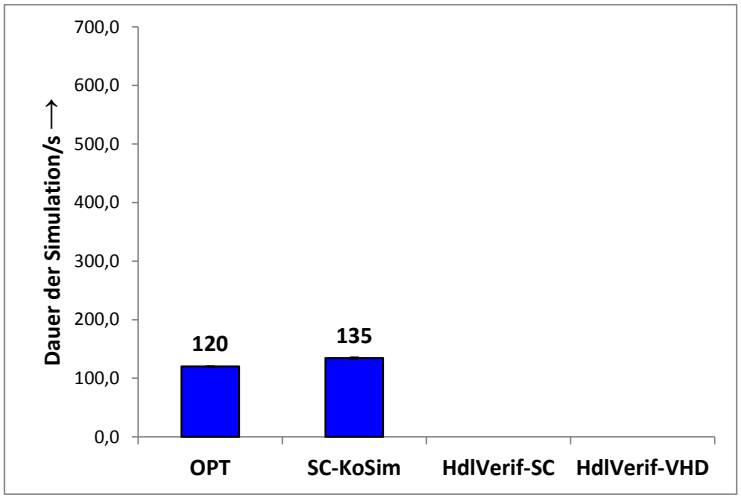

| | OPT | SC-KoSim | HdlVerif-SC | HdlVerif-VHD |
|---|---|---|---|---|
| Simulationsdauer | 137,2 s | 309,5 s | 319,9 s | 154,5 s |
| Standardabweichung | 0,730 s | 0,555 s | 0,730 s | 0,517 s |

Abbildung 8.10: Dauer der Simulation des GFB-Modells in Simulink-Umgebung mit Schrittweite 5.




| | OPT | SC-KoSim | HdlVerif-SC | HdlVerif-VHD |
|---|---|---|---|---|
| Simulationsdauer | 123,9 s | 153,5 s | 173,2 s | 21,3 s |
| Standardabweichung | 0,805 s | 1,108 s | 0,881 s | 0,359 s |

Abbildung 8.11: Dauer der Simulation des GFB-Modells in Simulink-Umgebung mit Schrittweite 50.




| | OPT | SC-KoSim | HdlVerif-SC | HdlVerif-VHD |
|---|---|---|---|---|
| Simulationsdauer | 120,1 s | 134,6 s | – | – |
| Standardabweichung | 0,427 s | 0,838 s | – | – |

Abbildung 8.12: Dauer der Simulation des GFB-Modells in Simulink-Umgebung mit variabler Schrittweite.

Die Dauer der Simulation mit variabler Schrittweite ist in Abbildung 8.12 dargestellt. Messungen der HdlVerif-SC- und HdlVerif-VHD-Variante konnten in diesem Fall nicht durchgeführt werden, da die HDL Verilator Toolbox die Verwendung variabler Schrittweiten nicht unterstützt. Im Vergleich mit der SC-KoSim-Variante zeigt sich, dass die OPT-Variante nach wie vor schneller ist, der Abstand ist jedoch erneut geringer geworden.

Wie im vorherigen Abschnitt, wurden auch bei den Simulationen mit Frequenzskalierung Aufzeichnungen des Ein-/Ausgabeverhaltens der HDL-Komponente gemacht, um sicherzustellen, dass ihr Verhalten in den einzelnen Varianten gleich ist. Auch in diesen Szenarien hat der Vergleich der Aufzeichnungen bei jeweils einer Schrittweite keine Unterschiede bei den einzelnen Varianten gezeigt.

Insgesamt lässt sich feststellen, dass der Vorteil der optimierten Integration der Hardwarekomponente in Simulink bei größeren Schrittweiten kleiner wird. Aufgrund der geringeren Anzahl an Synchronisationspunkten sinkt der relative Anteil des Mehraufwands für die Ko-Simulation am Gesamtaufwand. Da die optimierte Integration lediglich diesen Mehraufwand reduziert, sinkt auch das Optimierungspotenzial.

## 8.5 Validität und Simulatorkorrektheit

In Abschnitt 2.1 wurden die beiden Beziehungen Validität und Simulatorkorrektheit eingeführt, um eine Bewertung von Simulationsexperimenten bezüglich ihrer Aussagefähigkeit zu machen. Die Modellierungsbeziehung Validität sagt aus, ob ein Modell das zugehörige Quellsystem innerhalb des experimentellen Rahmens korrekt abbildet. Die Simulationsbeziehung Simulatorkorrektheit gibt an, ob ein Simulator das im Modell spezifizierte Ein-/Ausgabeverhalten korrekt wiedergibt. Da das transformierte Modell der Hardwarekomponente Modell und Simulator vereint, müssen beide Beziehungen geprüft werden.

Als Quellsystem kann das VHDL-Modell und die Ko-Simulation mit HDL-Verifier angesehen werden. Es stellt die Referenz für das Modellverhalten dar. Dabei werden industrielle Simulationswerkzeuge eingesetzt, bei denen Simulatorkorrektheit vorausgesetzt werden kann. Der experimentelle Rahmen der Simulation ist durch die Simulink-Umgebung definiert. Ein Modell ist demnach valide, wenn in Simulink kein Unterschied zum Quellsystem beobachtet werden kann.

Die Simulatorkorrektheit kann ebenfalls anhand der Simulink-Simulation gezeigt werden. Entspricht das Ein-/Ausgabeverhalten dem des Referenzsystems, so kann davon ausgegangen werden, dass das Verhalten korrekt wiedergegeben wird.

Zur Prüfung beider Beziehungen können folglich die Aufzeichnungen des Verhaltens der Hardwarekomponente in Simulink verwendet werden. Da deren Vergleich keinerlei Unterschiede zum Referenzsystem gezeigt hat, sind die Beziehungen Validität und Simulatorkorrektheit bezüglich des transformierten Modells vollständig erfüllt.

## 8.6 Zusammenfassung

Das in der vorliegenden Arbeit entwickelte Verfahren zur effizienten Integration der Modelle von Hardwarekomponenten in Simulink-Simulationen konnte mit unterschiedlichen Modellen durchgeführt werden und hat zu einer Verbesserung der Simulationsgeschwindigkeit geführt, ohne das sichtbare Verhalten der Hardwarekomponente zu beeinflussen.

Die in Kapitel 7 beschriebenen Schritte wurden mit vier unterschiedlichen Modellen durchgeführt und das Ergebnis in Simulink-Umgebungsmodellen simuliert. Ziel war dabei ein Vergleich der unterschiedlichen Methoden zur Integration von Hardwarekomponenten in Simulink. Neben der in dieser Arbeit entwickelten Methode wurden existierende Methoden zur Ko-Simulation verwendet. Die einzelnen Simulationsläufe wurden dann bezüglich des Modellverhaltens und der Simulationsgeschwindigkeit miteinander verglichen. Der Vergleich hat gezeigt, dass durch das optimierte Verfahren keinerlei Unterschiede im Modellverhalten auftreten. Die Simulationsgeschwindigkeit konnte allerdings deutlich erhöht werden. Es hat sich dabei gezeigt, dass die relative Verbesserung der Simulationsgeschwindigkeit stark von den einzelnen Modellen abhängt. Bei kleinen und mittleren Modellen konnte eine Beschleunigung um Faktor 1,8 bis Faktor 5 erreicht werden. Das große Modell konnte nur um etwa 25 % beschleunigt werden. In zusätzlichen Simulationsläufen ohne Simulink wurde gezeigt, dass das transformierte Modell der Hardwarekomponente und das im Vergleich verwendete SystemC-Modell etwa gleich schnell simuliert werden können. Die Geschwindigkeitssteigerung in Simulink ist somit tatsächlich auf die effizientere Integrationsmethode zurückzuführen und nicht auf die Modelltransformation. Diese Simulationsläufe haben außerdem gezeigt, dass die Simulation des VHDL-Modells deutlich schneller ist, weshalb die Ko-Simulation dieses Modells in Simulink nicht zur Bewertung herangezogen wurde.

Darüber hinaus wurde die Anwendbarkeit der Frequenzskalierung aus Kapitel 6 an einem Beispiel gezeigt. Dabei wurden unterschiedliche feste Skalierungsfaktoren sowie eine dynamische Skalierung mit variabler Schrittweite verwendet. In diesem Fall sind ebenfalls keine Unterschiede im Modellverhalten aufgetreten, und es konnte in allen Fällen eine höhere Simulationsgeschwindigkeit gemessen wer-

den. Erwartungsgemäß hängt die relative Beschleunigung der Simulation durch die effizientere Integrationsmethode stark von der gewählten Samplefrequenz ab. Eine niedrigere Frequenz führt zu weniger Synchronisationspunkten und damit zu weniger Mehraufwand für die Ko-Simulation.

Zusammenfassend lässt sich feststellen, dass das optimierte Verfahren in allen Fällen zu einer Steigerung der Simulationsgeschwindigkeit geführt hat, der Beschleunigungsfaktor hängt jedoch stark von den jeweiligen Modellen und der Schrittweite in Simulink ab.

# 9 Optimierung über stabile Zustände

Das bisherige Verfahren steigert die Simulationsgeschwindigkeit, indem das Zusammenspiel der beiden Modellteile – das Simulink-Modell und die Hardwarekomponente – optimiert und der Aufwand an der Schnittstelle zwischen diesen Teilen reduziert wird. Die dazu notwendige Transformation des Hardwaremodells ermöglicht einen weiteren Schritt zur Verbesserung der Simulationsgeschwindigkeit. Dieser erlaubt eine Reduktion des Aufwands zur Simulation der Hardwarekomponente selbst.

Die in Kapitel 7 beschriebene Transformation des Hardwaremodells führt dazu, dass der Zustand der Hardwaresimulation zu einem bestimmten logischen Zeitpunkt nur aus dem Zustand des Modells besteht, also aus dem Zustand seiner internen Signale und Variablen. Es gibt keinen Simulatorzustand. Ein gewöhnlicher HDL-Simulator hat Prozesslisten, Ereignislisten, eine Zeitverwaltung und weitere Elemente, die Teil des Simulationszustands sind. Dies fällt im transformierten Modell weg. Es gibt weder Prozesse noch Ereignisse und die Kontrolle der Zeit obliegt der umgebenden Simulink-Simulation. Diese Tatsache ermöglicht es, bei der Simulation der Hardwarekomponente Sprünge in der Zeit zuzulassen. Wenn sichergestellt werden kann, dass sich der interne Zustand der Hardwarekomponente bis zu einem bestimmten Zeitpunkt in der Zukunft nicht ändert, also ein temporär *stabiler Zustand* eingetreten ist, so kann die Simulation der Hardwarekomponente bis zu diesem Zeitpunkt einfach übersprungen werden. Da es keinen Simulatorzustand gibt und die Kontrolle der Zeit in Simulink stattfindet, ist dazu nichts weiter nötig, als die Simulation der Hardwarekomponente für die entsprechenden Zeitschritte einfach nicht auszuführen.

In der Praxis gibt es häufig Phasen, in denen eine Hardwarekomponente lediglich wartet. Sensoren werden zum Beispiel oft periodisch aktiviert, benötigen aber zum Erzeugen oder Verarbeiten eines neuen Werts nicht die gesamte Periode. Manche Signalverarbeitungskomponenten warten auf ein *TakeValue*-Signal bevor sie mit der Verarbeitung des dann anliegenden Wertes beginnen. Taktraten lassen sich nicht immer so frei wählen, dass eine hundertprozentige Auslastung einer Komponente gegeben ist. Solche Wartephasen müssen zwar in der Hardwarekomponente

auch korrekt implementiert sein, in der Simulation auf Systemebene liefern sie aber oft keinen relevanten Beitrag zum Simulationsergebnis.

Im folgenden Abschnitt wird die Grundidee der Optimierung über stabile Zustände zunächst an einem Beispiel erläutert, bevor seine Verallgemeinerung beschrieben wird. Im Anschluss daran werden die Auswirkungen auf die Simulationsgeschwindigkeit an zwei Beispielen untersucht.

## 9.1 Grundidee der stabilen Zustände

Die grundsätzliche Idee hinter der Optimierung über stabile Zustände liegt darin, dass logische Zeiträume übersprungen werden, in denen sich der Zustand der Hardwarekomponente nicht ändert oder lediglich kleine Veränderungen stattfinden, die in der Systemsimulation nicht relevant sind. Betrachten wir dazu das DSP_2-Beispiel aus Abschnitt 8.1.1. Es handelt sich bei der Hardwarekomponente um einen DSP, der über Interrupts gesteuert wird. Verschiedene Interrupts lösen die Abarbeitung unterschiedlicher Routinen des DSP-Programms aus. Ist die entsprechende Routine beendet, wartet der DSP auf den nächsten Interrupt. Dieser Ablauf ist schematisch in Abbildung 9.1 dargestellt.

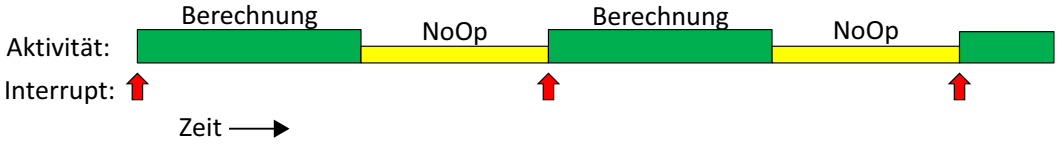



Abbildung 9.1: Schematischer Ablauf des DSP-Programms.

Die Dauer der Berechnungsblöcke muss dabei nicht immer gleich sein. Sie kann von der Art des Interrupts und den Eingangsdaten abhängen. Am Ende eines Berechnungsblocks springt das Programm in eine Warte-Schleife (No-Op), die erst verlassen wird, wenn der nächste Interrupt auftritt. Die Simulation der Warte-Schleife liefert keinerlei Ergebnisse, die in der System-Simulation von Interesse sind. Außerdem gibt es während der Warte-Schleife keine relevanten Änderungen am internen Zustand des DSP. Daher kann die Simulation der Warte-Schleife übersprungen und der mit ihr verbundene Berechnungsaufwand vermieden werden, ohne dass dadurch die Qualität der Gesamtsimulation beeinträchtigt wird.

Um diese Optimierung zu realisieren, müssen zunächst Beginn und Ende der Warte-Schleife aus Sicht des Modells definiert werden. Der Beginn der Warte-Schleife kann aufgrund einer bestimmten Sequenz von Operationen erkannt werden. Der DSP ist an eine externe ROM-Komponente angeschlossen, in der das

DSP-Programm gespeichert ist. Er hat einen Ausgang, der die ROM-Adresse der nächsten Operation festlegt, und einen Eingang, über den das Datenwort an dieser ROM-Adresse als Operationscode eingegeben wird. Eine Sequenz von vier bestimmten Operationscodes stellt die Implementierung der Warte-Schleife im DSP-Programm dar. Liegt diese Sequenz an vier aufeinanderfolgenden Taktschritten am ROM-Daten-Eingang des DSP an, so befindet sich das Programm in der Schleife. In der Struktur des DSP ist festgelegt, dass eine solche Schleife nur mit einem Interrupt verlassen werden kann. Das Ende der Warte-Schleife kann deshalb über den Interrupt-Eingang des DSP bestimmt werden. Befindet sich das DSP-Programm in der Schleife und ist dann der am Interrupt-Eingang anliegende Wert ungleich null, so ist das Ende der Schleife erreicht und die vollständige Simulation des DSP muss fortgesetzt werden.

Mit diesen Eigenschaften kann der Zeitraum der Warte-Schleife in der Simulation ermittelt werden und die Simulation des DSP kann für diesen Zeitraum ausgesetzt werden.

## 9.2 Verallgemeinerung über Garantie-Konzept

Nach der Einführung anhand eines Beispiels wird das Konzept nun allgemein beschrieben. Die Phase, die in der Simulation der Hardwarekomponente übersprungen werden kann, wird im Folgenden als *Stabiler Zustand* bezeichnet. Dieser Zustand, beziehungsweise die Eigenschaften, die ihn beschreiben, müssen explizit angegeben werden. Es ist vorstellbar, stabile Zustände in bestimmten Fällen auch automatisch zu ermitteln. Dies ist jedoch eine sehr komplexe Fragestellung, von der in dieser Arbeit abgesehen wird. Der Entwickler der Hardwarekomponente kennt in der Regel die charakteristischen Eigenschaften der stabilen Zustände, wie zum Beispiel die Instruktionen, die die Warte-Schleife im DSP implementieren. Oft können solche Eigenschaften auch anhand der Spezifikation oder des Datenblatts der Hardwarekomponente ermittelt werden. Die notwendigen Eigenschaften sind zum einen eine Eigenschaft, anhand der ein stabiler Zustand erkannt werden kann, und zum anderen eine Eigenschaft, die das Ende des stabilen Zustands beschreibt. Sie werden im Weiteren als *Zustandsbedingung* und *Abbruchbedingung* bezeichnet. Über diese Bedingungen können dann Verträge zwischen den beiden Teilen der Simulation formuliert werden, die vergleichbar sind mit den aus der verteilten Simulation bekannten Garantien. Der S-Function-Wrapper kann dann diese Garantien ausnutzen, um den Rechenaufwand für die Simulation der Hardwarekomponente zu reduzieren.

### 9.2.1 Spezifikation der Bedingungen

Die beiden Bedingungen müssen in Form von Ausdrücken formuliert sein, die zu einem Wahrheitswert evaluiert werden können. Ein einfaches Beispiel für eine Zustandsbedingung wäre eine von außen zugreifbare Variable im Modell, die einen bestimmten Wert annehmen muss: $model.state\_n == 1$. Eine Abbruchbedingung könnte auf dem Wert eines Eingangs beruhen: $input\_n! = 0$. Es können aber auch komplexere Ausdrücke und auch Folgen von Ausdrücken verwendet werden. Prinzipiell können alle Informationen verwendet werden, die im S-Function-Wrapper zur Verfügung stehen. Folgen von Ausdrücken können über die logische Zeit in der Simulink-Simulation oder aufeinanderfolgende Sampleschritte, aber bei Verwendung von Taktskalierung auch über aufeinanderfolgende Taktschritte oder eine Anzahl von Taktschritten realisiert werden.

#### Zustandsbedingung

Die Zustandsbedingung kann über die folgenden Eigenschaften spezifiziert werden:

**Modellzustand** Das Modell der Hardwarekomponente erreicht einen bestimmten Zustand, der durch den Wert eines Ausgangs oder einer Variable im Modell gekennzeichnet ist. Bei Variablen muss allerdings sichergestellt sein, dass diese von außen zugreifbar sind.

**Eingangszustand** Ein Eingang der Komponente hat einen bestimmten Wert.

**Anzahl von Sampleschritten** Eine bestimmte Anzahl von Sampleschritten, also Aktivierungen des S-Function-Blocks, wurde erreicht. Bei Verwendung von variablen Schrittweiten sollte diese Eigenschaft sehr vorsichtig verwendet werden.

**Anzahl von Taktereignissen** Eine bestimmte Anzahl von Taktereignissen, also Aufrufen der step-Funktion, wurde erreicht. Diese Eigenschaft kann insbesondere bei Skalierung der Taktfrequenz hilfreich sein.

**Zeiteigenschaften** Ein bestimmter logischer Zeitpunkt ist in der Simulation erreicht.

Diese Eigenschaften können beliebig über logische Verknüpfungen untereinander verbunden werden. Wie bereits erwähnt können darüber hinaus auch Folgen von Eigenschaften verwendet werden, wie zum Beispiel *Eingangszustand A und im nächsten Sampleschritt Eingangszustand B* oder *100 Taktereignisse nach Modellzustand C*.

#### Abbruchbedingung

Die Spezifikation der Abbruchbedingungen unterliegt einer Einschränkung. Der Zustand von Modell oder Ausgängen kann nicht herangezogen werden, da er sich im stabilen Zustand nicht verändert. Zusätzlich wird aber eine spezielle Variante der Bedingung über den Eingangszustand angeboten: Wertänderung an einem oder mehreren Eingängen.

**Eingangszustand** Ein Eingang der Komponente hat einen bestimmten Wert.

**Eingangsänderung** Wertänderung an einem oder mehreren bestimmten Eingängen oder an irgendeinem Eingang.

**Anzahl von Sampleschritten** Eine bestimmte Anzahl von Sampleschritten, also Aktivierungen des S-Function-Blocks, wurde erreicht. Bei Verwendung von variablen Schrittweiten sollte diese Eigenschaft sehr vorsichtig verwendet werden.

**Anzahl von Taktereignissen** Eine bestimmte Anzahl von Taktereignissen, also virtuellen Aufrufen der step-Funktion, wurde erreicht. Diese Eigenschaft kann insbesondere bei Skalierung der Taktfrequenz hilfreich sein.

**Zeiteigenschaften** Ein bestimmter logischer Zeitpunkt ist in der Simulation erreicht.

In dieser Bedingung können ebenfalls Verknüpfungen und Folgen angegeben werden. Folgen und Anzahlen von Sampleschritten, beziehungsweise Taktereignissen, sollten jedoch frühestens mit Eintreten des stabilen Zustands beginnen. Ist dies nicht der Fall, so steigt der Aufwand deutlich, da Teile der Abbruchbedingung auch außerhalb der stabilen Zustände geprüft werden müssen.

### 9.2.2 Garantien

Mit diesen Bedingungen können nun Verträge ähnlich den in Abschnitt 3.4.1 vorgestellten Garantien aus der verteilten Simulation formuliert werden. Die Gesamtsimulation besteht aus drei logischen Teilen, der Simulink-Umgebung, der Hardwarekomponente und dem S-Function-Wrapper, der als Scheduler für die Simulation der Hardwarekomponente aufgefasst werden kann. Die Garantien werden nicht zwischen beliebigen Teilen ausgetauscht, sondern nur von der Simulink-Umgebung an den S-Function-Wrapper und von der Hardwarekomponente ebenfalls an den S-Function-Wrapper. Diese beiden Arten von Garantien werden unterschieden als *implizite Garantie* von Simulink und *explizite Garantien* seitens der Hardwarekomponente.

#### Implizite Garantie von Simulink

Diese Garantie muss nicht vom Entwickler spezifiziert werden. Sie ergibt sich direkt aus dem zeitgetriebenen Simulationskonzept von Simulink. Formal lautet sie:

Seien $f : T \longrightarrow X$ die Trajektorie über alle Eingänge der Hardwarekomponente und $t_n$ und $t_{n+1}$ zwei aufeinanderfolgende Samplezeitpunkte, so gilt:

$$\forall t \in [t_n, t_{n+1}) : f(t) = f(t_n) \tag{9.1}$$

Das heißt, die Simulink-Umgebung garantiert gegenüber dem Scheduler, dass Wertänderungen an den Eingängen der Hardwarekomponente nur an Samplezeitpunkten stattfinden können. Zwischen den Samplezeitpunkten ändern sich die Werte nicht.

#### Explizite Garantie der Hardwarekomponente

Die Garantien seitens der Hardwarekomponente müssen vom Entwickler explizit über zuvor genannten Eigenschaften in Form von Zustandsbedingung und Abbruchbedingung angegeben werden. Die so formulierte Garantie lautet: Befindet sich das Modell der Hardwarekomponente in Zustand SZ, erkennbar durch Zustandsbedingung ZB, dann geschehen in der Simulation der Hardwarekomponente keine relevanten Zustandsänderungen, solange nicht Abbruchbedingung AB eingetreten ist. Sobald Zustand SZ eingetreten ist, kann die Simulation der Hardwarekomponente ausgesetzt werden, bis die Abbruchbedingung eintritt.

Es ist ohne Weiteres möglich, auch mehrere explizite Garantien für eine Hardwarekomponente anzugeben. Dazu sollte lediglich sichergestellt sein, dass die Zustandsbedingungen eindeutig sind und dass unterschiedliche stabile Zustände sich nicht gegenseitig überlappen.

## 9.3 S-Function-Wrapper mit Stabilen Zuständen

Dieses grundsätzliche Konzept der Optimierung über stabile Zustände muss nun in den Simulator übertragen werden. Dazu wird im Folgenden die Realisierung eines Simulationsschritts im S-Function-Wrapper beschrieben. Abbildung 9.2 zeigt zunächst den Ablauf im einfachen S-Function-Wrapper ohne Skalierung der Taktfrequenz. Nachdem die Eingangswerte gelesen wurden, prüft der Wrapper zunächst, ob die Abbruchbedingung erfüllt ist. Ist dies der Fall, so wird das Flag zur Markierung des stabilen Zustands zurückgenommen. Danach wird geprüft, ob sich das

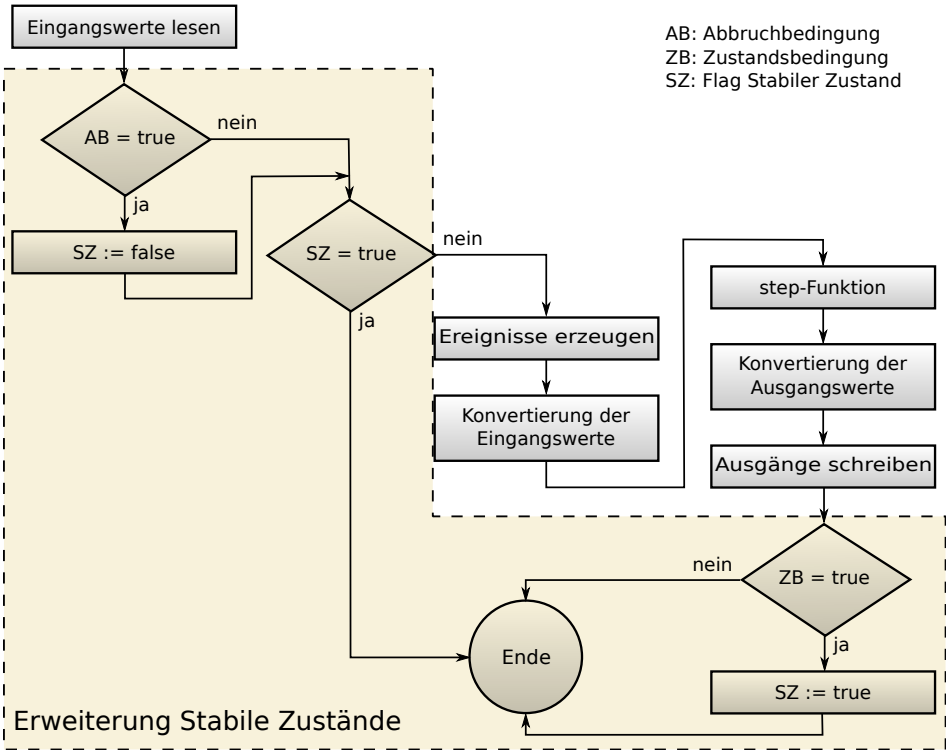



Abbildung 9.2: S-Function-Wrappers mit Erweiterung Stabile Zustände.

Modell in einem stabilen Zustand befindet. Wenn das zutrifft, dann kann sofort zum Ende gesprungen werden. Trifft dies nicht zu, dann wird die Ausführung zunächst wie gewohnt mit einem Simulationsschritt der Hardwarekomponente fortgesetzt. Nach dem Schreiben der Ausgangswerte folgt ein weiterer neuer Schritt, die Überprüfung der Zustandsbedingung. Ist die Zustandsbedingung erfüllt, so wird das Flag zur Markierung des stabilen Zustands auf *true* gesetzt, bevor der Simulationsschritt beendet ist.

Mit Skalierung der Taktfrequenz ist der Ablauf prinzipiell gleich. Die Erweiterung muss in diesem Fall lediglich an zwei Stellen eingefügt werden, bei der Verarbeitung der Taktereignisse vor dem Samplezeitpunkt und bei der Verarbeitung des potentiellen Taktereignisses am Samplezeitpunkt (Abbildung 9.3). Die Überprüfung der Abbruchbedingung und des Stabiler-Zustand-Flag geschieht vor jedem Aufruf der step-Funktion. Die Überprüfung, ob ein stabiler Zustand erreicht wurde, geschieht nach der step-Funktion, der Invertierung der Taktvariablen und dem Voranschreiten der Hardwaresimulationszeit. Eine besondere Situation findet sich am Ende des Simulationsschritts. Dort wird vor der Konvertierung und dem Schreiben der Ausgangswerte zunächst geprüft, ob die step-Funktion im Verlauf

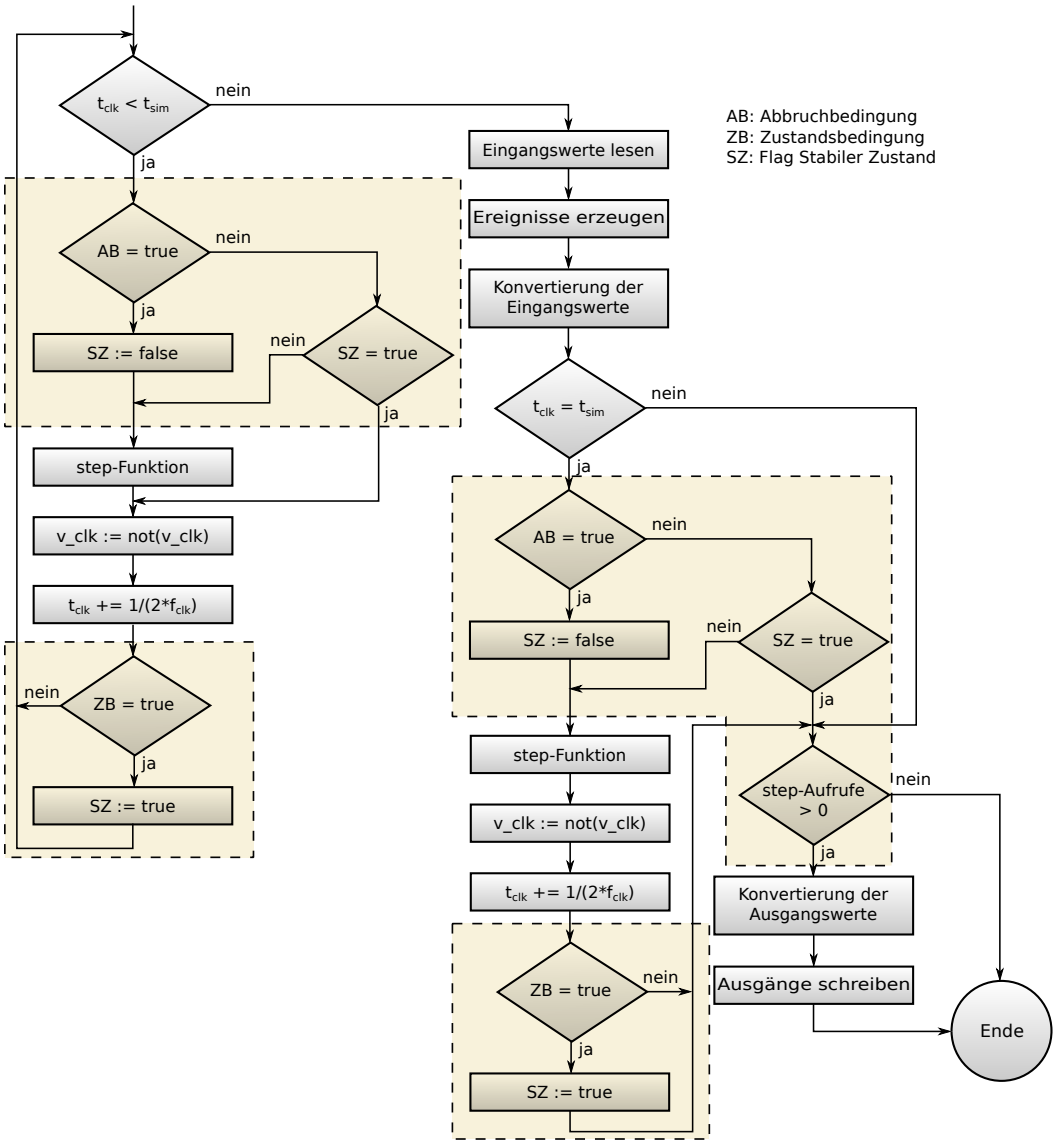



Abbildung 9.3: S-Function-Wrappers mit Erweiterung Stabile Zustände und Frequenz-Skalierung.

des Simulationsschritts mindestens einmal aufgerufen wurde. Gab es keinen Aufruf der step-Funktion, so können sich die Ausgangswerte nicht geändert haben. Demzufolge müssen sie auch nicht erneut konvertiert und geschrieben werden.

In vielen Fällen ist die Abbruchbedingung über bestimmte Eingangswerte definiert. Im DSP-Beispiel aus Abschnitt 9.1 ist es der Wert des Interrupt-Eingangs. Ist die Abbruchbedingung nur von Eingangswerten abhängig und nicht von Zeiteigenschaften oder der Taktanzahl, so kann der Ablauf im S-Function-Wrapper noch weiter optimiert werden. Die implizite Garantie von Simulink besagt, dass sich Eingangswerte nur zu Samplezeitpunkten ändern können. Deshalb kann eine Bedingung, die nur von diesen abhängt, ihren Wahrheitswert auch nur zu Samplezeitpunkten ändern. Der modifizierte Ablauf für diesen Fall ist in Abbildung 9.4 dargestellt. Befindet sich das Modell der Hardwarekomponente zu Beginn des Simulationsschritts in einem stabilen Zustand, so kann die Schleife zur Verarbeitung der Taktereignisse seit dem letzten Samplezeitpunkt komplett übersprungen werden. Wechselt die Komponente im Verlauf dieser Schleife in einen stabilen Zustand, so kann der verbleibende Teil der Schleife übersprungen werden. In diesen Fällen muss lediglich der Wert von $t_{clk}$ auf den Zeitpunkt des nächsten Taktereignisses, das nicht vor der aktuellen Zeit in der Simulink-Simulation $t_{sim}$ liegt, gesetzt werden. Wird eine ungerade Anzahl von Taktereignissen übersprungen, so muss zusätzlich der Wert der Taktvariablen invertiert werden, damit dieser bei der Verarbeitung des nächsten Taktereignisses korrekt ist. Dann werden die Eingangswerte gelesen. Ist nun die Abbruchbedingung nicht erfüllt und die Komponente befindet sich in einem stabilen Zustand, dann kann zusätzlich zur step-Funktion auch das Erzeugen der Eingangsereignisse und die Konvertierung der Eingangswerte übersprungen werden. Da sich vor dem nächsten Lesen der Eingangswerte der Wert des Stabiler-Zustand-Flag nicht mehr ändern kann, ist sichergestellt, dass bis dahin auch kein Aufruf der step-Funktion stattfinden muss. Deshalb werden die Eingangswerte nicht benötigt, müssen folglich auch nicht konvertiert werden. Abschließend wird auch hier geprüft, ob mindestens ein Aufruf der step-Funktion stattgefunden hat. Nur dann werden die Werte der Ausgänge konvertiert und auf die Ausgangsports geschrieben.

## 9.4 Einfluss auf die Simulationsgeschwindigkeit

Der optimierte Ablauf und das Ausnutzen von stabilen Zuständen soll die Geschwindigkeit der Gesamtsimulation erhöhen. Bevor dieser Einfluss anhand von Messungen betrachtet wird, folgt zunächst eine Erörterung der wesentlichen Merkmale der Erweiterung, die zu einer Veränderung der Simulationsgeschwindigkeit beitragen. Positiv, im Sinne einer Beschleunigung der Simulation, wirkt sich in

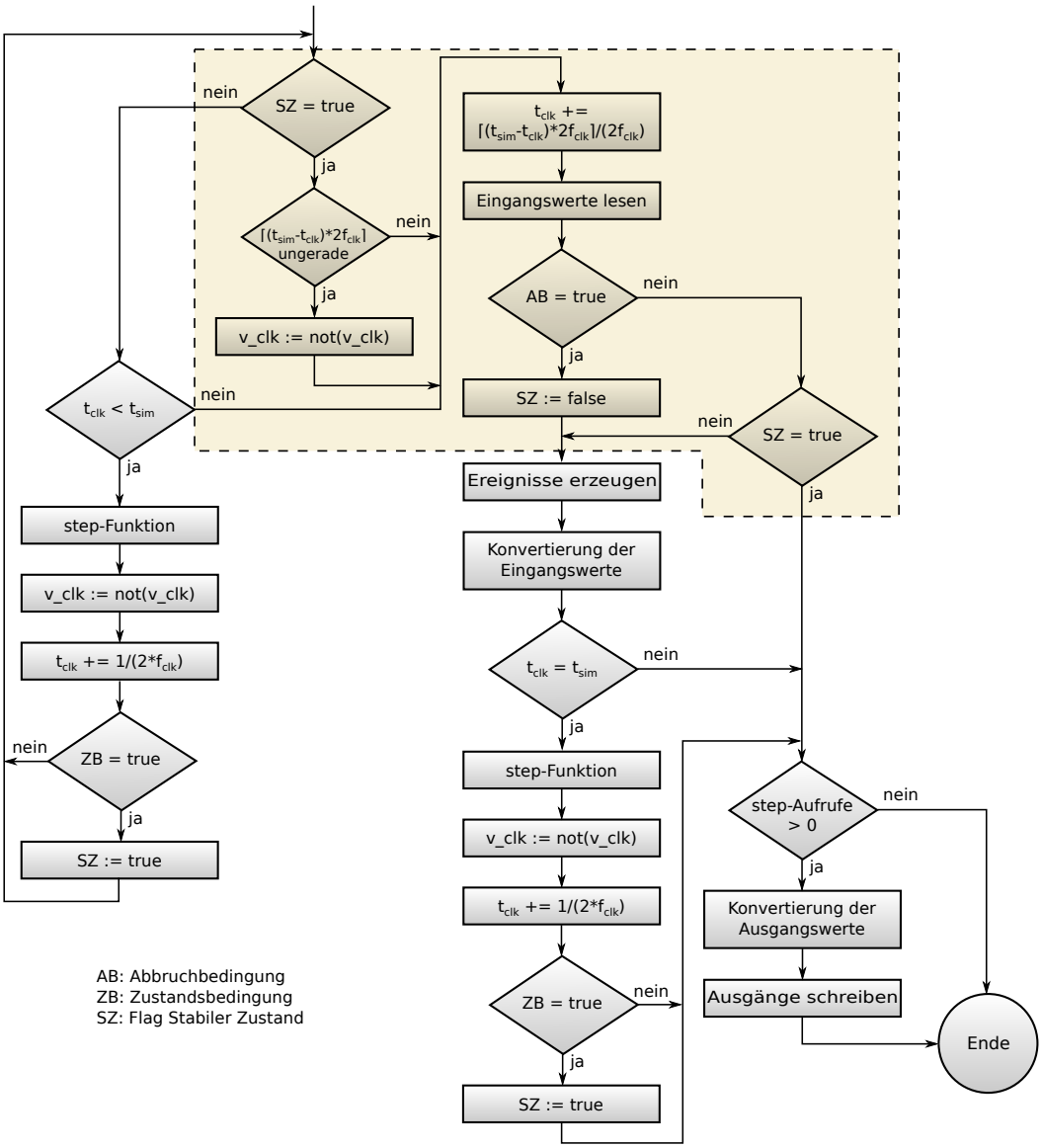



Abbildung 9.4: S-Function-Wrappers mit Abbruchbedingung, die nur von Eingängen abhängig ist.

erster Linie das Überspringen von Aufrufen der step-Funktion und gegebenenfalls auch Konversionen von Ein- und Ausgabedaten aus. Dem gegenüber steht der zusätzliche Aufwand zur Prüfung der Bedingungen und der etwas komplexere Kontrollfluss im S-Function-Wrapper.

Der Mehraufwand hängt wesentlich von der Komplexität der Zustands- und Abbruchbedingungen ab. Die übrigen zusätzlich benötigten Operationen beschränken sich auf das Abfragen und Setzen wahrheitswertiger Variablen zur Steuerung des Kontrollfluss sowie im Falle des Ablaufs aus Abbildung 9.4 einfache numerische Operationen zur Berechnung von $t_{clk}$. Der Aufwand für die Ausführung derartiger Operationen ist in modernen Computerarchitekturen sehr gering und kann vernachlässigt werden. Der Aufwand zur Prüfung der Zustands- und Abbruchbedingung hängt von deren Formulierung ab. Hängt eine Abbruchbedingung zum Beispiel nur von Eingangswerten ab, so reduziert sich die Prüfung der Bedingung auf den Vergleich von numerischen Daten. Der Aufwand dafür ist ebenfalls sehr gering. Ist andererseits eine Zustandsbedingung von komplexen Zusammenhängen im Modell der Hardwarekomponente abhängig, kann der Aufwand deutlich größer ausfallen.

Die Einsparung von Rechenaufwand resultiert aus dem Überspringen von Aufrufen der step-Funktion und Konversionen der Ein- und Ausgabedaten. Bei der Geschwindigkeit der Gesamtsimulation spielt natürlich die Anzahl und Länge der Phasen, in denen sich die Hardwarekomponente in einem stabilen Zustand befindet, das heißt die Häufigkeit, mit der die jeweiligen Schritte übersprungen werden, eine große Rolle. Dazu können keine allgemeinen Aussagen getroffen werden. Sie hängt individuell von der Hardwarekomponente und dem System, in dem diese integriert ist, ab. Ein weiterer Punkt ist der Aufwand für einen einzelnen Schritt. Bei der Konversion von Ein- und Ausgabedaten ist dieser Aufwand im Wesentlichen durch den Umfang der Schnittstelle, also Anzahl und Bitbreite von Einbeziehungsweise Ausgangsports, bestimmt. Das Einsparpotenzial beim Überspringen der step-Funktion ist ebenfalls individuell abhängig von der Implementierung der jeweiligen Hardwarekomponente. Gibt es in einem stabilen Zustand tatsächlich keine Aktivität und es wird nur auf ein bestimmtes Ereignis gewartet, so beschränkt sich die Einsparung auf den Aufwand für den Kontextwechsel in die step-Funktion und die Übergabe der Parameter. Die stabilen Zustände sind jedoch nicht auf Zustände *ohne jegliche* Aktivität beschränkt, sondern umfassen auch solche, die keine *in der Systemsimulation relevanten* Aktivitäten beinhalten. Gibt es in einem stabilen Zustand Aktivität im Inneren der Komponente, kann entsprechend mehr Aufwand durch das Überspringen des Funktionsaufrufs eingespart werden.

Zusammenfassend lässt sich sagen, dass die Beschleunigung, die durch die Optimierung über stabile Zustände erreicht wird, sehr stark vom jeweiligen System abhängig ist. Sie wird durch die Häufigkeit und Dauer der stabilen Zustände be-

stimmt. Der Mehraufwand für die zusätzlich benötigten Kontrollstrukturen zur Steuerung der Optimierung ist äußerst gering, solange die zur prüfenden Bedingungen nicht zu komplex sind. Insgesamt ist daher nur in seltenen Ausnahmefällen ein negativer Einfluss auf die Geschwindigkeit der Gesamtsimulation zu erwarten. Wie groß die zu erwartende Beschleunigung ist, kann jedoch nicht ohne Wissen über das zu simulierende System bestimmt werden.

## 9.5 Evaluation der Simulationsgeschwindigkeit

Um eine Bewertung der vorgeschlagenen Optimierung vornehmen zu können, wird diese nun in zwei Beispielen realisiert. Dabei handelt es sich zum einen um das DSP_2-Beispiel aus Abschnitt 8.1.1, das auch zur Beschreibung der Grundidee in diesem Kapitel verwendet wurde. Als zweites Beispiel wird das GFB-Beispiel (Abschnitt 8.1.1) verwendet. In diesem Beispiel kann die Optimierung auch in Kombination mit der Skalierung der Taktfrequenz und variablen Schrittweiten betrachtet werden.

Szenario und Messmethode entsprechen bei diesen Untersuchungen der Beschreibung in Abschnitt 8.1. Zusätzlich zu den dort beschriebenen und bereits untersuchten Varianten, wird nun die Variante *OPT+SZ* betrachtet, die die Optimierung über stabile Zustände enthält.

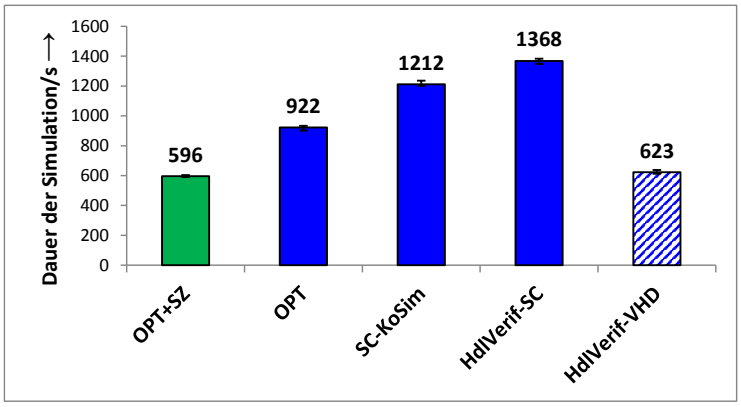
### 9.5.1 DSP_2

Wie bereits in Abschnitt 9.1 beschrieben, ist das DSP-Programm über Interrupts gesteuert. Ein Interrupt initiiert eine bestimmte Berechnungssequenz, und nach dieser Sequenz wartet das Programm auf den nächsten Interrupt. Das Warten ist durch eine Schleife über vier bestimmte Instruktionen implementiert. Mit diesen Informationen und dem Wissen, dass der DSP bei jeder positiven Taktflanke eine neue Instruktion vom *rom_data_in*-Eingang übernimmt, können die Bedingungen für die Stabile-Zustände-Optimierung formuliert werden.

Die Abbruchbedingung ist äußerst einfach: der stabile Zustand endet, wenn am Interrupt-Eingang ein Wert ungleich null anliegt.

Die Zustandsbedingung lautet: Ein stabiler Zustand ist erreicht, wenn bei vier aufeinanderfolgenden positiven Taktereignissen die Instruktionen I1, I2, I3 und I4 in dieser Reihenfolge an Eingang rom_data_in anliegen.

Abbildung 9.5 zeigt die Simulationsdauer des DSP_2-Modells. Bei der Variante OPT+SZ ist gegenüber den anderen Varianten nochmals eine deutliche Beschleu-




| | OPT+SZ | OPT | SC-KoSim | HdlVerif-SC | HdlVerif-VHD |
|---|---|---|---|---|---|
| Sim.-dauer | 596,2 s | 922,3 s | 1211,8 s | 1367,9 s | 622,7 s |
| Std.-abw. | 4,79 s | 12,30 s | 14,08 s | 15,20 s | 10,84 s |

Abbildung 9.5: Dauer der Simulation des DSP_2-Modells in Simulink-Umgebung.

nigung zu sehen. Gegenüber der OPT-Variante ist die Simulationsdauer um etwa ein Drittel geringer geworden. Verglichen mit der SC-KoSim-Variante wird sogar nur noch etwa die Hälfte der Zeit für die Simulation benötigt. Auch bei diesen Simulationsläufen wurden Aufzeichnungen des Ein-/Ausgabeverhaltens vorgenommen und mit den Aufzeichnungen der Referenzsimulation verglichen. In diesem Fall sind Unterschiede in den Phasen der stabilen Zustände aufgetreten, die jedoch zu erwarten waren. In der Referenzsimulation wird die Warte-Schleife aktiv im DSP ausgeführt. Darum gibt es auch in dieser Phase Aktivitäten am Ausgang für die ROM-Adresse und am Eingang für die ROM-Daten. Da die Simulation der Schleife in der OPT+SZ-Variante übersprungen wird, sind diese Aktivitäten auch in der Aufzeichnung nicht vorhanden. Außer diesen erwarteten Abweichungen in den Phasen der stabilen Zustände sind keinerlei Unterschiede aufgetreten. Damit kann festgestellt werden, dass das Ein-/Ausgabeverhalten im Rahmen der Erwartungen nach wie vor korrekt ist.

### 9.5.2 GFB

Im GFB-Beispiel ergibt sich der stabile Zustand aus der Tatsache, dass die Verarbeitung eines neuen Eingangswerts abgeschlossen ist, bevor der nächste Eingangswert übernommen werden soll. Ein neuer Wert wird bei einer positiven Flanke am Take-Eingang übernommen. Danach dauert es 1940 Takte, bis dieser Wert vollstän-

dig verarbeitet ist. Diese Anzahl ergibt sich aus 20 Takten für das initiale Übernehmen des Wertes und 32 Takten für die Berechnung eines Ausgangswerts. Bei zwei Ausgängen pro Kanal und 30 Kanälen ergibt sich so die Anzahl von

$$2 * 30 * 32\,\text{Takte} + 20\,\text{Takte} = 1940\,\text{Takte} \tag{9.2}$$

Takten.

Die Abtastrate für Eingangswerte liegt bei 20 kHz, die Taktrate für die Hardwarekomponente bei 50 MHz. Somit stehen für die Verarbeitung eines Eingangswerts 2500 Taktschritte zu Verfügung. Da davon nur 1940 benötigt werden, verbleiben 560 Taktschritte, in denen die Hardwarekomponente auf die nächste positive Flanke am Take-Eingang wartet. Für diesen Zeitraum kann die Simulation der Hardwarekomponente übersprungen werden.

Die Bedingungen für den stabilen Zustand lauten daher wie folgt:

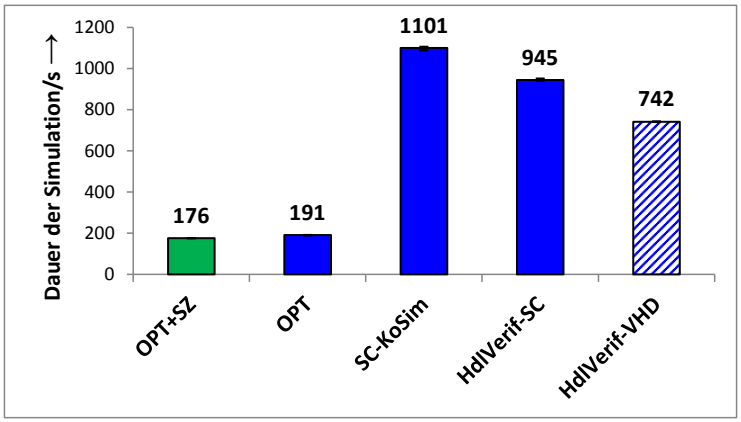
**Abbruchbedingung:** Der stabile Zustand endet, wenn am Eingang *TakeIIF* eine positive Flanke eintritt.

**Zustandsbedingung:** Der stabile Zustand tritt 1940 Taktzyklen nach dem Auftreten der positiven Flanke am *TakeIIF*-Eingang ein.

Damit kann die Optimierung in den S-Function-Wrapper integriert werden. Da in diesem Beispiel das Taktsignal aufgrund der Frequenzskalierung im Wrapper erzeugt wird, ist das Abzählen der Taktzyklen einfach möglich. Die Erkennung der Flanke am Take-Eingang ist ebenfalls unkompliziert.

Das Ergebnis der Geschwindigkeitsmessung ohne tatsächliche Frequenzskalierung, also mit einer Samplerate, die der doppelten Taktrate entspricht, ist in Abbildung 9.6 dargestellt. Die Simulation mit der Optimierung über stabile Zustände (OPT+SZ) ist schneller als die Simulation in der OPT-Variante. Der Gewinn ist jedoch gering. In diesem Beispiel gibt es in der Hardwarekomponente im stabilen Zustand tatsächlich keinerlei Aktivität. Es wird lediglich bei jeder positiven Taktflanke der Wert des Take-Eingangs überprüft. Da somit der Aufwand für die Simulation des stabilen Zustands nicht groß ist, ist auch der Gewinn, der durch das Überspringen der Simulation erreicht werden kann, gering. Dennoch ist zu erkennen, dass die Optimierung funktioniert und zumindest zu einer kleinen Beschleunigung führt.

In den folgenden Diagrammen ist die Dauer der Simulation mit Frequenzskalierung aufgezeigt. Die Schrittweite 1 entspricht dem Abstand der Taktereignisse. Abbildung 9.7 zeigt die Dauer der Simulation mit Schrittweite 2, womit die Samplerate gleich der einfachen Taktrate ist. In Abbildung 9.8 ist die Dauer der Simulation mit Schrittweite 5 dargestellt, und die Werte in Abbildung 9.9 entstammen




| | OPT+SZ | OPT | SC-KoSim | HdlVerif-SC | HdlVerif-VHD |
|---|---|---|---|---|---|
| Sim.-dauer | 175,9 s | 190,6 s | 1100,6 s | 944,7 s | 741,6 s |
| Std.-abw. | 0,959 s | 0,664 s | 5,211 s | 3,805 s | 1,071 s |

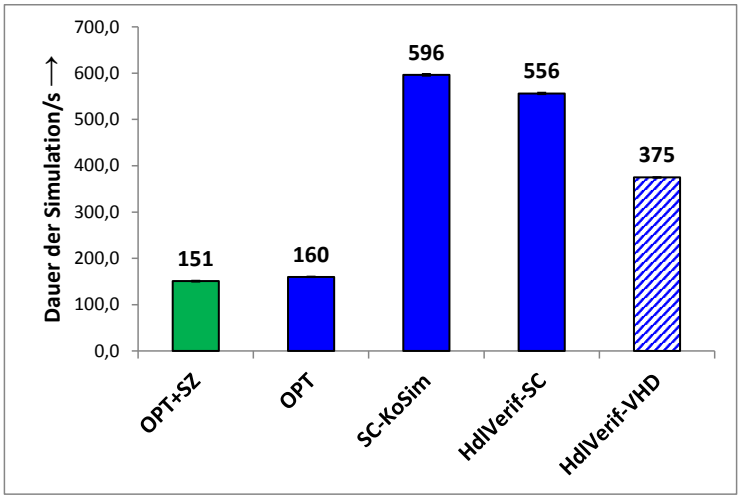
Abbildung 9.6: Dauer der Simulation des GFB-Modells in Simulink-Umgebung.

der Simulation mit Schrittweite 50. Abbildung 9.10 zeigt die ermittelten Ergebnisse bei Verwendung einer variablen Schrittweite. In allen Fällen ist gegenüber der OPT-Variante eine um etwa 9 Sekunden geringere Simulationsdauer zu sehen. Dies entspricht den Erwartungen, da der eingesparte Rechenaufwand in allen Fällen gleich ist: 1120 Aufrufe der step-Funktion pro Eingangswert (560 positive und 560 negative Taktflanken).

Bei diesem Beispiel wurden ebenfalls für alle getesteten Schrittweiten Aufzeichnungen des Ein-/Ausgabenverhaltens erstellt. Die Vergleiche mit den Aufzeichnungen der Referenzsimulationen haben keinerlei Unterschiede gezeigt.

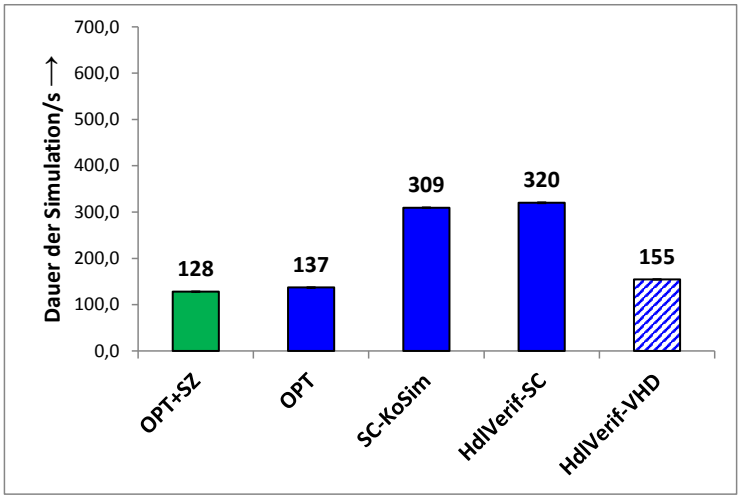
## 9.6 Zusammenfassung Stabile Zustände

Die Optimierung über stabile Zustände basiert darauf, dass bestimmte Phasen in der Simulation der Hardwarekomponente übersprungen werden können, wenn in diesen Phasen gar keine oder keine in der Systemsimulation relevanten Aktivitäten stattfinden. Das vollständige Überspringen von Simulationsschritten wurde erst möglich, weil der Zustand in der Simulation der Hardwarekomponente allein durch den internen Zustand der Komponente selbst bestimmt ist. Es gibt keinen Simulatorzustand der angepasst werden muss und die Verwaltung der Simulationszeit findet vollständig in Simulink und im S-Function-Wrapper statt.



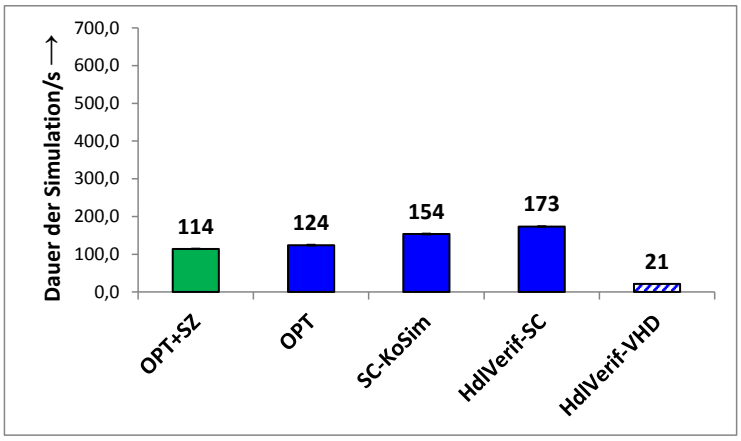

| | OPT+SZ | OPT | SC-KoSim | HdlVerif-SC | HdlVerif-VHD |
|---|---|---|---|---|---|
| Sim.-dauer | 150,9 s | 159,9 s | 596,1 s | 556,0 s | 374,9 s |
| Std.-abw. | 0,849 s | 0,540 s | 1,414 s | 1,084 s | 0,573 s |

Abbildung 9.7: Dauer der Simulation des GFB-Modells in Simulink-Umgebung mit Schrittweite 2.



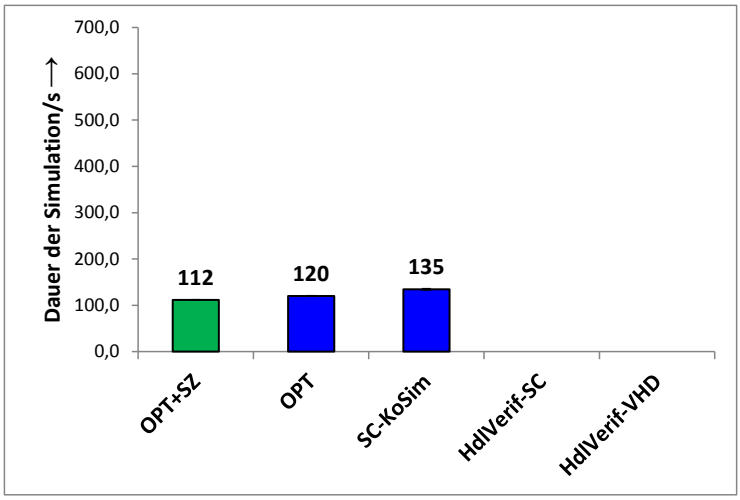

| | OPT+SZ | OPT | SC-KoSim | HdlVerif-SC | HdlVerif-VHD |
|---|---|---|---|---|---|
| Sim.-dauer | 128,1 s | 137,2 s | 309,5 s | 319,9 s | 154,5 s |
| Std.-abw. | 0,629 s | 0,730 s | 0,555 s | 0,730 s | 0,517 s |

Abbildung 9.8: Dauer der Simulation des GFB-Modells in Simulink-Umgebung mit Schrittweite 5.




| | OPT+SZ | OPT | SC-KoSim | HdlVerif-SC | HdlVerif-VHD |
|---|---|---|---|---|---|
| Sim.-dauer | 113,9 s | 123,9 s | 153,5 s | 173,2 s | 21,3 s |
| Std.-abw. | 0,629 s | 0,805 s | 1,108 s | 0,881 s | 0,359 s |

Abbildung 9.9: Dauer der Simulation des GFB-Modells in Simulink-Umgebung mit Schrittweite 50.




| | OPT+SZ | OPT | SC-KoSim | HdlVerif-SC | HdlVerif-VHD |
|---|---|---|---|---|---|
| Sim.-dauer | 111,6 s | 120,1 s | 134,6 s | – | – |
| Std.-abw. | 0,412 s | 0,427 s | 0,838 s | – | – |

Abbildung 9.10: Dauer der Simulation des GFB-Modells in Simulink-Umgebung mit variabler Schrittweite.

Zur Detektion der *stabile Zustände* genannten Phasen, die übersprungen werden können, muss der Entwickler zwei Bedingungen angeben. Die Zustandsbedingung definiert den stabilen Zustand und die Abbruchbedingung definiert sein Ende. Diese Bedingungen können dann im S-Function-Wrapper integriert werden, um die Simulation entsprechend zu steuern. Das Konzept konnte erfolgreich an zwei Beispielen ohne und mit Skalierung der Taktfrequenz angewendet werden. In der Simulation sind dabei keine Unterschiede im Ein-/Ausgabeverhalten der Hardwarekomponente aufgetreten, bis auf kleine Abweichungen während des stabilen Zustands im DSP_2-Beispiel, die allerdings absehbar und eingeplant waren. Der durch die Optimierung erreichbare Gewinn an Simulationsgeschwindigkeit ist stark abhängig vom Modell der Hardwarekomponente und vom Gesamtsystem, in dem es eingesetzt wird. Wesentlichen Einfluss darauf haben die Dauer und Häufigkeit der Phasen in stabilen Zuständen und die Komplexität der übersprungenen Simulationsschritte. Der Mehraufwand für die Kontrolle der stabilen Zustände ist jedoch äußerst gering, so dass auch bei ungünstigen Fällen nicht mit einer Verlangsamung der Gesamtsimulation zu rechnen ist.

# 10 Zusammenfassung und Ausblick

## 10.1 Zusammenfassung und Bewertung der Evaluationsergebnisse

Ziel dieser Arbeit ist es, eine Methode zu entwickeln, um HDL-Beschreibungen von Hardwarekomponenten effizient in das Simulink-Modell eines Gesamtsystems zu integrieren. Dazu wurden in Abschnitt 1.2 anhand des gegebenen Anwendungsszenarios Voraussetzungen und Anforderungen definiert. Die in den vorangegangenen Kapiteln vorgestellte Lösung soll nun bezüglich dieser Kriterien untersucht und bewertet werden.

Die vorgestellte Methode basiert auf einer Transformation der HDL-Komponente in ein Simulationsmodell, das effizienter in Simulink integriert werden kann und somit zu einer schnelleren Gesamtsimulation führt. Das in Simulink beobachtete Verhalten der HDL-Komponente bleibt dabei exakt erhalten.

Die Evaluation anhand von Beispielmodellen hat gezeigt, dass in allen Fällen in der Gesamtsimulation eine Beschleunigung erreicht werden konnte. Aufgrund des reduzierten Mehraufwands an der Schnittstelle zwischen Simulink-Modell und Hardwarekomponente ist eine schnellere Simulation möglich als mit herkömmlichen Methoden zur Ko-Simulation. Die relative Beschleunigung hängt jedoch von einigen Eigenschaften der Modelle ab. Bei kleinen und mittelgroßen Modellen konnte eine Beschleunigung um Faktor 1,8 bis Faktor 5 gemessen werden. Das größte der untersuchten Modelle konnte lediglich um 25 % beschleunigt werden. Da aber Simulationsläufe gerade bei großen Modellen oft sehr lange dauern, ist auch eine Reduktion der Simulationsdauer um ein Viertel ein großer Gewinn. Wird die Schrittweite in Simulink vergrößert und die Skalierung zwischen Samplerate und Hardware-Taktrate verwendet, so ist in den Simulationen mit herkömmlicher Ko-Simulation eine deutlich schnellere Simulation möglich. In der optimierten Variante fällt die Beschleunigung durch eine größere Schrittweite geringer aus. Der zusätzliche Rechenaufwand für die Ko-Simulation fällt bei jedem Synchronisationspunkt an. Da dieser Aufwand in der optimierten Variante äußerst gering ist, ist

der Vorteil, der sich durch eine geringere Anzahl an Synchronisationspunkten ergibt, ebenfalls gering. Letztlich ist aber auch hier die Simulation der optimierten Variante in allen Fällen schneller als die anderen Varianten. Die Forderung nach einer hohen Simulationsgeschwindigkeit kann folglich als erfüllt betrachtet werden.

Die zweite Anforderung ist die Automatisierbarkeit. Die Methode soll ohne manuelle Schritte anwendbar sein. Für wesentliche Teile der Methode wurden bereits Transformationsregeln implementiert, die im Rahmen der Evaluation bereits automatisiert verwendet wurden. Für die verbleibenden Schritte wurden Vorgehensweisen beschrieben, wie diese in Form von Transformationsregeln umgesetzt werden können. Diese Anforderung ist damit ebenfalls erfüllt.

Die Äquivalenz der Simulation des transformierten Modells mit der Simulation des ursprünglichen Modells wurde zunächst theoretisch anhand der Formalismen DTSS und DEVS gezeigt. Zusätzlich dazu wurde das Verhalten der Hardwarekomponente anhand von Aufzeichnungen ihrer Ein- und Ausgangswerte mit den Aufzeichnungen einer Referenzsimulation verglichen. Dabei konnte in keinem Simulationsschritt in Simulink eine Abweichung festgestellt werden. Damit ist gezeigt, dass das Verhalten der Hardwarekomponente in der Systemsimulation exakt wiedergegeben wird, sowohl zeitlich als auch funktional.

Bezüglich der Hardwaremodelle ist gefordert, dass beliebige synthetisierbare HDL-Modelle unterstützt werden sollen. Dies ist der Fall. Die einzige grundsätzliche Einschränkung bezüglich der Hardwarebeschreibungssprache ist, dass das Modell keine relevanten Zeitannotationen beinhalten darf. Dies gilt jedoch genauso für synthetisierbare Modelle. Liegt das Modell in VHDL oder Verilog vor, so können weitere Einschränkungen durch das verwendete Werkzeug zur Transformation nach C++ entstehen. Das synthetisierbare Subset der Sprachen ist in diesen Werkzeugen aber in der Regel vollständig unterstützt.

Zuletzt verbleiben noch die Anforderungen, dass es keine Einschränkungen im Simulink-Teil des Modells gibt und dass die Steuerung und Kontrolle der Simulation in Simulink stattfindet. Dadurch, dass die Hardwarekomponente als S-Function-Block in Simulink integriert wird, kann sie wie ein gewöhnlicher Simulink-Block behandelt werden. Im umgebenden Simulink-Teil des Modells steht die gesamte Funktionalität von Simulink zur Verfügung und die Steuerung der Simulation ist genau wie bei einem gewöhnlichen Simulink-Modell.

Jenseits dieser Anforderungen muss allerdings festgestellt werden, dass bei der Transformation des Modells zwei Eigenschaften des Originalmodells verloren gehen: die Synthetisierbarkeit des Hardwaremodells und einige Möglichkeiten zur Fehlersuche, die andere HDL-Simulatoren bieten. Der Verlust der Synthetisierbarkeit ist in diesem Zusammenhang unproblematisch, da das Originalmodell noch vorliegt und für die Synthese verwendet werden kann. Auch die eingeschränk-

ten Möglichkeiten zur Fehlersuche stellen kein großes Problem dar. In unserem Szenario soll das Simulationsmodell an ein anderes Team, die Systementwickler, weitergegeben werden. Tritt dort ein Fehler auf, so wird dieser Fehler an die Halbleiterentwicklungsabteilung kommuniziert und dort analysiert. Den Halbleiterentwicklern stehen das Originalmodell und daher auch alle gängigen Methoden zur Fehlersuche zur Verfügung. Ist der Fehler behoben, kann ein neues transformiertes Modell für die Systemsimulation erzeugt werden.

Als Erweiterung der grundlegenden Methode wurde in Kapitel 9 eine Optimierung vorgestellt, die eine weitere Beschleunigung der Simulation ermöglicht. Sie basiert auf dem Überspringen der Simulation von stabilen Zuständen, also Phasen in der Simulation, in denen sich der Zustand der Hardwarekomponente nicht verändert. Der Vorteil, der sich dadurch in der Simulationsgeschwindigkeit ergibt, ist stark vom jeweiligen Simulationsexperiment abhängig. Zum einen spielen natürlich Anzahl und Dauer der stabilen Phasen eine Rolle, zum anderen ist aber auch der Rechenaufwand relevant, der normalerweise für die Simulation des stabilen Zustands benötigt würde. Im DSP_2-Beispiel konnte eine sehr deutliche Beschleunigung erreicht werden, während der Geschwindigkeitsvorteil im GFB-Beispiel eher klein ist. Aufgrund des geringen Mehraufwands für die Verwaltung der stabilen Zustände ist grundsätzlich allerdings nicht davon auszugehen, dass sich die Erweiterung nachteilig auf die Geschwindigkeit der Gesamtsimulation auswirkt. Wie groß der Vorteil im Einzelfall ist, lässt sich jedoch nicht allgemein sagen.

Bezüglich der Anforderungen müssen für die Optimierung über stabile Zustände zwei Einschränkungen gemacht werden. Sie ist nicht vollständig automatisierbar und es kann Abweichungen im Verhalten der Komponente geben. Die Garantien der Hardwarekomponente müssen in Form von Zustands- und Abbruchbedingungen manuell vom Entwickler angegeben werden. Eine Automatisierung dieses Schritts für allgemeine Modelle wäre sehr aufwändig. Der manuelle Aufwand für einen Entwickler, der die Hardwarekomponente oder deren Datenblatt kennt, ist jedoch gering, wie die beiden Beispiele gezeigt haben. Im DSP_2-Beispiel sind die relevanten Bedingungen Teil der Programmierschnittstelle des DSP; im GFB-Beispiel können die benötigten Eigenschaften den Spezifikationen der Hardwarekomponente und des Anwendungsfalls entnommen werden. Umfassendes Wissen über die Implementierung der Hardwarekomponente ist in beiden Fällen nicht erforderlich.

Durch das Überspringen von stabilen Zuständen können Abweichungen im Verhalten der Hardwarekomponente auftreten. Da ein stabiler Zustand vom Entwickler als solcher definiert wird, kann nicht sichergestellt werden, dass es in diesem Zustand tatsächlich keine Aktivität in der Hardwarekomponente gibt. Prinzipiell können beliebige Bedingungen angegeben werden, um beliebige Phasen in der Si-

mulation zu überspringen. Die Tatsache, dass der Entwickler die stabilen Zustände definiert, bedeutet aber auch, dass dieser genau weiß, welche Phasen in der Simulation übersprungen werden. Damit bestimmt der Entwickler selbst, wo es Abweichungen im Verhalten geben kann und ob diese Abweichungen für das angestrebte Simulationsexperiment hinnehmbar sind oder nicht.

| | Eigener Ansatz | | Ko-Simulation | | | | Heterogene Umgebungen | |
|---|---|---|---|---|---|---|---|---|
| | DE-nach-DT-Transform | Stabile Zustände | HDL-Verifier | SystemC-Simulink | Verilog-Simulink | RT-nach-TLM | Ptolemy II, etc. | HDL-Erw. AMS |
| Simulations-geschwindigkeit | O | O | X | X | ? | O | O | O |
| Automatische Integration aller Modellteile | O | X | O | O | O | X | X | X |
| Exaktes Verhalten | O | X | O | O | X | X | O | O |
| Unterstützung synthetisierbare HDL-Modelle | O | O | O | O | O | O | ? | O |
| Keine Einschränkung in Simulink | O | O | O | O | O | ? | ? | X |
| Kontrolle in Simulink | O | O | O | O | O | ? | X | X |

Abbildung 10.1: Erfüllte Anforderungen im Vergleich mit Stand der Technik

In Abbildung 10.1 sind die Anforderungen nochmals im Vergleich mit anderen Lösungen dargestellt. Die grundlegende Methode zur Integration einer HDL-Komponente in eine Simulink-Simulation, die in dieser Arbeit entwickelt wurde, erfüllt alle zuvor gestellten Anforderungen. Zwar gehen bei der Transformation zwei Eigenschaften des Modells verloren, diese sind jedoch für den hier gegebenen Anwendungsfall nicht relevant und stellen somit keine Einschränkung dar. Die Optimierung über stabile Zustände ermöglicht eine weitere Beschleunigung der Simulation, verletzt aber zwei der gestellten Anforderungen. Es sind manuelle Schritte notwendig, der Aufwand dafür ist jedoch gering. Außerdem kann es zu Abweichungen im Verhalten der Hardwarekomponente kommen, die aber durch den Entwickler selbst definiert werden, und somit auch vom Entwickler kontrolliert werden können.

Betrachten wir zuletzt nochmals den in Abschnitt 1.1.1 beschriebenen Anwendungsfall: die Entwicklung eines MEMS-Sensors als System-in-Package. Das in Abschnitt 8.1.1 beschriebene DSP_2-Beispiel entspricht genau diesem Szenario. Es handelt sich bei dem Gesamtsystem um einen Drucksensor, der als System-in-Package realisiert werden soll. In Simulink liegt ein vollständiges Systemmo-

dell vor, und darin sollen nun die digitalen Verarbeitungsschritte durch ein DSP-Programm ersetzt werden. Der DSP liegt als synthetisierbares VHDL-Modell vor. Ebenso steht ein generisches ROM-Modell in Simulink zur Verfügung, in das ein DSP-Programm geladen werden kann. Um das DSP-Programm nun in der Gesamtanwendung testen und verifizieren zu können, muss das VHDL-Modell des DSP in das Simulink-Modell integriert werden. Die Simulation des Gesamtmodells mit integriertem DSP ist äußerst zeitintensiv. Im Rahmen der Evaluation der Simulationsgeschwindigkeit wurde lediglich ein Ausschnitt aus einem Testfall verwendet, der einen Druckverlauf über $30\,ms$ physikalischer Zeit simuliert. Mit gewöhnlichen Methoden zur Ko-Simulation dauert bereits diese Simulation etwa 20 Minuten. Der gesamte Testfall umfasst mehrere Sekunden physikalischer Zeit, und in einem industriellen Entwurfsprozess müssen zur Verifikation eine ganze Reihe solcher Testfälle simuliert werden. Insgesamt kommt so oft eine Simulationsdauer von vielen Stunden oder auch mehreren Tagen zusammen.

Die Evaluation hat gezeigt, dass ein großer Teil der Simulationszeit an der Ko-Simulationsschnittstelle verbraucht wird. Eine Vergrößerung der Schrittweite in der Simulation könnte diesen Anteil reduzieren, ist aber in diesem Fall nicht möglich, da RAM und ROM des DSP nicht Teil des DSP-Modells sind, sondern als eigenständige Teilmodelle in Simulink vorliegen. Das Taktsignal für RAM und ROM wird im DSP-Modell erzeugt und seine Frequenz entspricht der des DSP-Takts. Deshalb muss die Samplerate von RAM-, ROM- und DSP-Modell das Doppelte der gewünschten Taktfrequenz betragen. Eine Skalierung zwischen Takt- und Samplefrequenz ist folglich nicht möglich. Die Methode zur Ko-Simulation, die in dieser Arbeit entwickelt wurde, ermöglicht eine deutliche Reduktion des Rechenaufwands, der an der Schnittstelle zwischen Simulink-Modell und Hardwarekomponente benötigt wird. Mit ihr ist auch bei hohen Sampleraten eine schnellere Simulation des Gesamtmodells möglich. In diesem Beispiel konnte die Simulationsdauer so um etwa 25 % reduziert werden.

Zusätzlich dazu kann die Optimierung über stabile Zustände angewendet werden. Ziel der Simulationen ist die Verifikation des DSP-Programms und nicht des DSP selbst. Deshalb kann die in Abschnitt 9.5.1 beschriebene Optimierung angewendet werden, ohne dass dadurch die Qualität der Simulationsergebnisse beeinflusst wird. Insgesamt kann so die Dauer der Gesamtsimulation auf die Hälfte der ursprünglichen Dauer reduziert werden, was angesichts der sehr langen Laufzeit der Simulationen ein großer Vorteil ist.

## 10.2 Ausblick auf zukünftige Erweiterungen

Basierend auf der in dieser Arbeit entwickelten Methode haben sich einige Punkte ergeben, die in zukünftigen Arbeiten adressiert werden können.

### 10.2.1 Einfache Speicherung des Simulationszustands

Insbesondere die Tatsache, dass der Zustand der Simulation der Hardwarekomponente unabhängig von einem Simulatorzustand ist, bietet Möglichkeiten zur methodischen Erweiterung des bis hierher beschriebenen Konzepts. Da dieser Zustand nur vom internen Zustand des Hardwaremodells, also dem aktuellen Wert der internen Variablen, abhängt, ist das Speichern und Wiederherstellen von Simulationszuständen sehr einfach möglich. Neben dem grundsätzlichen Speichern von Arbeitspunkten eröffnen sich daraus auch zwei konkrete Anwendungen.

#### Anwendung zur Konfiguration der Hardwarekomponente

Hardwarekomponenten, wie zum Beispiel ein DSP, müssen häufig zunächst konfiguriert und initialisiert werden, bevor die eigentliche Funktionalität zur Verfügung steht oder die Programmausführung beginnen kann. Diese initiale Konfiguration wird oft über speziell dafür vorgesehene Standardschnittstellen wie SPI [Wikb] oder $I^2C$ [Wika] vorgenommen. Im Simulink-Modell auf Systemebene ist dieser Schritt in der Regel nicht vorgesehen. Das bedeutet, um die passende Konfiguration durchführen zu können, müssen die dafür notwendigen Elemente nachträglich in Simulink modelliert werden. Das kann äußerst aufwändig sein.

Die Tatsache, dass der Simulationszustand ohne viel Aufwand gespeichert und wiederhergestellt werden kann, eröffnet nun die Möglichkeit, dass die Konfiguration vorab in einer reinen HDL-Simulation durchgeführt wird. Aus vorhergehenden Komponententests stehen die notwendigen Elemente in der Regel als HDL-Modelle zur Verfügung. Damit kann nun eine Simulation der Hardwarekomponente bis zum Abschluss der Konfigurationsphase durchgeführt und der Zustand, der sich dann ergibt, gespeichert werden. In Simulink kann dieser Zustand zu Beginn der Simulation einfach geladen werden. Da das Modell der Hardwarekomponente keine eigene Zeitverwaltung hat, steht dem Benutzer darüber hinaus frei, ob die Konfigurationsphase in der Systemsimulation sichtbar sein soll oder ob die Simulation einfach direkt mit der konfigurierten Komponente beginnt. Im ersten Fall kann einfach ein Zeitraum am Anfang der Simulation definiert werden, in dem die Komponente nur wartet und nicht auf die Umgebung reagiert (vergl. stabile Zustände). Falls auf Systemebene verschiedene Konfigurationen evaluiert werden sollen,

dann können ohne Weiteres auch mehrere Modellzustände in der HDL-Simulation erzeugt und gespeichert werden. Auf diese Weise ist eine nachträgliche Modellierung der für die Konfiguration notwendigen Elemente in Simulink nicht mehr erforderlich.

### Anwendung bei variabler Schrittweite und kontinuierlicher Simulation

Eine weitere Anwendung ergibt sich bei kontinuierlichen Simulationen mit variabler Schrittweite. In Simulink wird dabei die sogenannte *Zero-Crossing-Detection* zum exakten Ermitteln der Zeitpunkte von Unstetigkeitsstellen eingesetzt (s. Abschnitt 2.4.1). Bei der Annäherung der Unstetigkeitsstelle wird ein Intervallverfahren verwendet, um den Zeitpunkt immer weiter einzugrenzen.

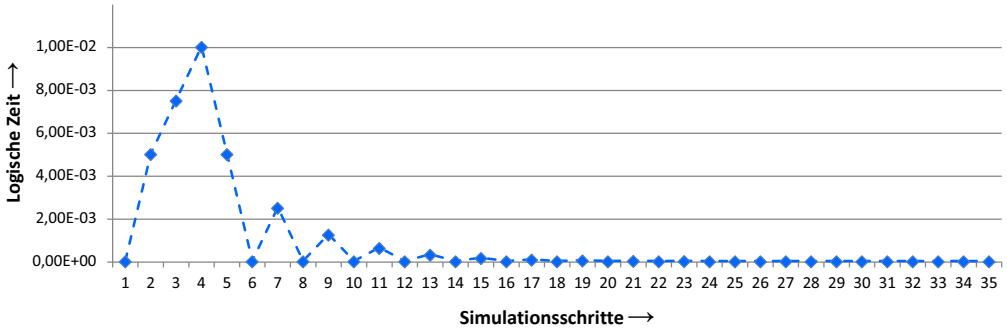



Abbildung 10.2: Verlauf der logischen Zeit bei Annäherung an Unstetigkeitsstelle.

In Abbildung 10.2 ist als Beispiel der Verlauf der logischen Zeit in der Simulation eines Simulink-Demonstrationsbeispiels (Bouncing Ball) dargestellt. Es ist zu erkennen, dass die logische Zeit nicht monoton ansteigt, sondern sich schrittweise von oben und unten dem Zeitpunkt der Unstetigkeitsstelle annähert. Wird in eine solche Simulation das Modell einer Hardwarekomponente integriert, so muss auch dieses Teilmodell in der Lage sein bei der Simulation zu vergangenen Zeitpunkten zurückzuspringen. Das ist bei einer Ko-Simulation mit einem gewöhnlichen HDL-Simulator nicht ohne Weiteres möglich.

Mit der in dieser Arbeit vorgeschlagenen Modelltransformation ist ein solches Verhalten allerdings einfach umsetzbar. Dazu ist es lediglich erforderlich sich den Zustand der Hardwarekomponente im jeweils letzten lokalen Minimum der logischen Zeit zu merken. Aufgrund der Natur des Intervallverfahrens werden diese lokalen Minima im weiteren Verlauf der Simulation nicht mehr unterschritten, ein Zurückspringen vor diesen Zeitpunkt ist somit nicht notwendig. Wenn dann der Zeitpunkt eines Simulationsschritts in der logischen Vergangenheit liegt, kann der

frühere Zustand des Modells wiederhergestellt und die Simulation der Hardwarekomponente bis zum Zeitpunkt des aktuellen Simulationsschritts fortgeführt werden. In der praktischen Umsetzung ist sogar ein explizites Speichern des Modellzustands überflüssig, wenn es zwei vollständige Sätze der internen Variablen gibt und an den passenden Stellen zwischen diesen Variablensätzen umgeschaltet wird. Damit wird eine Möglichkeit eröffnet, die kein anderes existierendes Verfahren zur Simulation von HDL-Modellen in einer Simulink-Umgebung bietet.

### 10.2.2 Sprache zur Notation und Synthese von Zustands- und Abbruchbedingungen

Ein weiterer Punkt, der in dieser Arbeit offen geblieben ist, ist die Frage einer allgemeinen Notation für die Zustands- und Abbruchbedingungen, die für die Optimierung über stabile Zustände benötigt werden. Vorstellbar wäre eine Sprache, die an eine formale Sprache zur Beschreibung von Eigenschaften digitaler Hardwaremodelle, zum Beispiel PSL [62212], angelehnt ist. In solchen Sprachen ist es möglich boolesche Eigenschaften, sowie Kombination und Folgen von booleschen Eigenschaften auszudrücken. Darüber hinaus gibt es Methoden und Werkzeuge zur automatischen Generierung von Monitormodulen, die diese Eigenschaften in einer Simulation überwachen. Diese Ansätze könnten auch hier angewendet werden, um die entsprechenden Teile im S-Function-Wrapper automatisch zu erzeugen.

## 10.3 Fazit

Ziel der vorliegenden Arbeit war es, eine Methode zu entwickeln, die eine effiziente Integration von synthetisierbaren Hardwarebeschreibungen in Simulink-Simulationen ermöglicht. Die Herausforderung ergibt sich dabei aus den unterschiedlichen Simulationskonzepten. Während die Simulation von Hardwaremodellen ereignisgetrieben ist, geschieht die Simulation von Simulink-Modellen datenflussorientiert und zeitgetrieben. Bei existierenden Ansätzen zur Simulation von Hardwaremodellen in Simulink werden zwei eigenständige Simulatoren eingesetzt. Dadurch fällt ein großer Mehraufwand für die Synchronisation, den Datenaustausch und die Kontextwechsel zwischen den beiden Modellteilen an. Das entwickelte Konzept nutzt die besonderen Eigenschaften des Anwendungsfalls aus, um diesen Mehraufwand zu reduzieren.

Das Modell der Hardwarekomponente ist in Form einer synthetisierbaren HDL-Beschreibung gegeben. Daher ist sichergestellt, dass das Modell bestimmte Elemente der Hardwarebeschreibungssprache nicht nutzt, insbesondere Zeitannota-

tionen werden in synthetisierbaren Modellen nicht verwendet. Dies und die Tatsache, dass das umgebende Systemmodell ein Simulink-Modell ist, können ausgenutzt werden, um eine Transformation des Hardware-Modells vorzunehmen. Ergebnis der Transformation ist ein Simulationsmodell, das ohne einen eigenständigen HDL-Simulator in Simulink simuliert werden kann. Die Kontrolle der Simulation und der logischen Zeit findet vollständig in Simulink statt. Deshalb ist der Aufwand, der für Synchronisation, Datenaustausch und Kontextwechsel zwischen den beiden Modellteilen erforderlich ist, deutlich geringer. Die semantische Äquivalenz des transformierten Modells mit dem Originalmodell der Hardwarekomponente wurde anhand der formalen Systemspezifikationsmodelle DTSS und DEVS gezeigt. Außerdem wurde im Rahmen der Evaluation gezeigt, dass das Ein-/Ausgabeverhalten beider Modelle gleich ist. Die Evaluation hat ebenfalls ergeben, dass aufgrund der entwickelten Methode eine höhere Geschwindigkeit in der Gesamtsimulation erreicht wird.

Eine weitere Anforderung war die automatische Anwendbarkeit der Methode. Dazu wurden Verfahren vorgeschlagen und zu großen Teilen auch prototypisch implementiert, mit denen die Transformation automatisiert und ohne manuelle Schritte durchgeführt werden kann. Damit sind die Forderungen nach einer hohen Simulationsgeschwindigkeit, automatischer Anwendbarkeit und der exakten Wiedergabe des Verhaltens in der Simulation vollständig erfüllt.

Eine weitere Optimierung der Simulationsgeschwindigkeit wurde möglich, weil durch die Modelltransformation der Zustand der Simulation der Hardwarekomponente nur noch durch den Zustand der Komponente selbst bestimmt ist. Es gibt keinen Simulatorzustand, der zusätzlich Einfluss auf den Simulationszustand hat. Deshalb können stabile Zustände in der Simulation der Hardwarekomponente einfach und effizient übersprungen werden. Auf diese Weise wird der dafür benötigte Rechenaufwand reduziert und die Geschwindigkeit der Gesamtsimulation kann nochmals gesteigert werden.

Als wesentlicher Beitrag der Arbeit ist zunächst die Formalisierung des Problems zu nennen, also die Abbildung der Simulationssemantik von Simulink und VHDL auf bekannte Formalismen zur Systemspezifikation sowie die Formulierung der speziellen Eigenschaften des Anwendungsfalls innerhalb dieser Formalismen. Auf dieser Basis wurde des Weiteren die Transformation des eingeschränkten Discrete-Event-Systems zu einem Discrete-Time-System entwickelt, die dann bezüglich besonderer Eigenschaften Takt-synchroner Hardwaremodelle erweitert wurde. Außerdem wurde die theoretische Methode auf das praktische Problem übertragen und ein Konzept zu ihrer automatisierten Anwendung beschrieben. Darüber hinaus wurde eine zusätzliche Optimierung entwickelt, die stabile Zustände in Hardwarekomponenten ausnutzt, um die Simulationsgeschwindigkeit noch weiter zu

steigern. Schließlich wurden die Methoden anhand von Beispielen bezüglich der gestellten Anforderungen evaluiert, und es konnte festgestellt werden, dass diese erfüllt sind.

Als Ergebnis dieser Arbeit steht somit eine Methode zur Verfügung, die eine automatische und effiziente Integration von Hardwarekomponenten in Simulink-Simulationen ermöglicht. Sie erlaubt eine schnellere Simulation, ohne dass dabei Einschränkungen bezüglich der Qualität der Simulationsergebnisse hingenommen werden müssen.

# Literaturverzeichnis

Die Einträge in diesem Literaturverzeichnis sind alphabetisch anhand der Namen der Autoren sortiert. Seitenzahlen am Ende der Einträge verweisen auf die Stelle der Referenz im Text.


[55510] IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)– Framework and Rules. In: *IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000)* (2010), S. 1–38. http://dx.doi.org/10.1109/IEEESTD.2010.5553440. – DOI 10.1109/IEEESTD.2010.5553440 40

[62212] IEC 62531:2012(E) (IEEE Std 1850-2010): Standard for Property Specification Language (PSL). In: *IEC 62531:2012(E) (IEEE Std 1850-2010)* (2012), S. 1–184. http://dx.doi.org/10.1109/IEEESTD.2012.6228486. – DOI 10.1109/IEEESTD.2012.6228486 146

[Ash01] *Kapitel* Appendix A: Synthesis. In: Ashenden, Peter J.: *The Designer's Guide to VHDL*. 2nd. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2001. – ISBN 1558606742, S. 639 – 654 24

[BBN+06] Bouchhima, F. ; Briere, M. ; Nicolescu, G. ; Abid, M. ; Aboulhamid, E.M.: A SystemC/Simulink Co-Simulation Framework for Continuous/Discrete-Events Simulation. In: *Behavioral Modeling and Simulation Workshop, Proceedings of the 2006 IEEE International*, 2006, S. 1 –6 36

[BC12] Bouissou, Olivier ; Chapoutot, Alexandre: An operational semantics for Simulink's simulation engine. In: *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*. New York, NY, USA : ACM, 2012 (LCTES '12). – ISBN 978–1–4503–1212–7, S. 129–138 43

[BFK94] Breuer, P.T. ; Fernandez, L.S. ; Kloos, C.D.: Clean formal semantics for VHDL. In: *European Design and Test Conference, 1994. EDAC, The European Conference on Design Automation. ETC European Test Conference. EUROASIC, The European Event in ASIC Design, Proceedings.*, 1994, S. 641–647 42

[BFP10] Bombieri, N. ; Fummi, F. ; Pravadelli, G.: Automatic Abstraction of RTL IPs into Equivalent TLM Descriptions. In: *IEEE Transactions on Computers* (2010) 37

[BTZ05] Boland, Jean-Francois ; Thibeault, Claude ; Zilic, Zeljko: Using MATLAB and Simulink in a SystemC Verification Environment. In: *Proc. of Design and Verification Conference & Exhibition*, 2005 36

[Cad] Cadence Design Systems: *Incisive Enterprice Simulator*. http://www.cadence.com/products/fv/enterprise_simulator, 35

[Car] Carbon Design Systems: *Model Studio*. http://carbondesignsystems.com, 24, 82, 97, 105

[CAT11] CATRENE: *Vision, Mission and Strategy - R&D in European Micro- and Nanoelectronics*. http://www.catrene.org/web/calls/whitebook2.php, feb 2011 1, 2

[CBFB03] Capocchi, L. ; Bernardi, F. ; Federici, D. ; Bisgambiglia, P.: Transformation of VHDL descriptions into DEVS models for fault modeling. In: *Systems, Man and Cybernetics, 2003. IEEE International Conference on* Bd. 2, 2003. – ISSN 1062–922X, S. 1205–1210 vol.2 42

[CG03] Cai, Lukai ; Gajski, Daniel: Transaction level modeling: an overview. In: *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. New York, NY, USA : ACM, 2003 (CODES+ISSS '03). – ISBN 1–58113–742–7, 19–24 37

[CHL97] Chang, Wan-Teh ; Ha, Soonhoi ; Lee, Edward A.: Heterogeneous Simulation – Mixing Discrete-Event Models with Dataflow. In: *J. VLSI Signal Process. Syst.* 15 (1997), jan, Nr. 1/2, S. 127–144 38

[DBSYB11] Duranton, M. ; Black-Schaffer, D. ; Yehia, S. ; Bosschere, K. D.: *Computing Systems: Research Challenges Ahead*. http://www.hipeac.net/system/files/hipeac-roadmap2011.pdf, oct 2011 2

[DJS93] Damm, W. ; Josko, B. ; Schlör, R.: A Net-Based Semantics for VHDL. In: *Proceedings EURO-DAC with EURO-VHDL 93*, 1993, S. 514–519 42

[EJL+03] Eker, Johan ; Janneck, Jorn ; Lee, Edward A. ; Liu, Jie ; Liu, Xiaojun ; Ludvig, Jozsef ; Sachs, Sonia ; Xiong, Yuhong ; Neuendorffer, Stephen: Taming heterogeneity - the Ptolemy approach. In: *Proceedings of the IEEE* 91 (2003), Nr. 1, S. 127–144 38

[FLLO95] French, Robert S. ; Lam, Monica S. ; Levitt, Jeremy R. ; Olukotun, Kunle: A general method for compiling event-driven simulations. In: *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, ACM, 1995. – ISBN 0–89791–725–1 23

[Fö94] Föllinger, Otto: Differenzengleichungsglieder. In: *Regelungstechnik*. Hüthing Buch Verlag Heidelberg, 1994, S. 54–55 11

[GBA+07] Goderis, Antoon ; Brooks, Christopher ; Altintas, Ilkay ; Lee, Edward A. ; Goble, Carol: Heterogeneous Composition of Models of Computation / EECS Department, University of California, Berkeley. 2007 (UCB/EECS-2007-139). – Forschungsbericht 38

[GN00] Ghasemzadeh, L. ; Navabi, Z.: A fast cycle-based approach for synthesizable RT level VHDL simulation. In: *Microelectronics, 2000. ICM 2000. Proceedings of the 12th International Conference on*, 2000, S. 281–284 82

[GNU] GNU: *GNU Diffutils*. http://www.gnu.org/software/diffutils/, 99

[GON12] Görgen, Ralph ; Oetjens, Jan-Hendrik ; Nebel, Wolfgang: Automatic Integration of Hardware Descriptions into System-Level Models. In: *Proceedings of the 2012 IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits and Systems* IEEE, 2012. – ISBN 978–1–4244–9754–6 83, 98

[Gö] Görgen, Ralph: *Cycle-Based Static Scheduling of VHDL*. http://vhome.offis.de/ralphg/schedvhdl.pdf, 80

[HBAV12] Hassairi, Walid ; Bousselmi, Moncef ; Abid, Mohamed ; Valderrama, Carlos: A SystemC/Simulink Co-Simulation Environment of the JPEG Algorithm. In: *International Journal of VLSI Design & Communication Systems*. Academy & Industry Research Collaboration Center (AIRCC), 2012, S. 1 –17 36

[HMU02] Hopcroft, John E. ; Motwani, Rajeev ; Ullmann, Jeffrey D.: Endliche Automaten. In: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Addison-Wesley, 2002, S. 45–91 11

[HON08] Hylla, Kai ; Oetjens, Jan-Hendrik ; Nebel, Wolfgang: Using SystemC for an extended MATLAB/Simulink verification flow. In: *FDL '08: Proceedings of the Forum on Specification and Design Languages*, 2008 36, 98

[iee09] Behavioural languages - Part 7: SystemC Language Reference Manual. In: *IEEE Std 1666 IEC61691-7 Edition 1.0 2009-12* (2009), 14, S. 1 –447. http://dx.doi.org/10.1109/IEEESTD.2009.5473172. – DOI 10.1109/IEEESTD.2009.5473172 20

[KXG+11] Kleen, Henning ; Xiao, Shangkun ; Görgen, Ralph ; Bannow, Nico ; Nebel, Wolfgang: Automatische Übersetzung von MATLAB/Simulink-Modellen nach SystemC-AMS. In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, 2011, S. 173–182 39

[LS00] Lungeanu, D. ; Shi, C.J.R.: Parallel and distributed VHDL simulation. In: *Design, Automation and Test in Europe Conference and Exhibition 2000. Proceedings*, 2000, S. 658–662 40

[Meh94] Mehl, H.: *Methoden Verteilter Simulation*. Vieweg+Teubner Verlag, 1994 (Programm angewandte Informatik). – ISBN 3528054395 41

[Men] Mentor Graphics: *ModelSim*. http://www.mentor.com/products/fv/modelsim/, 35

[MKBMG11] Mendoza, F. ; Kollner, C. ; Becker, J. ; Muller-Glaser, K.D.: An automated approach to SystemC/Simulink co-simulation. In: *Rapid System Prototyping (RSP), 2011 22nd IEEE International Symposium on*, 2011. – ISSN Pending, S. 135 –141 36

[MW05] MEHTA, Shaylesh ; WAINER, Gabriel A.: DEVS for mixed-signal Modeling based on VHDL. In: *Proceedings of the 2005 DEVS Integrative M&S Symposium, Spring Simulation Conference*. San Diego, CA. U.S.A, 2005 42

[Nar98] NAROSKA, E.: Parallel VHDL simulation. In: *Proceedings of the conference on Design, automation and test in Europe*. Washington, DC, USA : IEEE Computer Society, 1998 (DATE '98). – ISBN 0–8186–8359–7, 159–165 40

[NMS+04] NEIFERT, William ; MARANTZ, Joshua ; SAYDE, Richard ; TATHAM, Joseph ; LEHOTSKY, Alan ; LADD, Andrew ; SENESKI, Mark ; ATKINS, Aron: *SIMULATION OF SOFTWARE OBJECTS GENERATED FROM A HARDWARE DESCRIPTION*. Patent Application. `http://www.patentlens.net/patentlens/patent/WO_2004_044797_A2/en/`. Version: 05 2004. – WO 2004/044797 A2 24, 80, 82

[OGR06] OETJENS, Jan-Hendrik ; GERLACH, Joachim ; ROSENSTIEL, Wolfgang: Flexible specification and application of rule-based transformations in an automotive design flow. In: *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, 2006. – ISBN 3–9810801–0–6 89

[Par95] PARKS, Thomas M.: *Bounded Scheduling of Process Networks*, EECS Department, University of California, Berkeley, Diss., 1995. `http://www.eecs.berkeley.edu/Pubs/TechRpts/1995/2926.html` 38

[Pen06] PENRY, David A.: *The Acceleration of Structural Microarchitectural Simulation via Scheduling*, Princeton University, Diss., 2006 23

[Pop98] POPPEN, Frank: *Redesign eines VLSI Chips zur gehörgerechten Signalverarbeitung akustischer Signale*, Carl von Ossietzky Universität Oldenburg, Diplomarbeit, 1998 100

[Pto] PTOLEMY: *The Ptolemy Project*. `http://ptolemy.eecs.berkeley.edu/`, 38

[RHG+01] RUF, J. ; HOFFMANN, D. ; GERLACH, J. ; KROPF, T. ; ROSENSTIEHL, W. ; MUELLER, W.: The simulation semantics of systemC. In: *Proceedings of the conference on Design, automation and test in Europe*. Piscataway, NJ, USA : IEEE Press, 2001 (DATE '01). – ISBN 0–7695–0993–2, 64–70 42

[Rom] Romanov, Alexey: *Vmodel*. http://code.google.com/p/vmodel, 36

[Sal03] Salem, Ashraf: Formal Semantics of Synchronous SystemC. In: *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*. Washington, DC, USA : IEEE Computer Society, 2003 (DATE '03). – ISBN 0–7695–1870–2, 10376– 42

[Sys] *Accellera SystemC Simulator*. – www.accellera.org/downloads/standards/systemc 98

[Sys10] SystemC AMS Working Group: SystemC AMS Extension 1.0. (2010). – http://www.accellera.org 39

[Tas93] Tassel, John V.: A Formalisation of the VHDL Simulation Cycle. In: *Proceedings of the IFIP TC10/WG10.2 Workshop on Higher Order Logic Theorem Proving and its Applications*, North-Holland/Elsevier, 1993 (HOL'92). – ISBN 0–444–89880–8, 359–374 42

[Thea] The Mathworks Inc.: *HDL Verifier*. http://www.mathworks.com/products/hdl-verifier, 35

[Theb] The Mathworks Inc.: *MATLAB - The Language Of Technical Computing*. http://www.mathworks.com/products/matlab, 26

[Thec] The Mathworks Inc.: *Simulating Dynamic System in Simulink*. http://www.mathworks.com/help/simulink/ug/simulating-dynamic-systems.html, 26

[Thed] The Mathworks Inc.: *Simulink - Simulation and Model-Based Design*. http://www.mathworks.de/products/simulink, 26

[Tiw02] Tiwari, A.: Formal semantics and analysis methods for Simulink Stateflow models / SRI International. 2002. – Forschungsbericht. – http://www.csl.sri.com/~tiwari/stateflow.html 43

[Ver] Veripool: *Verilator*. http://www.veripool.org/wiki/verilator, 36, 83

[VER06] IEEE Standard for Verilog Hardware Description Language. In: *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), S. 0 –560. http://dx.doi.org/10.1109/IEEESTD.2006.99495. – DOI 10.1109/IEEESTD.2006.99495 20

[Ver09] Verilog-AMS Technical Subcommittee: Verilog-AMS Language Reference Manual. (2009). – http://www.accellera.org 39

[VHD04] IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis. In: *IEEE Std 1076.6-2004 (Revision of IEEE Std 1076.6-1999)* (2004), S. 1 –112. http://dx.doi.org/10.1109/IEEESTD.2004.94802. – DOI 10.1109/IEEESTD.2004.94802 24

[VHD09] Behavioural languages - Part 6: VHDL Analog and Mixed-Signal Extensions. In: *IEEE Std 1076.1 IEC 61691-6 Edition 1.0 2009-12* (2009), 14, S. 1 –0. http://dx.doi.org/10.1109/IEEESTD.2009.5464492. – DOI 10.1109/IEEESTD.2009.5464492 39

[VHD11a] Behavioural languages - Part 1-1: VHDL Language Reference Manual. In: *IEC 61691-1-1:2011(E) IEEE Std 1076-2008* (2011), 19. http://dx.doi.org/10.1109/IEEESTD.2011.5967868. – DOI 10.1109/IEEESTD.2011.5967868 20, 155

[VHD11b] *Kapitel* 12. Elaboration and Execution. In: [VHD11a] 81

[Wai02] Wainer, Gabriel A.: CD++: a toolkit to define discrete-event models, 2002, S. 1261–1306 42

[Wika] Wikipedia: $I^2C$. http://de.wikipedia.org/wiki/IIC, 144

[Wikb] Wikipedia: *Serial Peripheral Interface*. http://de.wikipedia.org/wiki/Serial_Peripheral_Interface, 144

[WLL95] Willis, J. ; Li, Zhiyuan ; Lin, Tsang-Puu: Use of embedded scheduling to compile VHDL for effective parallel simulation. In: *Design Automation Conference, 1995, with EURO-VHDL, Proceedings EURO-DAC '95., European*, 1995, S. 400–405 24

[Wor07] World Wide Web Consortium: *XSL Transformtions (XSLT) Version 2.0*. http://www.w3.org/TR/xslt20, 2007 89

[ZPK07] Zeigler, Bernard P. ; Praehofer, Herbert ; Kim, Tak K.: *Theory of modeling and simulation*. Acad. Press, 2007. – XXI, 510 S. 9, 10, 13, 15, 16, 159

# Abbildungsverzeichnis

# Tabellenverzeichnis