Zeitgenaue Simulation gemischt virtuell-realer Prototypen

Der Fakultät für Elektrotechnik und Informatik
der Universität Hannover
zur Erlangung des akademischen Grades
Doktor-Ingenieur
genehmigte

Dissertation

von Dipl.-Ing. Stefan Eilers

geboren am 5. Juni 1971 in Wilhelmshaven

2006

Referent: Prof. Dr.-Ing. C. Müller-Schloer
Korreferent: Prof. Dr.-Ing. Bernardo Wagner
Tag der Promotion: 12.01.2006

Vorwort

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Systems Engineering, FG System- und Rechnerarchitektur der Universität Hannover.

Mein besonderer Dank gilt Herrn Prof. Dr.-Ing. C. Müller-Schloer für die wissenschaftliche Betreuung dieser Arbeit. Besonders dankbar bin ich dafür, dass er mich in meinen Bemühungen immer bestärkt und kritisch begleitet hat sowie mir den nötigen Freiraum für die Gestaltung dieser Arbeit gewährte.

Herrn Prof. Dr.-Ing. Bernardo Wagner danke ich für die Übernahme des Korreferats.

Allen Kollegen und Studenten danke ich für die persönliche und wissenschaftliche Unterstützung. Besonders möchte ich mich bei Herrn Dipl.-Ing. Tim Oodes bedanken, dessen konstruktiv kritische Haltung mir immer ermöglichte, den richtigen Weg zu finden.

Abschließend möchte ich diese Arbeit Dr. Annette Schultz widmen. Ohne sie erschiene mir nichts als erstrebenswert.

Kurzfassung

In Systemen der Automobil-, Telekommunikations- und Automatisierungstechnik steigt der Anteil eingebetteter Systeme drastisch an. Neben komplexer werdenden mechatronischen Komponenten, ist ebenfalls ein zunehmender Einsatz von eingebetteten Steuerungen auf Basis von *Embedded Control Units* (ECU) zu verzeichnen. Diese steuern und überwachen nicht nur die mechatronischen Systemteile, sondern stehen untereinander in Verbindung und beeinflussen sich gegenseitig. Dies führt zu einer deutlichen Erhöhung der Komplexität und des prozentualen Anteils an Software innerhalb des Gesamtsystems.

Ein Entwickler hat somit einen Weg zu finden, um die Lücke zu schließen, die sich aufgrund der gegensätzlichen Randbedingungen – die Reduzierung der Entwicklungskosten und der Entwurfszeiten einerseits sowie die Gewährleistung der Korrektheit des Designs andererseits – ergibt. Hierzu existieren bereits verschiedene Methoden, wie z.B. das *Virtual Prototyping* und das *Rapid Prototyping*.

Werden das *Virtual Prototyping* und das *Rapid Prototyping* genau betrachtet, so ist zu erkennen, dass sich beide Konzepte nicht gegenseitig widersprechen, sondern sich ergänzen. Aus dieser Erkenntnis heraus entstand die Idee, beides miteinander zu einer gemeinsamen Entwurfsmethode zu vereinigen und somit eine effiziente Lösung für das Überprüfen der Korrektheit des Designs zu finden.

In der vorliegenden Arbeit werden beide Methoden diskutiert und bewertet. Aufbauend auf dieser Bewertung wird die Kombination beider Methoden zu einer neuen vorgestellt, wobei die Schwächen des Virtual Prototypings durch die Stärken des Rapid Prototypings ausgeglichen werden. Dies führt zu der Vorstellung einer inkrementellen virtuell-realen (V/R) Entwurfsmethode.

Mittels einer Beispielimplementation wird gezeigt, inwieweit diese Methode auf aktuellen Simulationsrechnern umsetzbar ist. Desweiteren wird demonstriert, wo die technischen Grenzen hinsichtlich maximaler Modellkomplexität bei der verwendeten Echtzeitsimulation und bei den unvermeidlichen Verzögerungen der notwendigen Schnittstellen zwischen virtuellen und realen Systemteilen liegen.

Die Einsatzfähigkeit der hier geschaffenen Realisierung und die Nutzbarkeit des vorgestellten V/R-Entwurfsablaufs werden abschließend anhand von Beispielen untermauert.

Schlagworte

Echtzeit-Simulation

Entwurfsmethode

Eingebettete Systeme

Softwareintensive Systeme

Abstract

Modern systems of the automobile, telecommunication and automation technology tend towards huge systems, contain mechatronic components and so-called embedded control units (ECU). These ECUs control the mechanical components and therefore leads to software extensive systems. Interaction via communication networks, is responsible for the bloat of the software complexity.

Developers have to close the increasing gap between minimizing design costs and time market on one hand, while they still have to assure the correctness of the design on the other hand. Existing methods are for example Virtual Prototyping and Rapid Prototyping which have several disadvantages if they are used on its own. The combination of Virtual- and Rapid Prototyping minimize these disadvantages and direct to an incremental virtual-real (V/R) design method.

The aim of this work is to discuss the advantages and disadvantages of both methods and how to combine them by using a proof of concept implementation of a hard real time simulation environment. It is shown, how this design method is practically realizable on a todays industrial personal computer.

The maximum complexity of the models to gain real time is analyzed as well as and the unavoidable delays by interfacing the virtual and the real parts of the system.

The practical usablility of the introduced V/R design method is shown successfully by means of real examples.

Keywords

real time simulation

design flow

embedded systems

software extensive systems

Inhaltsverzeichnis

Abbildungsverzeichnis

Häufig verwendete Abkürzungen

ABS	Anti Blockier System
ASIC	Application Specific Integrated Circuit
AWL	Anweisungsliste
CAN	Controller Area Network
CFSM	Co-design Finite State Machine
COSYMA	Cosynthesis for Embedded Architectures
Cx	C Extended
DFG	Deutsche Forschungsgemeinschaft
DIN	Deutsche Industrie Norm
DLL	Dynamic Linked Library
DVB-T	Digital Video Broadcasting - Terrestrial
ECU	Embedded Control Units
ESP	Elektronisches Stabilitätsprogramm
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GP-OS	General Purpose Operating System
GPS	Global Positioning System
GSM	Global System for Mobile Communication
HIL	Hardware-in-the-Loop
HW	Hardware
I/O	Input/Output
MDA	Model Driven Architecture
MSC	Message Sequence Charts
OS	Operating System
PIM	Platform Independent Model
PSM	Platform Specific Model
R	Realtime-Faktor
RT	Real Time
RTAI	Real Time Application Interface
RTL	Real Time Linux
RTL	Register Transfer Layer
SDL	Specification and Description Language
SOC	System on a Chip
SPS	Speicher Programmierbare Steuerung
STL	Standard Template Library
SUD	System Under Development
SUT	System Under Test
SW	Software
UML	Unified Modeling Language

UPSI ..... Universal Portable Simulation Interface
V/R ..... Virtual/Real
VHDL ..... Very High Speed Integrated Circuit Hardware Description Language
XMI ..... XML Metadata Interchange
XML ..... Extensible Markup Language

1 Einführung in die Themenstellung

In Systemen der Automobil-, Telekommunikations- und Automatisierungstechnik steigt der Anteil eingebetteter Systeme drastisch an. Neben komplexer werdenden mechatronischen Komponenten ist ebenfalls ein zunehmender Einsatz von eingebetteten Steuerungen auf Basis von *Embedded Control Units* (ECU) zu verzeichnen. Diese steuern und überwachen nicht nur die mechatronischen Systemteile, sondern stehen untereinander in Verbindung und beeinflussen sich gegenseitig. Dies führt zu einer deutlichen Erhöhung der Komplexität und des prozentualen Anteils an Software innerhalb des Gesamtsystems.

Hinsichtlich der notwendigen Kostensenkungen und der immer kürzer werdenden Produktionszyklen (time to market) erhöht sich der Druck auf die Entwickler in zunehmendem Maße. Verschlimmernd kommt hinzu, dass immer mehr sicherheitsrelevante Komponenten durch softwarebasierte Systeme gesteuert oder geregelt werden müssen. Während vor einigen Jahren, beispielsweise in der Automobilindustrie, ECUs lediglich für einfache Aufgaben, wie z.B. die Steuerung der Scheibenwischer und der Zentralverriegelung, verwendet wurden, sind sie heute ebenfalls für das Anti-Blockiersystem (ABS), für die Fahrzeugstabilisierung (ESP) und für das korrekte Auslösen der Airbags zuständig. In naher Zukunft werden sie außerdem die komplette Steuerung des Fahrzeugs kontrollieren (x-by-wire), was bei Fehlfunktionen lebensbedrohliche Folgen haben könnte.

In Folge dessen ist der Entwickler in immer stärkerem Maße gezwungen, unterschiedliche Subsysteme in den Systementwurf einzubeziehen und vor allem die Einflüsse aufeinander zu prüfen und zu bewerten. Weiterhin hat dies aufgrund des Zeitdrucks oftmals vor der Verfügbarkeit einiger der realen Komponenten zu erfolgen, wodurch ein realer Aufbau des gesamten Systems erst zu einem sehr späten Entwurfszeitpunkt möglich wird und somit hohe Kosten für notwendige Änderungen entstehen.

In der Vergangenheit führten diese Randbedingungen in der Automobilindustrie gerade bei den Luxus-Modellen zu erheblichen Ausfällen, in die besonders viele dieser neuen und softwarebasierten Systemen integriert wurden. Die aktuelle Folge dieser Entwicklung ist, dass die Anzahl und Komplexität softwarebasierter Systeme entweder wieder reduziert oder zumindest nicht weiter erhöht werden, da sich die resultierende Gesamtkomplexität als nicht beherrschbar darstellt.

Im Flugzeugbau sind Systeme bereits erfolgreich und relativ störungsfrei im Einsatz, obwohl sie deutlich komplexer als heutige Kraftfahrzeuge sind, wenn z.B. die Technik eines modernen Airbus als Vergleich herangezogen wird. Es stellt sich somit die Frage, warum die offensichtlich funktionierenden Entwurfsmethoden der Flugzeugindustrie nicht verwendet werden, um auch in anderen Bereichen ähnliche Ziele zu erreichen. Die Antwort muss wohl lauten, dass ein Automobil, welches mit den Entwurfs- und vor allem den Testmethoden des Flugzeugbaus entworfen wäre, wahrscheinlich technisch einwandfrei funktioniert, aber bezüglich des Preises gegenüber konkurrierender Automobile kaum eine Chance hätte. Auch eine Mehrfachauslegung von sicherheitsrelevanten Systemkomponenten wäre nicht zu bezahlen.

Ein Entwickler hat somit einen Weg zu finden, um die Lücke zu schließen, die sich aufgrund der gegensätzlichen Randbedingungen – die Reduzierung der Entwicklungskosten und der Entwurfszeiten

einerseits sowie die Gewährleistung der Korrektheit des Designs andererseits – ergibt.

In der Praxis existieren hierzu verschiedene Ansätze, die versuchen, dieses Spannungsfeld aufzulösen. Diese Methoden und Techniken können erfolgreich demonstrieren, wie die Entwurfszeiten reduziert werden können. Um aber korrekte Implementationen zu erhalten, muss eine Verifikation oder Validierung des Entwurfs herangezogen werden. Während die *Verifikation* bei heterogenen Systemen sehr schwierig ist, wird in der Praxis üblicherweise auf die Validierung mittels *Simulation* oder auf praktische Tests anhand *realer Modelle* oder so genannter *Rapid-Prototypen* zurückgegriffen. Anhand möglichst vollständiger Testszenarien wird mit diesen Vorgehensweisen die Korrektheit des Designs überprüft.

Die *Simulation* heterogener Systeme konnte bereits mittels des so genannten *Virtual Prototyping* erfolgreich realisiert werden. Bei dieser Methode werden das zu entwickelnde System und die Umgebung in verschiedene Domänen (z.B. digitale und analoge Elemente, Software, usw.) zerlegt und jeweils das zeitliche sowie das funktionale Verhalten modelliert, was zu Beginn durchaus auf abstrakter Ebene geschehen kann und im Laufe des Entwurfsprozesses zunehmend konkretisiert wird. Die so modellierten Domänen können immer zu einem simulierbaren Gesamtsystem zusammengefügt werden, wodurch zu jedem Entwurfszeitpunkt eine sehr genaue Validierung des Prototyps ermöglicht wird. Weitere Vorteile dieser Methode liegen in der guten Beobachtbarkeit des Systemverhaltens und der prinzipiellen Möglichkeit, jegliche Art von Messeinflüssen herauszurechnen.

Diese Methode zeigt in der Praxis allerdings verschiedene Schwachpunkte, da sie sehr genaue und validierte Modelle verlangt und darüber hinaus eine Methodik für den Übergang zum realen System nicht vorsieht. Zeigt der angefertigte reale Prototyp später ein fehlerhaftes Verhalten, so wird es für den Entwickler sehr schwierig, dieses mit Hilfe seiner virtuellen Entwurfsumgebung aufzuklären.

Ein anderer Ansatz wurde mit dem *Rapid Prototyping* verfolgt. Bei dieser Methode wird der Modellierungsaufwand bewusst vermieden, indem die zu testende Software möglichst früh in eine reale Umgebung integriert wird. Modellierungsfehler der Software werden somit prinzipiell erkennbar, gleichzeitig werden aber Aussagen über die notwendige Leistungsfähigkeit der später einzusetzenden ECUs schwierig oder unmöglich, da diese nicht modelliert und somit auch nicht berücksichtigt werden. Ebenfalls werden verschiedene Fehlerquellen durch diese Methode nicht sichtbar, da das Rapid-Prototyping-System in den meisten Fällen auf einer anderen Hardware basiert als das spätere Zielsystem und somit Einflüsse der Target Toolchain (Codegenerator, Compiler, Bibliotheken) nicht in die Beurteilung der Korrektheit einfließen können.

Werden das *Virtual Prototyping* und das *Rapid Prototyping* genau betrachtet, so ist zu erkennen, dass sich beide Konzepte nicht gegenseitig widersprechen, sondern sich ergänzen. Aus dieser Erkenntnis heraus entstand die Idee, beides miteinander zu einer gemeinsamen Entwurfsmethode zu vereinigen und somit eine effiziente Lösung für das Überprüfen der Korrektheit des Designs zu finden.

In der vorliegenden Arbeit werden beide Methoden diskutiert und bewertet. Aufbauend auf dieser Bewertung wird die Kombination dieser beiden Methoden zu einer neuen vorgestellt, wobei die Schwächen des Virtual Prototypings durch die Stärken des Rapid Prototypings ausgeglichen werden. Dies führt zu der Vorstellung einer inkrementellen virtuell-realen (V/R) Entwurfsmethode, die anhand einer Beispielimplementierung praktisch demonstriert wird.

Da das hier angesprochene Themengebiet naturgemäß sehr umfangreich und fragmentiert ist, wurde die folgende Analyse auf den Bereich der Entwicklung „softwareintensiver eingebetteter

Systeme“ eingegrenzt, der den Hauptfokus dieser Arbeit darstellt. Der Entwurf von Hardware ist in dieser Betrachtung nur von untergeordneter Bedeutung und wird somit nur am Rande behandelt.

1.1 Übersicht über diese Arbeit

Im *nachfolgenden Kapitel* wird der Stand der Technik in dem oben skizzierten Umfeld – der Entwicklung softwareintensiver eingebetteter Systeme – dargelegt. Zum besseren Verständnis wird zu Beginn auf grundlegende Begriffe eingegangen, bevor mit der Vorstellung der Entwurfs-, Modellierungs- und Simulationsmethoden begonnen wird. Ergänzend zu den jeweiligen Methoden werden ebenfalls Entwurfstechniken aufgeführt und verglichen. Das Kapitel schließt mit einer zusammenfassenden Bewertung, in der diskutiert wird, inwieweit die vorgestellten Methoden für die oben beschriebene Aufgabenstellung befriedigende Lösungsansätze liefern und an welchen Stellen noch Bedarf an Verbesserung gesehen wird.

In *Kapitel 3* werden die aus der Analyse resultierenden Anforderungen, in ein Lösungskonzept überführt, aus dem im Anschluss die notwendigen Voraussetzungen für die Realisierung der Beispielimplementierung formuliert werden.

Im Kapitel 4 werden die zur Realisierung notwendigen Aspekte weiter konkretisiert, indem sie in eine technische Realisierung übersetzt werden. Als Basis wird der im Institut für Systems Engineering, System- und Rechnerarchitektur entwickelte Systemsimulator ClearSim-MultiDomain für diese Arbeit herangezogen und hinsichtlich harter Echtzeitsimulation und der Realisierung von virtuell-realen Mischsystemen erweitert. Hierzu wird mit der Beschreibung des grundlegenden Aufbaus begonnen. Daraufhin werden die durchgeführten Änderungen und Erweiterungen erläutert in im Anschluss im Anschluss in *Kapitel 5* validiert. In diesem Kapitel werden weiterhin die Eigenschaften und Randbedingungen des Systems mittels Messungen untersucht. Eine zusammenfassende Analyse der aus den gewonnenen Erkenntnissen resultierenden Anwendungsmöglichkeiten und Grenzen dieser Realisierung schließt das Kapitel ab.

Aufgrund der Tatsache, dass eine Echtzeitsimulation eine anspruchsvolle Zielsetzung ist und von den verschiedensten Randbedingungen, wie z.B. der Komplexität der simulierten Modelle und der Leistungsfähigkeit des Simulationssystems, abhängt, wird in *Kapitel 6* die Fragestellung diskutiert und beantwortet, inwieweit die maximal zulässige Modellkomplexität für eine erfolgreiche Simulation unter Echtzeitbedingungen vorhersagbar ist. Anschließend wird mittels zweier Anwendungsbeispiele, die praktische Nutzbarkeit des Systems in *Kapitel 7* demonstriert. Die Beispiele sind jeweils so gewählt, so dass sie die Aspekte des hier vorgestellten inkrementellen Entwurfsablaufs inklusive des so genannten Cross Checkings demonstrieren und den Vorteil gegenüber den herkömmlichen Ansätzen unterstreichen.

Abgeschlossen wird diese Arbeit mit einer Zusammenfassung, einer abschließende Bewertung sowie einem Ausblick auf zukünftige Erweiterungen in *Kapitel 8*.

2 Stand der Technik

Im Folgenden soll – nach einer kurzen Einführung grundlegender Begriffe – mit einer Darstellung der gegenwärtigen Entwurfsmethoden und -techniken begonnen werden. Anschließend wird in Abschnitt 2.3 auf bekannte Modellierungsmethoden und -techniken eingegangen, die das Fundament für eine korrekte Realisierung der anschließend in Abschnitt 2.4 dargelegten Simulationsmethoden legen, gefolgt von einer Darstellung der Nutzungsmöglichkeiten der Simulation beim Test von Prototypen (Abschnitt 2.5). Abschließend werden in Abschnitt 2.6 die wesentlichen Punkte dieses Kapitels zusammengefasst und beurteilt.

2.1 Grundlegende Begriffe

Im Folgenden werden zunächst die grundlegenden Begriffe erläutert, die zum Verständnis der vorliegenden Arbeit entscheidend sind.

2.1.1 Die Domäne

Unter einer Anwendungsdomäne, häufig auch verkürzend Domäne, versteht man in der Informatik und insbesondere in der Softwaretechnik ein abgrenzbares Problemfeld des täglichen Lebens oder – etwas spezieller – einen bestimmten Einsatzbereich für Computersysteme oder Software. Anwendungsdomänen stellen typischerweise sehr spezielle Anforderungen an ein technisches System, welches zur Bewältigung der domänenspezifischen Aufgaben und Probleme eingesetzt werden soll. Diese Anforderungen fließen insbesondere im Rahmen der Anforderungsanalyse, die einer Systementwicklung vorausgeht, und während des Entwurfs des Systems in den Entwicklungsprozess ein und bestimmen maßgeblich die Modellbildung, die der späteren Realisierung zu Grunde liegt. Der Begriff wird häufig dann eingesetzt, wenn es für den betreffenden Einsatzbereich eine Vielzahl ähnlicher Systeme gibt, die allesamt die Anforderungen der Domäne umsetzen müssen. Anwendungsdomänen eignen sich daher gut für die Wiederverwendung von Architekturen und Systemkomponenten [4].

2.1.2 Der Systembegriff

Im Allgemeinen wird als *System* ein aus Einzelteilen bestehendes Ganzes bezeichnet (siehe Brockhaus). Was unter einem System genau verstanden wird, hängt sehr stark von dem jeweiligen Anwendungsfall ab, daher kann es keine befriedigende allgemeingültige Definition geben.

Bei den verschiedenen Entwurfsmethoden wird dieser Begriff oft zur Bezeichnung des Objektes verwendet, welches im Interesse des Entwicklers liegt und von der *Systemumgebung* unterschieden wird. In diesem Fall wird von dem *System under test* (SUT) bzw. *System under development* (SUD)

gesprochen. Beide befinden sich zueinander in einer Wechselbeziehung, indem zwischen ihnen Informationen ausgetauscht werden. Beide bilden zusammen als *Gesamtsystem* ein vollständiges Ganzes (siehe Abbildung 2.1).

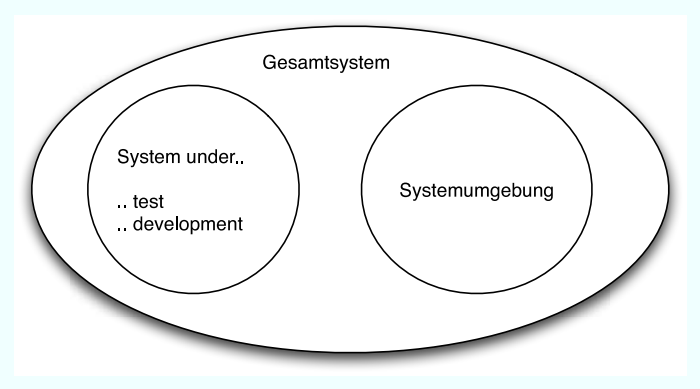


Abbildung 2.1: Aufbau eines Gesamtsystems

Auch wenn das Entwurfsziel sich ausschließlich auf das zu entwickelnde System bezieht, kann ein Systemtest nur anhand eines Gesamtsystems realisiert werden. Somit muss die Umgebung ebenfalls berücksichtigt werden. Hierzu existieren die verschiedensten Konzepte und Methoden, die im Kapitel 2.2 näher erläutert werden.

In unserem Kontext besteht das Gesamtsystem in der Regel aus technischen Komponenten. Woraus diese Komponenten aber bestehen, kann höchst unterschiedlich sein:

Während sich beim Hardware/Software Codesign das System ausschließlich auf digitale Komponenten und Software bezieht, kann dieser Begriff bei anderen Entwurfsansätzen, wie z.B. dem Multi-Domain Ansatz, sehr viel umfangreicher gefasst sein. Als weitere Domänen sind z.B. analoge oder auch stochastische Verhaltensmodelle denkbar, die mit den anderen Domänen in ständiger Wechselbeziehung stehen.

Somit ist festzuhalten, dass sich der Umfang und die Komplexität eines Systems und dessen Umgebung stark unterscheiden kann, je nachdem was genau im Interesse des Entwicklers liegt und somit beschrieben werden muss!

2.1.3 Das Aktorkonzept

Das Aktorkonzept oder Aktormodell wurde unter anderem durch Lisp, Simula und frühere Smalltalk Versionen inspiriert und 1973 zum ersten mal von Carl Hewitt et al. formuliert. Dieses Modell beschreibt verteilte, konkurrierende Systeme, indem sie als so genannte Aktoren dargestellt werden, die untereinander Nachrichten austauschen. Erhält ein Aktor eine Nachricht, so kann es lokale Entscheidungen treffen, neue Aktoren erzeugen und weitere Nachrichten aussenden. Weiterhin kann der Empfang einer Nachricht die Art der Verarbeitung einer zukünftigen Nachricht beeinflussen. Ein Aktor hat somit ein Gedächtnis.

Dieses Konzept beschreibt einen Paradigmen-Wechsel, der sich zu dieser Zeit in der Informatik vollzog. Konkurrierende Computermodelle verwendeten zu dieser Zeit globale Zustände, die über

gemeinsamen Speicher (Shared Memory) untereinander geteilt wurden und somit eine nichtdeterministische globale Zustandsmaschine (nondeterministic global state machine) darstellten.

Im Gegensatz hierzu zeichnet sich das Aktorkonzept durch folgende Aspekte aus:

- Es existieren keine globalen Zustände.
- Es herrscht Lokalität bezüglich der Nachrichtenverarbeitung, indem ein Aktor nur Nachrichten an andere Aktoren schicken kann, die durch Absender in empfangenen Nachrichten bekannt wurden, die selbst erzeugt wurden oder deren Adresse durch die Verarbeitung von Nachrichten errechnet werden konnten.
- Ebenfalls besteht Lokalität, indem keine simultanen Änderungen in mehreren Aktoren möglich sind, wie z.B. bei den Petri-Netzen, bei der so genannte Token simultan an mehreren Orten entfernt werden können.
- Es wird das Konzept der Kompositionalität (Compositionality) verwirklicht, also die Möglichkeit des Zusammenfügens von Aktor-Modellen zu größeren Strukturen, was die Grundlage für Modularität darstellt.

Ungefähr 8 Jahre nach der ersten Veröffentlichung, wurde das Aktorkonzept durch Will Clinger in seiner Domänen-Theorie (Domain Theory) erstmals als mathematisch beschreibbares Modell formuliert [4].

2.1.4 Echtzeitbetriebssysteme

Grundsätzlich kann zwischen zwei Arten von Echtzeitbetriebssystemen unterschieden werden. Es sind hier einerseits die homogenen Echtzeitbetriebssysteme und andererseits die Betriebsystem-Erweiterungen.

Betriebssystem-Erweiterungen sind Systeme, bei denen ein Standard-Betriebssystem (general purpose operating system, GP-OS), wie Windows oder Linux, durch eine Komponente ergänzt wird, die dem Entwickler ermöglicht, hart-echtzeitfähige Prozesse zu verwenden. Hierbei handelt es sich in der Regel um eine Software, welche die eingesetzte Hardware gegenüber dem GP-OS kapselt, die Interrupts virtualisiert und einen eigenen Echtzeitscheduler einsetzt, der vom GP-OS unabhängig ist. Dieser Scheduler ist so konzipiert, dass alle Realtime-Prozesse höher priorisiert sind als das GP-OS selbst.

Homogene Echtzeitbetriebssysteme sind Betriebssysteme, die vollständig und homogen echtzeitfähig konzipiert wurden. Somit gibt es bei diesen Systemen nicht die oben beschriebene Dualität.

Typische Vertreter der Betriebssystem-Erweiterungen sind „Real Time Linux“ (RTL) und das „Real-Time Application Interface“ (RTAI) für Linux und OnTime, sowie RTX für Windows NT. Bei den homogenen Betriebsystemen sind z.B. QNX, RTOS-UH, Nucleus, VxWorks, OS-9 und LynxOS zu nennen.

Beide Varianten ermöglichen prinzipiell ähnlich gute Echtzeiteigenschaften und bieten neben eigenen API's (Application Programming Interface) ebenfalls POSIX [54]-ähnliche Programmierinterfaces an. Der Vorteil der Erweiterungen liegt darin, dass gleichzeitig mit dem Einsatz von Echtzeitprozessen die Möglichkeit besteht, das System wie gewohnt einzusetzen, da es sich gegenüber Standardapplikationen identisch zu einer unmodifizierten Variante verhält. Bei den homogenen Systemen ist im Gegensatz hierzu eine unterschiedlich starke Einschränkung bezüglich der Kompatibilität gegenüber Standardapplikationen festzustellen. Weiterhin sind alle dem Autor bekannten homogenen Lösungen kommerziell und nicht quelloffen, was die Einflussnahme der Entwickler auf das System deutlich einschränkt.

Bei den Erweiterungen wird zumindest bei den linuxbasierten Lösungen die Quelloffenheit garantiert. Somit weisen sie eine sehr hohe Dynamik bezüglich Weiterentwicklung und Unterstützung verschiedenster Hardware auf. Die Dualität der Erweiterungen stellt für den Entwickler allerdings eine nicht zu unterschätzende Komplexität und Gefahr dar:

Da diese Dualität in der zu entwickelnden Software ebenfalls zu berücksichtigen ist, muss der Programmierer die Unterscheidung von Programmteilen mit bzw. ohne Echtzeitanforderungen programmtechnisch korrekt umsetzen. Hierzu kann schon die Verwendung eines einfachen „printf()“ anstatt z.B. eines „rt_printf()“ theoretisch zu einer Nichtvorhersagbarkeit der Zeitschranken führen. Weiterhin sind standardmäßig keine Lösungen vorgesehen, aus den Programmteilen mit harter Echtzeitforderung notwendige Operationen wie Datei- oder Socketkommunikation durchzuführen, was bei den homogenen Echtzeibetriebssystemen üblicherweise einfach möglich ist.

2.1.5 Unterscheidung: Entwurfsmethoden, -techniken und -prozesse

Zum besseren Verständnis der folgenden Kapitel ist die genaue Unterscheidung zwischen „Entwurfsmethoden“, „Entwurfstechniken“ und „Entwurfsprozessen“ wichtig:
Bei einer Entwurfsmethode handelt es sich um ein prinzipielles methodisches Vorgehen, ohne die technische Realisierung zu betrachten. Eine Entwurfstechnik stellt die technischen Grundlagen zur Realisierung einer Entwurfsmethode dar, wobei der Entwufsprozess den genauen Ablauf des Entwurfs inklusive der Benutzung und Kombination der Tools beschreibt.

2.1.6 Eingebettete Systeme

Unter einem *eingebetteten System* versteht man ein spezialisiertes (special purpose) Computersystem, welches ein anderes System kontrolliert, von dem es komplett umlossen ist. Ein eingebettetes System erfüllt im Gegensatz zu einem allgemeinen (general purpose) Computersystem ausschließlich eine spezielle Aufgabe [4].

2.2 Entwurfsmethoden und Techniken

Das Design eines Produktes kann mittels verschiedener Abstraktionsebenen beschrieben werden, wobei jede an der Entwicklung beteiligte Gruppe entsprechend ihrer Aufgabe eine unterschiedliche Perspektive einnimmt. Im Allgemeinen wird während des Entwurfsprozesses schrittweise die Abstraktionsebene von einer reinen Verhaltensbeschreibung über eine gröbere strukturelle Beschreibung bis hin zu einer exakten physikalischen Repräsentation präzisiert. Zum Schluss des Prozesses

liegt somit das fertige Produkt in einer der verwendeten Technologie und den vorher spezifizierten Randbedingungen entsprechenden Beschreibung vor, die keinen Spielraum für Interpretationen oder Ungenauigkeiten mehr enthält.

Diese Methode kann als allgemeingültig angesehen werden und wird häufig auch als „Top-Down Approach“ bezeichnet [22].

Da der Mensch nur ein begrenztes Maß an Komplexität wirklich erfassen kann, sind ihm beim Abstieg der Abstraktionsebenen und dem damit verbundenen Anstieg an Komplexität Grenzen gesetzt. Generell besteht die Gefahr, dass ab einer gewissen Komplexitätsgrenze die Übersicht verloren geht und sich somit Probleme oder Fehler in das Design einschleichen, die nur schwer zu finden sind, da die Entwickler überfordert sind. Lösungsansatz für zu hohe Komplexität ist somit die Aufteilung eines Systems in viele, weniger komplexe Einzelteile, die noch von den Entwicklern beherrscht werden können (so genanntes „Teile und Herrsche“ Prinzip [17]).

Die oben bezeichnete, allgemeingültige Entwurfsmethode kann sich in der Praxis in Variationen darstellen, die von Professor Daniel D. Gajski in drei verschiedene Gruppen aufgeteilt wurden [18]:

1. Capture-and-Simulate
2. Describe-and-Synthesize
3. Specify, Explore and Refine

Bei frühen Entwurfsmethoden wurde auf den hohen Abstraktionsebenen ausschließlich mit natürlichsprachlichen Beschreibungen gearbeitet, um die Eigenschaften und das gewünschte Verhalten – also die Spezifikation – des zu entwickelnden Systems zu beschreiben. Automatische Entwicklungstools unterstützten hier den Entwurfsprozess ausschließlich auf den unteren Abstraktionsebenen, wie beispielsweise der Logik- und der Schaltungsebene. Diese Methode kann als „Capture-and-Simulate“-Methode bezeichnet werden.

Modernere Entwurfsmethoden begannen schließlich, das Verhalten eines Systems mittels einer *formalen* Beschreibungssprache anstatt natürlichsprachlich zu definieren, um somit die Ungenauigkeiten der natürlichen Sprache zu vermeiden. Da die strukturelle und physikalische Beschreibung automatisch aus dieser formalen Beschreibung erzeugt (oder synthetisiert) werden kann, führte dies zu einer deutlichen Beschleunigung des Entwurfsprozesses und zur Vermeidung vieler Fehlerquellen. Dieser so genannte „Describe-and-Synthesize“ Ansatz beschränkte sich zu Beginn auf Beschreibungen der Gate-Ebene, wo mittels Zustandsmaschinen (FSM) und booleschen Gleichungen das funktionale Verhalten beschrieben wurde. Mit Verbesserung der Entwurfstools wurde es später möglich, auch mittels höherer Sprachkonstrukte, wie Flow-Charts, Datenflussgraphen und Hardware-Beschreibungssprachen, diese Verhaltensbeschreibungen auf Register-Transfer-Ebene (RTL) durchzuführen.

Aktuelle und zukünftige Modellierungsmethoden erweitern nun diesen Ansatz dahingehend, dass die rechnergestützte Beschreibung des Verhaltens zu einer formalen und ausführbaren Spezifikation wird, die zum frühen Entwurfszeitpunkt frei von technologischen oder physikalischen Eigenschaften definiert ist, aber zum späteren Entwurfszeitpunkt entsprechend präzisiert werden muss. Sie enthält nun erstmalig auch die formale Beschreibung von Eigenschaften und Randbedingungen (Constraints), welche das System einzuhalten hat. Diese vollständige und formale Systembeschreibung hat die Eigenschaft einer vollständigen Dokumentation und ermöglicht das automatisierte Erforschen verschiedener Designentscheidungen. Sobald die Entscheidung für eine spezifische

Technologie von den Entwicklern durchgeführt wird, wird es unmittelbar und einfach möglich, die Auswirkungen von Änderungen in der Spezifikation auf das System zu beobachten. Eine vollständige und ausführbare Systembeschreibung ermöglicht weiterhin, dass die Entwickler sich erst zu einem späten Entwurfszeitpunkt entscheiden müssen, welche Teile des Systems in Hardware oder in Software synthetisiert werden sollten. Die Folgen dieser Entscheidungen sind mittels Simulation der Spezifikation analysierbar und dienen somit als Kriterium für die endgültige Entscheidung des zu verwendenden Designs, weswegen Gajski diese Methode als „Specify-Explore-Refine“ Konzept bezeichnet hat (siehe allgemeinen Entwurfsablauf in Abbildung 2.2).

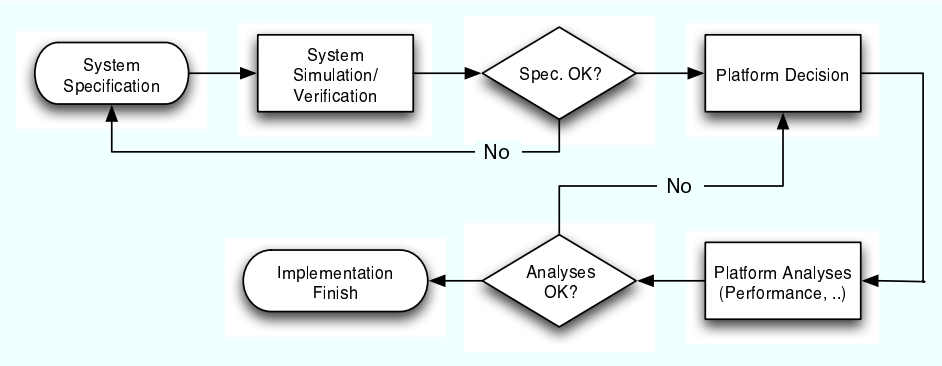


Abbildung 2.2: Beispiel eines allgemeinen Entwurfablaufs

Die hier beschriebenen Vorgehensweisen haben allgemeingültigen Charakter und schlagen sich in den verschiedenen Entwurfsmethoden entsprechend der jeweiligen Aufgabenstellung und der gewählten Perspektive unterschiedlich nieder. Im Folgenden sollen typische Entwurfsmethoden dargestellt werden, die für den Fokus dieses Papiers – die Entwicklung von eingebetteten Systemen mit dem Schwerpunkt auf Software-Entwicklung – relevant sind. Die folgende Auflistung kann nicht alles abdecken, was zu diesem Thema verfügbar ist, da der Schwerpunkt auf der Darstellung der üblichen Entwurfsmethoden liegt.

Es wird hierbei mit der Beschreibung des „Hardware-Software-Codesign“ und des „Interface Based Systemdesign“ begonnen (2.2.1, 2.2.2), bei denen rein digitale Systeme auf hoher Ebene beschrieben werden, die ausschließlich aus Software und Hardware bestehen. Bei der Model-Driven Architecture steht der Entwurf reiner Softwaresysteme im Vordergrund, indem Systeme mittels UML oder SDL abstrakt beschrieben und möglichst automatisch für eine später zu definierende Zielarchitektur übersetzt werden (2.2.3).

Ein so spezifiziertes System bildet die Grundvoraussetzung des oben eingeführten Konzeptes der „Specify-Explore-Refine“-Entwurfsmethode, wobei für die vollständige Umsetzung die Validierung oder Verifizierung der Spezifikation sowie die Validierung der möglichen Implementationen fehlt. Dies wird mittels Rapid Prototyping (2.2.4) und Virtual Prototyping (2.2.5) ermöglicht.

Zu jeder der praktisch eingesetzten Entwurfsmethoden wird eine Auswahl der verfügbaren Entwurfstechniken dargestellt, um das Bild abzurunden.

2.2.1 Hardware-Software Codesign

Da Software und Hardware konzeptuell sehr ähnlich sind und heutzutage Hardware generell mit ähnlichen Sprachen wie Software beschrieben wird (über so genannte Hardware-Beschreibungssprachen), ist der Gedanke naheliegend, beides mit einer Sprache zu beschreiben und somit eine homogene Systembeschreibung zu bilden. Als mögliche Eingabesprachen sind, neben vielfältigen proprietären Spracherweiterungen wie z.B. Cx bei COSYMA [30], vor allem SystemC [43] und verschiedene Varianten von endlichen Automaten, wie z.B. CFSM [16] sowie in Zukunft auch UML-Statecharts [59] zu nennen. Ein Entwickler erhält bei diesem Ansatz die Möglichkeit, sich zu Beginn seines Entwurfs vollkommen auf die funktionalen Aspekte seines Designs zu konzentrieren. Erst zu einem späten Zeitpunkt im Entwurf, nach Implementierung und Validierung der Funktion des Systems, wird entschieden, welche der Elemente des spezifizierten Systems in Hardware und in Software synthetisiert werden. Dadurch wird das System partitioniert. Als Grundlage für eine Entscheidung werden die Kosten für die benötigten Hardware-Komponenten, welche die entsprechenden Aufgaben übernehmen sollen, sowie die Analyse der Gesamtleistung eines partitionierten Systems herangezogen. Ist das System zu langsam, so müssen besonders zeitaufwendige Systemelemente in schnelle Hardware übersetzt werden. Ist ein System zu teuer (da Hardwarekomponenten zu aufwendig werden), so muss stärker in Software synthetisiert werden [28] (siehe Abbildung 2.3).

Nach Durchführung der Partitionierung wird die Systembeschreibung in eine Hochsprache für Software (typischerweise in C/C++) und Hardware (oftmals VHDL) synthetisiert und somit von einer homogenen zu einer heterogenen Beschreibung überführt. Die Software wird anschließend über eine übliche Compiler-Toolchain in Maschinensprache für den einzusetzenden Mikrocontroller übersetzt. Die Hardware-Beschreibung wird mittels Hardware-Synthesetools abhängig von der Zielarchitektur in Implementationen für FPGAs, ASICs oder Custom-Chips überführt.

Das Design kann durch eine Simulation der homogenen, aber auch der heterogenen Beschreibung überprüft werden. Weitere Möglichkeiten zur Validierung ergeben sich durch eine Ausführung auf einem Prototypensystem, was dann zu der Methode des Rapid Prototyping führt (siehe Kapitel 2.2.4).

2.2.2 Interface-Based System Design

Das Interface Based System Design beschreibt eine Erweiterung der Hardware/Software Codesign Methode. Diese Erweiterung wurde eingeführt, da für komplexere Systeme die Kommunikation zwischen den digitalen Systemkomponenten untereinander und mit der Software nicht mehr ausschließlich über speichergekoppelte Schnittstellen (memory mapped I/O) realisiert werden kann. Heutige Schnittstellen beinhalten verschiedene Bussysteme und sogar Netzwerkschnittstellen [26, 33]. Somit ist die oben beschriebene Entscheidung bezüglich der Implementierung einer Spezifikation, die sich beim reinen Hardware-Software-Codesign auf die Frage der optimalen Software/Hardware-Partitionierung beschränkte, um den Aspekt der Auswahl und Implementierung einer optimalen Schnittstelle erweitert.

Das Ziel dieser Methode ist somit, die im Hardware-Software-Codesign entwickelten Konzepte um Beschreibungsmöglichkeiten für Schnittstellen zu erweitern [63] und dadurch einen Top-Down Entwurf von der Spezifikation bis zur Implementation zu ermöglichen. Der Ansatz hierfür ist die Trennung der Kommunikation eines Systems von deren Funktion. Dies ermöglicht eine erhöhte

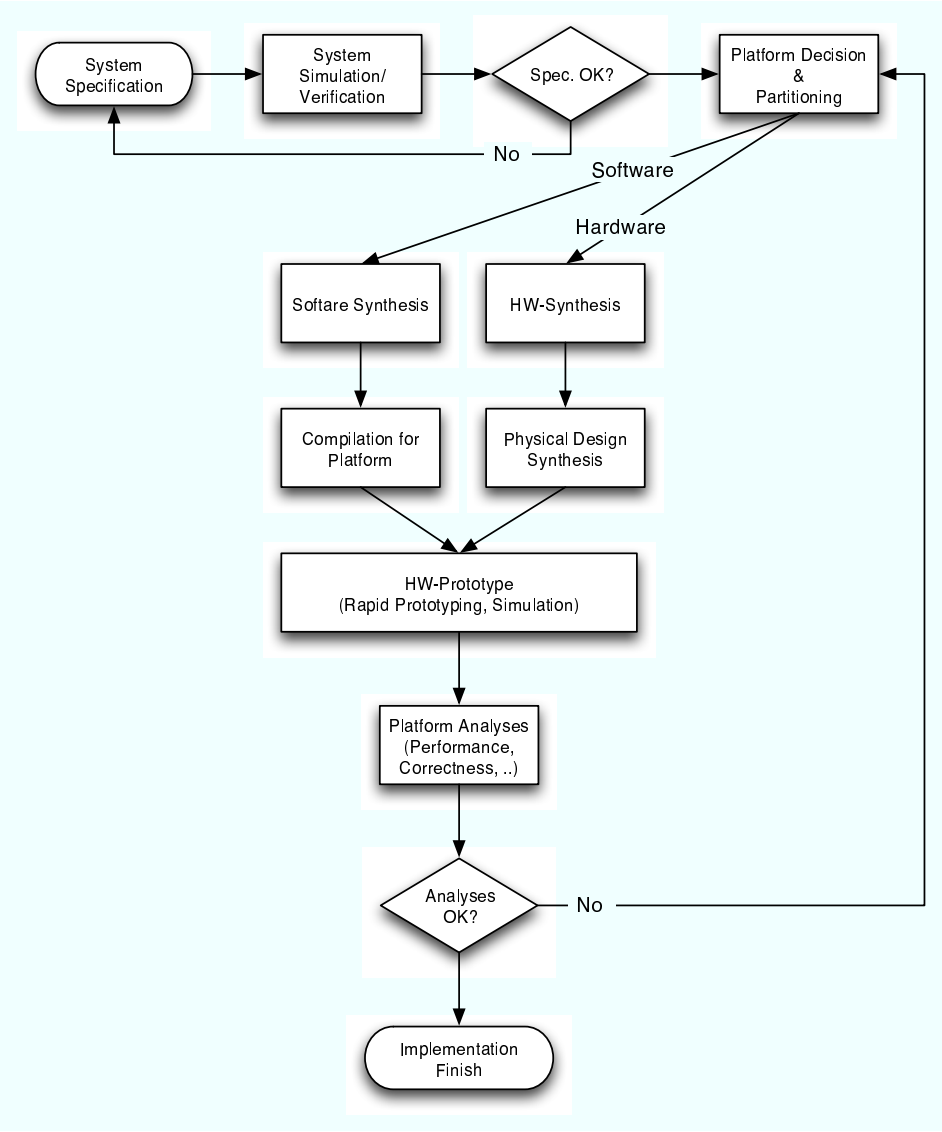


Abbildung 2.3: Hardware/Software Codesign

Modularität beim Chipdesign, da Funktionsblöcke somit einfacher und freier kombinierbar sind. Das Interface wird entsprechend der ausgewählten Kombination dieser Funktionsblöcke entwickelt, aus Bibliotheken ausgewählt (z.B. um standardisierte Busse zu nutzen) oder sogar automatisch erzeugt [55].

2.2.3 Model Driven Architecture

Als so genannte „Model-Driven Architecture“ (MDA) wird eine Methode bezeichnet, bei der das Verhalten der Software unabhängig von der Zielplattform modelliert und simuliert wird (Platform Independent Model, PIM). Nach der Definition der Zielplattform erfolgt anschließend automatisch die Erzeugung des plattformabhängigen Modells (Platform Specific Model, PSM) mittels entsprechender Tools unter Verwendung von Plattform-Bibliotheken [3]. In diesem Konzept wird die Rückwirkung der Plattformentscheidung auf das Verhalten des Systems nicht berücksichtigt (siehe Abbildung 2.4). Somit wird angenommen, dass die automatische Codegenerierung und die Abbildung auf die jeweiligen Fähigkeiten der Zielplattform befriedigend funktionieren.

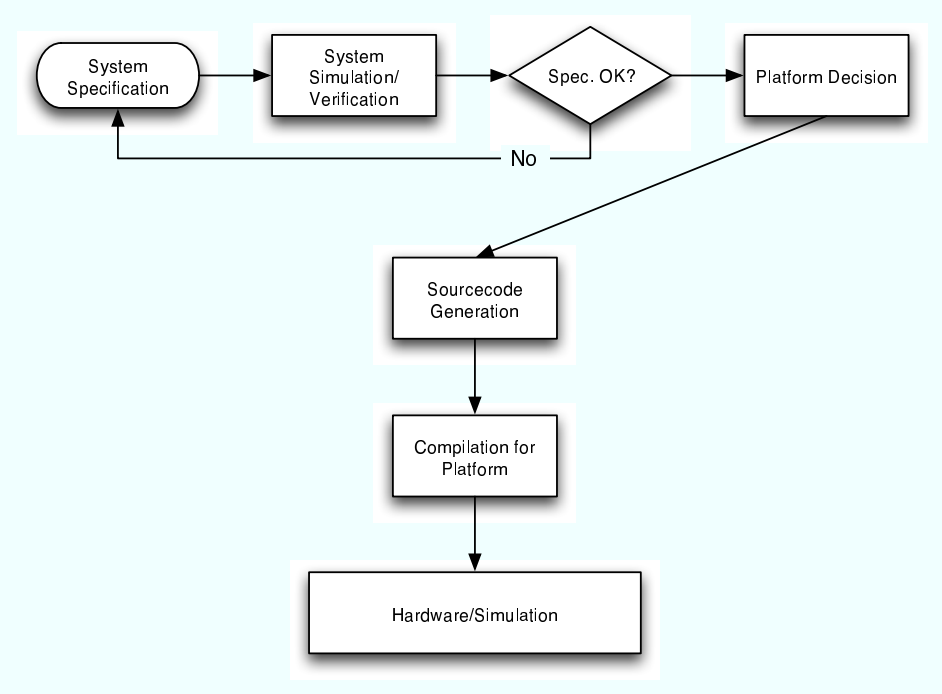


Abbildung 2.4: Model Driven Architecture

2.2.3.1 Techniken zur Model Driven Architecture

Zu dieser Methode gibt es bereits kommerzielle Entwurftools, die vor allem für Entwicklungen geeignet sind, welche ihren Schwerpunkt in dem Entwurf von Software für eingebettete Systeme haben. Der Entwurf von Hardware wird somit nicht unterstützt.

ETAS (ASCET-MD): Das zu entwickelnde System wird auf physikalischer Ebene anhand von Blockdiagrammen, Zustandsautomaten, Konditionstabellen oder Textuell in den Hochsprachen ESDL oder C spezifiziert. So spezifizierte Modelle können beliebig miteinander zu einem System verbunden werden. Dieses System wird nun in C und anschließend für verschiedene *Embedded Control Units (ECU)* oder einer Rapid-Prototyping-Hardware übersetzt, wodurch Hardware-in-the-loop-Tests ermöglicht werden. Diese Übersetzung ist sehr aufwendig, da auf der hochsprachlichen Ebene keine Rücksicht auf spezielle Eigenschaften der später zu verwendenden Hardware genommen wird. Fehlt auf dem Zielsystem z.B. eine arithmetische Einheit für Fließkomma-Operationen, so muss dies bei der Codegenerierung mühselig berücksichtigt/nachgebildet werden. Eine fehlerfreie Funktion dieser sehr komplexen Umsetzung ist ebenfalls fraglich und müsste auf dem realen Target aufwendig überprüft werden.

Telelogic (Telelogic Tau Suites): Die Tools von Telelogic bieten verschiedene Sprachansätze für den Entwickler und entsprechen somit dem Multi-Language Ansatz, wie er in Kapitel 2.3.2 beschrieben wird. Je nach eingesetztem Tool kann das System mit *UML* oder *SDL* (Specification and Description Language) spezifiziert und implementiert werden. Wird UML gewählt, so wird er in eine SDL-Beschreibung überführt, die anschließend in ausführbaren Code für *ECUs* mündet.

Die beiden hier beschriebenen Tools setzen den schon von Gajski geforderten Trend um, bei dem Entwurf der Spezifikation und der Implementierung möglichst auf hoher Ebene zu bleiben. Die hier entwickelte Software wird im praktischen Ansatz auf ein Rapid-Prototypingsystem übertragen und anhand einer realen Testumgebung überprüft. Ist der Test erfolgreich, so wird die Software für das einzusetzende Zielsystem übersetzt und dort ausgeführt. Die Folgen für den Fall, dass die Software auf dem Target System nicht oder nur unvollkommen funktioniert, werden hierbei nicht erwähnt. Dieser Fall kann aufgrund vielfältiger Ursachen auftreten, selbst wenn die Spezifikation an sich auf hoher Ebene korrekt sein sollte:

1. Die Codegenerierung für die Zielplattform schlägt fehl (Compiler-Fehler, Inkompatibilitäten zwischen Bibliotheken, usw.)
2. Die Bibliotheken, die fehlende Funktionen auf der Zielplattform nachbilden sollen (z.B. das Fehlen einer Fließkomma-arithmetischen Einheit), zeigen nicht exakt das Verhalten, das auf der hohen Ebene erwartet wurde. Dies kann z.B. dadurch passieren, dass bei einer Fließkomma-Berechnung eine andere Genauigkeit realisiert wurde als im Modell vorausgesetzt. Weiterhin kann eine reale Hardware, wie z.B. ein Digital/Analog Wandler, andere Auflösungen zeigen, als es auf hoher Ebene berücksichtigt/erwartet wurde. So ein Problem lässt sich auch mit einer Plattform-Bibliothek nicht lösen und darf prinzipiell nicht zu einer Erzeugung von Programmcode führen!
3. Das Zeitverhalten auf der realen Plattform führt zu fehlerhaftem Verhalten, da z.B. Informationen nicht rechtzeitig verarbeitet werden können und somit vorgeschriebene Sampling-Raten nicht eingehalten werden können.

All die hier genannten Probleme sind erst nach der Produktion der Zielhardware zu erkennen und verursachen möglicherweise hohe Kosten bei der Änderung oder bei der Analyse der Ursachen, zumal das Vorhandensein einer Testumgebung auf dem Zielsystem nicht vorgesehen und somit auch kein Mapping von Fehlern auf das Modell (Back annotation) realisiert ist.

2.2.4 Rapid Prototyping

In der Forschung wird die Notwendigkeit der Verbindung von virtuellen und realen Komponenten zu einem Gesamtsystem schon länger untersucht und z.B. im DFG Schwerpunktprogramm 1020 mit dem Thema „Rapid Prototyping für integrierte Steuerungssysteme mit harten Zeitbedingungen" behandelt. Das erklärte Ziel dieses Ansatzes ist es, möglichst schnell von einer ausführbaren Spezifikation zu einem lauffähigen Prototypen zu kommen, welcher über programmierbare Schnittstellen in eine reale Umwelt integriert wird. Ziel ist somit das Validieren einer Spezifikation (!) in der Praxis.

Die Motivation beim Rapid Prototyping liegt klar in der Vermeidung von Simulationen. Insbesondere soll eine Modellierung einer Umwelt vermieden werden. Es ist damit in diesem Punkt von dem Ansatz des virtuellen Prototypen zu unterscheiden (siehe Abschnitt 2.2.5).

Entsprechend der Abstraktionsebene des Prototypen kann zwischen den folgenden Varianten dieser Methode unterschieden werden [15, 48]:

- Konzept-orientiertes Rapid Prototyping: Hier wird die Spezifikation automatisch in einen ausführbaren Prototypen übertragen, wobei der hier auftretende Speicherverbrauch und die Hardwarekosten nicht von Interesse sind. Für die Abarbeitung der Software wird Echtzeitfähigkeit gefordert. Im Automobilbereich wird diese Art von Prototypen auch als A-Muster bezeichnet, welches die Machbarkeit eines Konzeptes aufzeigen soll und somit ausschließlich Demonstratorcharakter hat. Das Ziel dieses Prototypen ist ausschließlich die Klärung der Systemziele.
- Architektur-orientiertes Rapid Prototyping: Die endgültige Architektur des Systems steht fest. Einige Komponenten sind vorhanden, während andere sich noch in der Entwicklung befinden. Die Zielhardware (Mikrocontroller, FPGA, Signalprozessor) findet schon Verwendung, ist aber noch nicht für den Serieneinsatz optimiert. Diese Art von Prototyp wird im Automobilbereich als B-Muster bezeichnet und stellt das Vorbild für die Serienfertigung dar.
- Implementierungs-orientiertes Rapid Prototyping: Es handelt sich hier um ein hochgradig optimiertes System, welches zur Evaluierung der einzelnen Applikationen für den Serieneinsatz verwendet wird. Es werden nur noch letzte Änderungen evaluiert. Diese Prototypen sind als Vorlagen für die Serienfertigung geeignet und werden im Automobilbereich als C-Muster bezeichnet.

In der Forschung liegt der Schwerpunkt auf dem Konzept-orientierten Rapid Prototyping und – aufgrund seiner geringeren Allgemeingültigkeit – eher nachranging auf dem Architektur-orientierten und Implementierungs-orientierten Rapid Prototyping. Mittels HW/SW Codesign kann von dieser Methode aus ebenfalls der Entwurf der B- und C-Muster realisiert werden.

Im Gegensatz zum Ansatz des Virtual Prototyping wird dabei auf die Vorteile der Simulation weitestgehend verzichtet, da die Modellierung als zu zeitraubend angenommen wird, wie im nächsten Kapitel erläutert wird.

2.2.4.1 Techniken zum Rapid Prototyping

Wie bereits oben beschrieben wurde, versucht das Rapid Prototyping den Ansatz, möglichst früh von einer ausführbaren Systemspezifikation zu einem realen Prototypen zu kommen, welcher an-

hand einer realen Umgebung auf seine Tauglichkeit getestet wird. Der Schwerpunkt liegt in diesem Zusammenhang bei dem konzept-orientierten Rapid Prototyping. In der Forschung sind verschiedene Ansätze zu finden, wobei einige exemplarisch im Folgenden aufgezählt werden:

- Bei dem Projekt EVENTS [50] der Universität Oldenburg wird, aufbauend auf selbstentwickelten SPARC-Prozessoren (nach der Sparc V.8 Spezifikation) und der Verbindung mit FPGAs, versucht, ein System zu konstruieren, welches extrem schnell auf äußere asynchrone Ereignisse reagieren kann. Die eingesetzten Prozessoren sind jeweils in der Lage, bis zu 4 Threads direkt auf dem Prozessor zu verwalten (Multithreading Prozessor), welche innerhalb von nur 5 Prozessortakten gewechselt werden können. Da die Prozessoren über einen, hier nicht näher zu erwähnenden, Mechanismus als Slave-Einheit an die FPGA-Einheit angeschlossen sind, wird die Umschaltung der Threads über eine spezielle FPGA-Schaltung ausgelöst, die sehr schnell auf externe Ereignisse reagiert. Dieses Konzept ersetzt das übliche interruptbasierte Reagieren auf Ereignisse.
  Neben diesem extrem schnellen Reaktionsmechanismus ist es weiterhin möglich, über herkömmliches Software-Scheduling auf normale Ereignisse zu reagieren.
  Dieses Konzept ist aufgrund seiner Zielsetzung und seiner technischen Umsetzung nur für Spezialfälle und kaum für den Standardeinsatz geeignet, zumal die Spezialhardware aufgrund der hohen Kosten nicht für große Stückzahlen Verwendung finden wird. Erkenntnisse aus einem solchen Prototypen sind wohl auch kaum allgemeingültig zu verwerten.

- Im Zuge einer Kooperation zwischen den Universitäten München und Erlangen-Nürnberg ist ein System entwickelt worden, das aus einer formalen Systemspezifikation auf SDL- und MSC- (Message Sequence Charts) Basis, welche mit Zeitanforderungen angereichert wurde (SDL*, PMSC), eine Prototypen-Implementation erzeugt [23]. Nach dem üblichen Hardware/Software-Codesign Ansatz wird diese Beschreibung in C und VHDL Code synthetisiert und ausgeführt. Während die C-Beschreibung auf einem Echtzeit-Betriebssystem (RTEMS) ausgeführt wird, wird die VHDL Beschreibung entweder auf einem HW-Laufzeitsystem in Software ausgeführt oder in ein FPGA-System übertragen. Für die Ausführung stehen hierbei zwei Laufzeitumgebungen zur Verfügung, welche für unterschiedliche Fragestellung eingesetzt werden können und sich gegenseitig ergänzen:
  Das System, mit dem Namen „Phoenix“, besteht aus einer CPU, welche mit derjenigen des jeweiligen Zielsystems übereinstimmen sollte, und FPGAs. Hauptaufgabe für dieses System ist die Überprüfung der in SDL* spezifizierten Zeitschranken, indem in der Spezifikation die Instrumentierungspunkte definiert werden und anschließend mittels HW-Monitoring ausgemessen werden. Somit kann die Einhaltung der Zeitgrenzen validiert werden.
  Bei dem zweiten System, mit dem Namen „REAR“, wird die Spezifikation auf einem skalierbaren heterogenen Multiprozessorsystem ausgeführt, das keinen Bezug zu einem späteren Target hat. Ziel dieses Systems ist die Ermöglichung einer Echtzeitanalyse der Spezifikation, für die alle Ausführungszeiten der in SDL* spezifizierten Transitionen benötigt werden. Hierzu wird jede Transition am Eingangs- und Ausgangspunkt der Zustandsmaschine instrumentiert und mittels Hardware-Monitoring ausgemessen.

- Das Mechatronik-Laboratorium Paderborn legt seinen Schwerpunkt auf die Realisierung von hierarchischen Regelungen, die auf einem Workstation-Cluster als Offline-Simulation ausgeführt werden [65]. Bei diesem System werden die gleichen Regel-Algorithmen (in ANSI-C

programmiert) auf dem Simulator wie auch in einer HIL-Umgebung[1] eingesetzt, um Vergleichbarkeit zu gewährleisten, wobei kein Prozessormodell des späteren Target-Systems Verwendung findet.
Stufenweise werden Elemente der Simulation durch reale Komponenten ersetzt, wobei die Schnittstelle zwischen realem und virtuellem System ausschließlich auf gleicher Abstraktionsebene (einfache digital/analog Schnittstellen) realisiert wird. Die Ermittlung der Leistungsanforderung für ein später zu wählendes Target-System basiert auf Zeitmessungen (Hardware-Monitoring) im Zusammenhang mit Codeanalysen (Zählen und Gewichten) und Worst-Case Abschätzungen der Kommunikationslatenzzeiten.
Der Prototyp besteht aus mindestens einem Board mit PowerPC Prozessor und besitzt so die Möglichkeit, mehrere Boards transputerähnlich oder über einen CAN-Bus zu verbinden. Somit werden mögliche Einflüsse der Codegenerierung bezüglich des realen Targets in diesem System nicht betrachtet.

Auch in kommerziellen Produkten wird das Rapid Prototyping bereits genutzt. Der aktuelle Stand der Technik soll beispielhaft an den folgenden Produkten dargestellt werden:

Mathworks inc. (Matlab/Simulink): Die Firma Mathworks bietet ein erfolgreiches Softwarepaket an, welches die Bereiche *Software-in-the-loop*, *Hardware-in-the-loop* und eingeschränkt auch *Control-Prototyping*[2] unterstützt. Mittels Blockdiagrammen ermöglicht das Paket den Aufbau komplexer Regelstrecken, die auch in C-Code und somit in ausführbare Programme gewandelt werden können.

dSPACE: Die Firma dSpace erweitert das Paket von Mathworks durch ein selbst entwickeltes Rapid-Prototyping-System, basierend auf AMD-Opteron und PPC-Prozessoren, die über einen speziellen I/O-Prozessor vernetzbar sind. Der Schwerpunkt liegt auf *Hardware-in-the-loo*p, aber es ist mittels spezieller Software ebenfalls möglich, die C-Sourcen von Matlab/Simulink für spezielle Target-ECUs zu übersetzen.
Somit besteht auch mit diesem Produkt die Möglichkeit zum *Control Prototyping*.

Beide oben genannten Firmen verfolgen einen traditionellen, regelungstechnischen Ansatz, der durch die Verwendung von Blockdiagrammen deutlich wird. In neueren Arbeiten wird der stärkeren Verbreitung von standardisierten Notationen, wie SDL und vor allem UML, Rechnung getragen sowie die Kopplung kommerzieller CASE-Tools über das CDIF-Datenaustauschformat [20, 15] und XMI [6] realisiert. Somit werden die Techniken der Model Driven Architecture (MDA) verstärkt genutzt und die Spezifikation des Systems standardisiert. Weiterhin wird erkannt, dass die existierenden kommerziellen Entwurfstools die Kopplung von zeitkontinuierlichen und informationstechnischen Subsystemen in vielen Fällen nur auf der Codeebene unterstützen und dadurch die Erweiterung vom sehr eng gefassten Ansatz des Hardware/Software Codesigns auf einen umfassenderen Systembegriff erschwert. Ziel muss daher sein, die Kopplung auf Modellebene zu realisieren. Hierzu ist gerade UML hervorragend geeignet, da es die Umwandlung zwischen Block- und UML-Diagrammen ermöglicht [41].

All diese Ansätze zeigen einige wesentliche *Schwächen*: Da die abstrakten Spezifikationen lediglich auf einer Testhardware ausgeführt werden, fehlen Modelle der Zielsysteme und der Umgebung sowie

[1] Definitionen siehe in Kapitel: 2.5
[2] Definitionen siehe in Kapitel: 2.5

die Berücksichtung der später verwendeten Compiler-Toolchains. Auf den Prototypen-Systemen wird keine Simulation unter Berücksichtigung eines Plattformmodells durchgeführt, sondern die lauffähige Spezifikation ausgeführt, was einen wesentlichen Unterschied zum Virtual Prototyping darstellt. Somit ist neben der schon erwähnten Problematik einer korrekten Performance-Analyse auch z.B. der durchaus relevante Einfluss der Compiler nicht in der Untersuchung enthalten. Die beste Performance-Analyse auf einem synthetischen Prototypen hat nur geringe Aussagekraft, wenn auf dem späteren Target-System ein komplett anderer Compiler mit z.B. schlechterer Codeoptimierung eingesetzt wird oder eventuell in bestimmten Situationen einen fehlerhaften Code erzeugt. Die Möglichkeit durch statische Analysen und Hardware-Monitoring zumindest eine Worst-Case Abschätzung des Zeitverhaltens von Software durchzuführen ist zwar eine Verbesserung der hier geschilderten Problematik, aber stellt keine Lösung dar [62].
Auch die Annahme, dass das Validieren ausschließlich in einer realen Umgebung Vorteile für die Entwicklung bringt, kann bei heutigen Systemen durchaus in Zweifel gezogen werden. Denn die Verwendung massiv verteilter eingebetteter Systeme und die damit ebenfalls steigende Einflussname der Umgebung auf die Kommunikation untereinander kann kaum noch bezüglich Grenzbelastungen und Störeinflüssen systematisch überprüft werden, ohne dass die Kosten für die Testaufbauten und der zum Testen notwendige Zeitbedarf untragbar werden. Dies kann beispielsweise anhand eines modernen Automobils vor Augen geführt werden, welches in einen Tunnel einfährt. Die Unterbrechung diverser Kommunikationskanäle mit der Umwelt (GPS, GSM, Radio, DVB-T, ...) führt zu einer hohen Belastung des internen Bussystems. Kommen dann noch ungewöhnliche interne Ereignisse dazu, kann es zu einem Kollaps der Kommunikation führen Derartige Situationen wären in einem Praxistest anhand einer realen Umgebung kaum systematisch überprüfbar.

Die *Vorteile* des Rapid Prototyping liegen vor allem in der frühen Analyse der Systemziele und der grundlegenden Überprüfung der Korrektheit einer Spezifikation. Dadurch werden teure Rückschritte durch Fehlentscheidungen vermieden. Um die Entwicklung realer Prototypen heutiger Komplexität zu unterstützen, werden weitere Techniken benötigt, die die oben genannten Einflüsse berücksichtigen können und ebenfalls das systematische Testen unterstützten. Diese Techniken sollen im nächsten Kapitel beschrieben werden.

2.2.5 Virtual Prototyping

Dieses Konzept ist mit dem Bereich der so genannten *Virtual Reality* eng verbunden und erweitert die in diesem Zusammenhang entwickelten Technologien, um technische Abläufe, wie z.B. Produktionsabläufe, Anlagenautomatisierungen, bis hin zu eingebetteten Systemen mit Mikrocontrollern und Bussen, zu simulieren und zu visualisieren [34]. Im Gegensatz zum Rapid Prototyping wird beim Virtual Prototyping der Vorteil einer *vollständigen Simulation* eines Prototypen zusammen mit seiner Umwelt betont (siehe Abbildung 2.5). Besteht ein System auch aus nichtdigitalen Komponenten, so ist es nicht mehr sinnvoll, nur eine Sprache für die Spezifikation des Gesamtsystems zu verwenden (siehe Abbildung 2.6). Die Spezifikation wird somit in verschiedene Teilspezifikationen aufgebrochen, die jeweils entsprechend ihrer Domäne (Digital, Analog, Software, ...) mit einer jeweils optimalen Modellierungssprache abgebildet werden (siehe Kapitel 2.3).

Auch wenn die Modellierung – je nach Abstraktionsgrad der Modelle – teilweise viel Zeit in Anspruch nehmen kann, ist doch nur durch eine zusätzliche Modellierung einer kompletten Umgebung die Möglichkeit einer systematischen und reproduzierbaren Validierung des Prototypen gegeben. Auch die Vermeidung von Gefahren für Mensch oder Maschine sind in einer rein virtuellen Umgebung kostengünstiger zu realisieren. Bei der heutigen Komplexität von Systemen tritt ein weiterer

wichtiger Aspekt immer stärker im Vordergrund: Gerade im Falle einer sehr komplexen Interaktion mit der Umwelt ist das systematische Überprüfen von Störfaktoren durch die Umwelt nur durch das Virtual Prototyping praktisch realisierbar, da neben den hohen Kosten für komplexe Testaufbauten gerade der benötigte Zeitaufwand für die systematische Abarbeitung von Testsequenzen schnell zu einer Überschreitung der tolerierbaren Entwicklungszeit (time to market) führen kann.

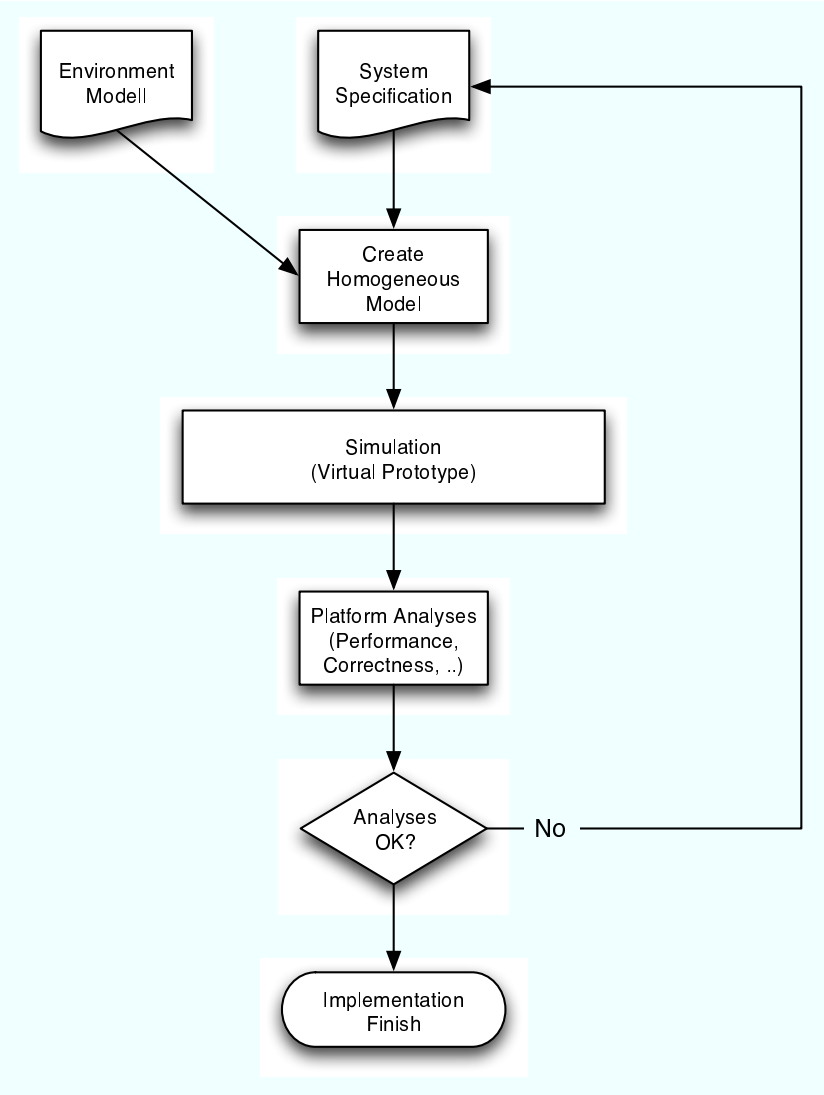


Abbildung 2.5: Virtual Prototyping mit homogener Modellsimulation (Single-Language Ansatz)

Um den hier beschriebenen Anforderung wirklich gerecht werden zu können, müssen die Modelle ausreichend genau sein, um überhaupt verwertbare Aussagen gegenüber dem späteren echten Prototypen machen zu können. Weiterhin ist die zeitlich genaue Simulation der verwendeten Modelle wichtig, da die korrekte Berücksichtigung des Zeitverhaltens neben der Notwendigkeit der Durchführung von Performance-Analysen einen nicht zu vernachlässigbaren Einfluss auf das funktionale Verhalten eines Systems hat. Dies verlangt einen erhöhten Aufwand bei der Modellierung sowie die unbedingte Notwendigkeit der Validierung der Modelle gegenüber den originalen Komponenten,

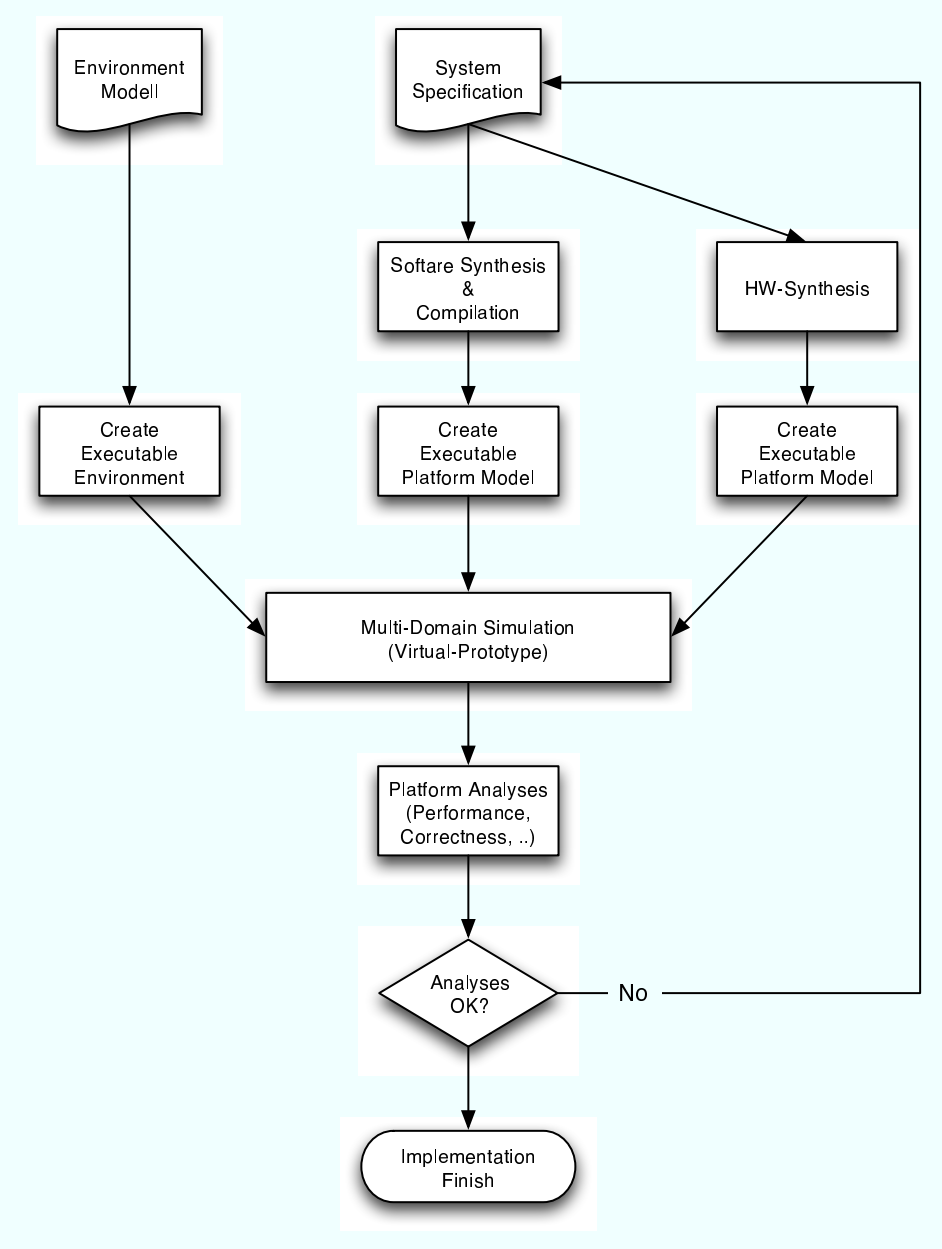


Abbildung 2.6: Virtual Prototyping mit heterogener Modellsimulation (Multi-Language Ansatz)

wie z.B. den einzusetzenden Mikrocontrollern.

2.2.5.1 Techniken zum Virtual Prototyping

ClearSim-MultiDomain: Der Schwerpunkt liegt bei diesem an der Universität Hannover entwickelten Simulationspaket in einer durchgängigen Unterstützung von zeitbasierten Modellen und der zeitlich korrekten Synchronisation dieser untereinander. Somit ist es nicht nur möglich, das funktionale, sondern auch das zeitliche Verhalten zu modellieren, da alle Modelle ein Zeitmodell besitzen müssen. Die Modellierung mittels EFSM (Extended Finite State Machines), SDL oder UML-StateCharts wurde um die Möglichkeit ergänzt, den Übergang von Zuständen mit einer Inkrementation der virtuellen Zeit zu koppeln und somit den Ablauf von Zeit zu modellieren. Aber auch die Modellierung analoger Komponenten kann erfolgen und wird im Zeitverhalten ebenfalls berücksichtigt. Bei Modellen von interpretierenden Systemen, wie z.B. einem Mikrocontroller oder einer SPS, wird das Zeitverhalten entsprechend des realen Pendants auf Instruktionsebene durchgeführt und ist dadurch die Voraussetzung führ eine sehr genaue Simulation der Software bezüglich ihres funktionalen als auch zeitlichen Verhaltens [11]. Um compilerabhängige Einflüsse zu berücksichtigen, wird die Software im originalen Binärformat interpretiert und ausgeführt [60]. Eine weitere Spezialität stellt in diesem Zusammenhang die Möglichkeit dar, Testroutinen in der Software zu belassen, die selber keinen Einfluss auf das Zeitverhalten haben dürfen. Damit lassen sich schon im frühen Entwurfsstadium korrekte Zeit- und Performanceanalysen durchführen. Weiterhin können die Programme durch so genannte Assertions erweitert werden, welche die Einhaltung bestimmter spezifizierter abstrakter Randbedingungen überprüft, um bei Verletzung dieser angemessen zu reagieren [52].
Zusammen mit der Verfügbarkeit von Modellen zur Simulation von Buskommunikation (z.B. den CAN-Bus), wird die zeitgenaue Simulation eines aus modernen Komponenten bestehenden, virtuellen Prototypen mit seiner Umgebung realisierbar.

Ansoft (Simplorer): Der Simplorer ist ein Multi-Domain Simulations-Framework der Firma Ansoft. Ähnlich ClearSim-MultiDomain gehört dieser ebenfalls zu der Klasse der Multi-Language modellierten Systeme (siehe Kaptitel 2.3.2). Diese Klasse zeichnet sich durch die Möglichkeit der Modellierung durch verschiedene Modellierungssprachen aus, wie in diesem Fall VHDL-AMS, CIRCUITS (numerische Modellierung von Schaltungen), Block Diagrammen (analoge, oder diskrete Signalfluss Modellierung) und State-Machines (Zustandmaschinen, FSM). Weiterhin besteht die Möglichkeit zur Einbindung anderer Simulationsumgebungen, wie Matlab/Simulink, MathCad, ADVISOR und anderen selbstgeschriebenen C/C++ Modellen. Hardware-in-the-loop oder ähnliches ist nicht möglich. Alle Modelle besitzen ebenfalls ein Zeitmodell und bilden somit eine zeitgenaue Simulationsumgebung, wobei aber bezüglich der Modellierung von komplexen Prozessoren oder sogar Mikrocontrollern keine Informationen vorliegen. Soll die Software für einen Mikrocontroller in die Simulation integriert werden, so wird diese als Shared-Library (DLL) gekapselt und in den Simulator eingefügt. Somit wird die Software ausschließlich funktional ausgeführt und nicht entsprechend eines späteren Target-Systems simuliert. Die Existenz von Modellen zur Simulation des Verhaltens von Bussen ist ebenfalls nicht dokumentiert. Der Datenaustausch zwischen den Simulations-Modellen findet auf niedriger Abstraktionsebene (Spannungen, Ströme) statt und hat seinen Schwerpunkt in der Simulation analoger Abläufe.

Ptolemy: Das Ptolemy Framework wurde von der Universität Berkeley ins Leben gerufen und wird inzwischen in der zweiten Version als Ptolemy II als komplett redesignte Java-Version zur Verfügung gestellt. Bei dieser Multi-Domain-Simulationsumgebung wird ebenfalls das Aktor- zusammen mit dem Multi-Language Konzept verwirklicht, wobei im Gegensatz zu z.B. ClearSim-MultiDomain eine objektorientierte Umsetzung existiert, die auch Vererbung und hierarchische Strukturen mit Aktoren erlaubt. Die Datentypen für die Kommunikation und das Zeitmanagement werden im Gegensatz zu ClearSim-MultiDomain nicht vom Simulations-Framework definiert, sondern durch die Domänenbeschreibung festgelegt, was einerseits zu einer höheren Flexibilität führt, aber andererseits eine Einschränkung bezüglich der prinzipiellen Kombinationsmöglichkeit untereinander zur Folge hat. Somit ist die Verbindung prinzipiell nur zwischen kompatiblen Domänen und Aktoren möglich. Als kompatibel sind hier Domänen zu bezeichnen, die bezüglich ihrer Vererbung den gleichen Ursprung haben und somit zwangsläufig gleichartig sind, also über die gleichen Datentypen verfügen und mit einem kompatiblen Zeitmodell arbeiten. Um ein Maximum an Variabilität zu erhalten, ist der Entwickler von Aktoren angehalten, seine Modelle *domain-polymorphic* und sogar *data-polymorphic* zu gestalten. Dadurch können die Aktoren in verschiedenen Domänen eingesetzt werden [14, 13]. Die Kommunikation zwischen den Aktoren erfolgt nachrichtenbasiert.

Die hier beschriebenen Simulationssysteme sind in der Lage, komplexe Systeme zusammen mit einer ebenfalls virtuellen Umgebung zu validieren. Aufgrund der guten Beobachtungsmöglichkeiten virtueller Umgebungen und der Fähigkeit, Messungen und Testroutinen ohne Beeinflussung des zu messenden Systems durchzuführen, ermöglicht diese Methode dem Entwickler eine sehr viel weitreichendere Analyse als es praktische Messungen an einem realen Prototypensystem könnten, bei der die Zielsoftware direkt ausgeführt wird.

Während beim Rapid Prototyping das funktionale- und das zeitliche Verhalten durch eventuell fehlerhafte Codegenerierung oder durch unterschiedliche Codeoptimierungen bzw. Plattformbibliotheken negativ beeinflusst werden kann, so werden beim Virtual Prototyping diese Einflussgrößen ausgeschlossen, indem das Prozessormodell funktional und zeitlich simuliert und der originale Binärcode direkt ausgeführt wird.

Die sich in diesem Zusammenhang aufdrängende Frage stellt sich aber bezüglich der Korrektheit der Modellierung. Die hier beschriebenen Vorteile verlangen eine sehr hohe Genauigkeit der Modelle und aus diesem Grunde einen hohen Modellierungsaufwand. Hierzu ist es notwendig, jedes Modell im Vergleich zu seinem realen Gegenstück zu validieren, da ansonsten keine gesicherten Erkenntnisse durch die Simulation gewonnen werden können, was den theoretischen Vorteil relativiert, mit einer Simulation vor der Existenz der echten Hardware beginnen zu können. Dieser Nachteil wird in der Praxis aber durch die Tatsache reduziert, dass heutige virtuelle Prototypen oftmals aus fertigen und validierten Komponenten kombiniert werden können, die in einer Bibliothek vorliegen.

Trotz aller Validierungen sind die Modelle niemals perfekt, sei es durch Modellierungsfehler oder unvermeidbare Ungenauigkeiten im funktionalen sowie zeitlichen Verhalten. Somit ist weiterhin nicht ausgeschlossen, dass der spätere reale Prototyp sich inkorrekt verhält, obwohl der virtuelle Prototyp problemlos funktioniert. In diesem Fall ist der Entwickler in einem ähnlichen Dilemma wie bei der Verwendung des Rapid Prototypings oder der Model Driven Architecture: Wie soll der Fehler gefunden werden, wo sich doch die Entwicklungs- und Testumgebung nicht auf dem realen aber fehlerhaften System befindet?

2.3 Modellierungsmethoden

Jede Entwurfsmethode muss Modelle verwenden, die das System für die gewählte Perspektive und Abstraktionsebene ausreichend genau beschreiben. Hierzu existieren eine Vielzahl von Notationsmöglichkeiten und Beschreibungssprachen (hier als *Modellierungstechniken* zusammenfassend bezeichnet), die für verschiedenste Anwendungen und Abstraktionsebenen effizient verwendbar sind. Eine gute Übersicht etablierter Sprachen und Techniken kann in [18] eingesehen werden.

Modellierungsmethoden können grob anhand zweier Überbegriffe zusammengefasst werden: Die so genannten „Single-Language“ Ansätze versuchen eine allgemeine Sprache zu verwenden, welche zur Beschreibung des kompletten Systems verwendet wird und somit eine homogene Spezifikation darstellt. Im Gegegensatz hierzu wird beim so genannten „Multi-Language“ Ansatz für jede Domäne jeweils eine Sprache gewählt. Dies führt zu einer heterogenen Beschreibung des Systems.

Im Folgenden soll der „Single-Language“ Ansatz gefolgt vom „Multi-Language“ Ansatz näher beleuchtet werden.

2.3.1 Single-Language Ansätze

Einen möglichen Ansatz zur Beschreibung von Systemen verfolgen die so genannten „Single-Language“ Ansätze, die vornehmlich beim *Hardware/Software Codesign* verwendet werden.

Bei diesen Ansätzen wird versucht, mittels *einer* möglichst allgemeinen Sprache das gesamte System (bestehend aus Software und Hardware) zu beschreiben.

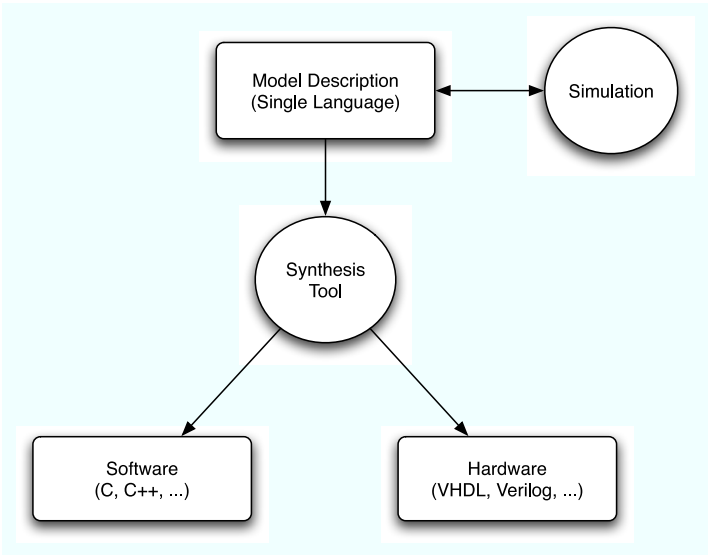


Abbildung 2.7: Single-Language Ansatz

Dieser Ansatz ist aber nur dann sinnvoll und effizient, wenn die beschriebenen Domänen sich sehr ähnlich sind (z.B. Software und digitale Systeme), da andernfalls die Sprachkonstrukte sehr komplex und unhandlich werden. Aus diesem Grund wird bei diesen Sprachen in der Regel auf die

Beschreibung analoger Systeme verzichtet, da nur relativ einfache digitale Systeme (SOC, System on a Chip) mit diesem Konzept befriedigend beschreibbar sind. Der Vorteil dieses Ansatzes liegt zum einen in der Möglichkeit der formalen Verifikation des kompletten Systems, zum anderen wird eine sehr späte Entscheidung ermöglicht, ob die Synthetisierung bestimmter Systemteile in Hardware oder Software erfolgen soll.

Der Entwickler kann sich somit zu Beginn des Entwurfsprozesses vollkommen auf die Implementierung der Funktionalität konzentrieren und muss zu diesem Zeitpunkt noch keine Technologieentscheidungen treffen. Das komplette, so modellierte und ausführbare System kann jederzeit während des Entwurfsprozesses mittels Simulator auf seine funktionale Korrektheit überprüft werden. Anschließend erfolgt die Partitionierung und Synthetisierung, die in den meisten Fällen C/C++ für die Software und VHDL als Hardwarebeschreibung extrahiert (siehe Abbildung 2.7).

Typische Vertreter dieser Kategorie sind unter anderem SystemC [43], COSYMA [30] und SpecC [25], aber auch Petri-Netze [67].

2.3.2 Multi-Language Ansätze

Beim Multi-Language Ansatz werden verschiedene Beschreibungssprachen nebeneinander verwendet. Hierzu müssen zwei verschiedene Ausprägungen unterschieden werden:

1. Die verschiedenen Beschreibungssprachen werden automatisch in eine gemeinsame „Zwischensprache“ überführt (siehe Abbildung 2.8) , durch welche dann die Simulation und anschließende Partitionierung und Synthetisierung für Hardware und Software erfolgt. Hierbei handelt es sich also um eine Erweiterung des oben beschriebenen Single-Language Ansatzes. Diese Erweiterung vermeidet den Kontakt mit der durch die Generalisierung unhandlich gewordenen Spezifikationssprache. Weiterhin erspart es dem Entwickler die Gewöhnung an eine ungewohnte und oft proprietäre Sprache, wie z.B. IRSYD [21]. Ein weiterer Vorteil dieser Zwischensprache ist die Möglichkeit, das Gesamtsystem, welches meist als endlicher Automat (FSM) vorliegt, auf seine Konsistenz zu überprüfen und formal zu verifizieren.
   Für die Anwendung einer allgemeinen Zwischensprache gilt aber grundsätzlich das Gleiche wie beim oben beschriebenen Single-Language Ansatz: Die Effizienz einer allgemeinen Sprache ist oftmals gegenüber den speziellen Sprachen deutlich geringer.

2. Es wird für jede Domäne die optimale oder verbreitetste Sprache und eine beliebige Abstraktionsebene gewählt. Aus diesem Grund wird dieser Ansatz auch als Multi-Domain Ansatz bezeichnet. Anschließend wird das mit der jeweiligen Sprache modellierte Modell in eine ausführbare Form überführt und zu einem Gesamtsystem kombiniert. Anders als beim „Single-Language“ Ansatz muss schon zu Beginn der Entwicklung die Partitionierung zwischen Hardware, Software und anderen Domänen (wie z.B. analogen Komponenten) erfolgen. Dieser vermeintliche Nachteil relativiert sich in der Praxis allerdings, da besonders bei komplexen Systemen diese Aufteilung zwangsläufig durch die in der Realität vorzufindene Trennung der Domänen untereinander erfolgt.
   Dieser Ansatz ermöglicht den Aufbau von komplexen virtuellen Prototypen, die aus dem zu testenden System und deren Umgebung bestehen können. Da keine Übersetzung in eine gemeinsame Sprache vorgenommen wird, bleibt eine gute Handhabbarkeit der Beschreibungssprachen erhalten und eine effiziente Ausführung möglich.

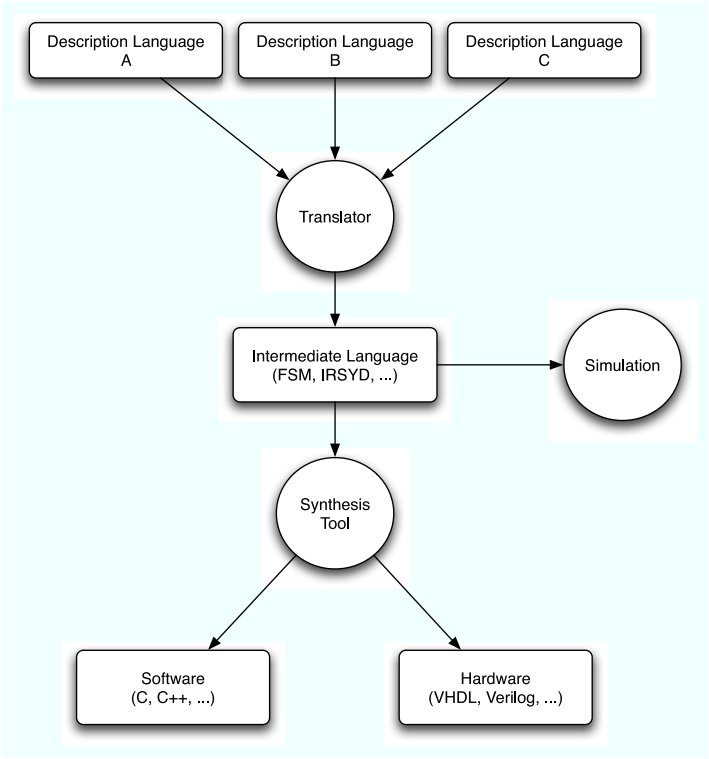


Abbildung 2.8: Multi-Language Ansatz mit gemeinsamer Zwischensprache

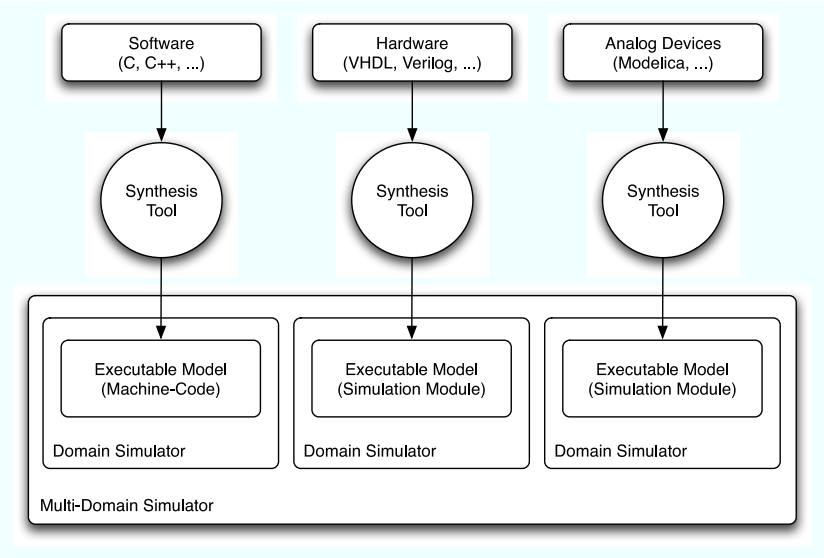


Abbildung 2.9: Multi-Language Ansatz ohne gemeinsame Zwischensprache

Eine formale Verifikation des Gesamtsystems ist mit diesem Ansatz nur schwer möglich. Die Simulation des so modellierten Gesamtsystems bleibt aber realisierbar und erfolgt über einen Multi-Domain Simulator als so genannte Cosimulation, indem die Modelle jeweils ausgeführt und über eine geeignete Lösung miteinander synchronisiert werden (siehe Abbildung 2.9 und Abschnitt 2.4.4).
Grundsätzlich handelt es sich bei bei dem Multi-Domain Ansatz um den allgemeingültigsten aller Ansätze, welcher die anderen Modellierungsansätze jeweils als Domänen zu integrieren vermag, um von deren Vorteilen dort zu profitieren wo es sinnvoll erscheint.

Ein typischer Vertreter der 1. Kategorie ist z.B. das POLIS [16] Framework, bei dem das System mit der Sprache Esterel [9], mit grafischen FSM's und Untermengen von Verilog und VHDL implementiert werden kann. Anschließend werden die Sprachen in endliche Automaten (CFSM) übersetzt, die als Komplettsystem z.B. mit Ptolemy [14] simulierbar sind oder auch formal verifiziert werden können.

Als Beispiel für die 2. Kategorie sind z.B. Ptolemy und das am Institut für Systems Engineering, System und Rechnerarchitektur (SRA) entwickelte ClearSim-MultiDomain zu nennen [61]. Beide verwenden das Aktor-Prinzip von Hewitt [31], bei dem die verschiedenen Domänen als Aktoren dargestellt werden. Diese sind untereinander über einen Graphen verbunden, über dessen Kanten (auch als Kanäle bezeichnet) die Daten ausgetauscht werden. Die Kommunikation übernimmt hierbei ein zentraler Simulations-Manager (auch Simulations-Kernel genannt), der bei ClearSim-MultiDomain zusätzlich zur reinen Nachrichtenkommunikation ebenfalls die zeitliche Synchronisation übernimmt [60, 61].

2.4 Simulationsmethoden

Die Simulation ist eine Vorgehensweise, bei der dynamische Systeme analysiert werden. Es werden hierzu Modelle dieser Systeme erzeugt, an denen Experimente durchgeführt werden. Grundsätzlich muss die Korrektheit eines Modells vor den Experimenten nachgewiesen werden, da ansonsten die Aussagen der Simulation nicht vertrauenswürdig sind. Eine wesentliche Voraussetzung für die Korrektheit und Nutzbarkeit eines Modells besteht in der Wahl einer angemessenen Abstraktion gegenüber dem realen System, die ausreichend genaue Ergebnisse garantiert, aber eine nicht zu geringe Simulationsgeschwindigkeit besitzt.

Während einer Simulation verändert sich der Zustand des Modells. Diese Veränderung des Zustandes kann als *Simulationszeit* aufgefasst werden, wobei diese Zeiteinteilung grundsätzlich keinen Bezug zu der *Rechenzeit des Simulators* (seiner Simulationsgeschwindigkeit) haben muss. Stimmen die Simulationszeit und die Rechenzeit allerdings überein, dann spricht man auch von Echtzeitsimulation (siehe 2.4.5).

In den folgenden Abschnitten soll auf weitere Unterscheidungsmögichkeiten der Simulationsarten eingegangen werden.

2.4.1 Kontinuierliche Simulation

Bei der kontinuierlichen Simulation erfolgt die Veränderung des Modells kontinuierlich, indem in der Regel Differenzialgleichungen der expliziten Art ($\dot{x} = f(x, u, t)$, mit $x(t)$als Zustandsgröße und

$u(t)$als Eingangsgröße), mittels numerischer Integration gelöst werden und somit *pro Zeitintervall* beliebig viele Zustandsveränderungen möglich sind. Wird das Zeitintervall Δt ausreichend verkleinert, so nähert sich die Simulation einem quasi-kontinuierlichen System an. Da bei jedem Intervall die gleichen Berechnungsschritte auszuführen sind, erhöht sich die Rechenzeit der Simulation bei Verkleinerung der Zeitschritte ebenfalls entsprechend. Weitere Informationen zur kontinuierlichen Simulation sind in [60] nachzulesen.

2.4.2 Diskrete Simulation

Bei der diskreten Simulation werden pro Zeitintervall nur eine endliche Zahl von Zustandsänderungen betrachtet, was besonders der Beschreibung von digitalen – und somit diskreten – Systemen nahe kommt. Die Zustandsänderung erfolgt sprunghaft zu bestimmten, klar unterscheidbaren Zeitpunkten. Als *Ursache* für eine Zustandsänderung kann das Fortschreiten einer Simulationszeit, aber auch das Auftreten von Ereignissen herangezogen werden. Der erste Fall wird somit als *„zeitgesteuerte Simulation“* und der zweite Fall als *„ereignisgesteuerte Simulation“* bezeichnet.

2.4.2.1 Ereignisgesteuerte Simulation (Event-Driven Simulation)

Bei der ereignisgesteuerten Simulation wird der Zustand eines Modells durch das Auftreten eines Ereignisses verändert, welches zu einer bestimmten, dem Ereignis aufgeprägten Ereigniszeit (auch als Zeitstempel bezeichnet) berücksichtigt oder verarbeitet werden muss. Wichtig ist hierbei, dass die Ereigniszeit zum Zeitpunkt der Verarbeitung möglichst identisch mit der Simulationszeit des Empfängers ist, da es ansonsten zu Ungenauigkeiten kommen würde.

Der große Vorteil der ereignisgesteuerten Simulation ist seine Effizienz bei sehr hoher Genauigkeit, da der Simulator nur Rechenzeit verbraucht, wenn wirklich Ereignisse im System erzeugt werden, die möglicherweise eine Änderung des Zustandes verursachen können. Ist das System für einen gewissen Zeitbereich stabil (es treten keine Ereignisse auf), dann kann diese so genannte *Totzeit* übersprungen werden.

Besteht eine diskrete Simulation nur aus *einem* Simulationsmodell (im Gegensatz zur Hybriden- oder Cosimulation, siehe Kapitel 2.4.3 und 2.4.4), dann kann die Funktionsweise einer Simulation folgendermassen vereinfacht beschrieben werden [47]:

Der Simulator besteht aus einer globalen Ereignisliste, einer Uhr und Ereignisroutinen. Alle Ereignisse treten in chronologischer Reihenfolge auf und werden in der Ereignisliste in der Reihenfolge des Erscheinens eingehängt. In einer Schleife wird das Ereignis mit der kleinsten Zeit aus der Liste entnommen, die Uhr auf diese Zeit gestellt[3] und die zugehörige Ereignisroutine aufgerufen. Werden hierbei neue Ereignisse erzeugt, dann werden sie entsprechend ihrer Ereigniszeit in die Liste eingegliedert. Existieren keine Ereignisse mehr in der Liste oder ist die maximale Simulationszeit erreicht, dann ist die Simulation beendet.

Soll eine verteilte Simulation durchgeführt werden, bei der eine Simulation aus mehreren Modellen besteht, dann muss dieses Konzept erweitert werden. Da jedes Modell über eine eigene Simulationszeit verfügt, müssen diese Modelle miteinander synchronisiert werden, um die Kausalordnung zwischen den auszuführenden Ereignissen entsprechend der vom Benutzer vorgegebenen Modellbeschreibung einzuhalten. Dies wird in der Realität dadurch erschwert, dass diese Kausalordnung

[3] Dies bildet die Simulationszeit.

nur implizit definiert ist und die Menge aller auszuführenden Ereignisse sich erst zur Laufzeit der Simulation ergibt. Hierzu gibt es verschiedene Realisierungsverfahren, bei denen grob zwischen *konservativ* und *optimistisch* zu unterscheiden ist [10, 47]:

- Konservative Verfahren stellen die oben beschriebene Kausalordnung vor der Ausführung der Ereignisse sicher. Hierbei wird die Tatsache ausgenutzt, dass ein Ereignis zum Zeitpunkt t niemals von einem Ereignis zur Zeit $t + n;\ mit\ n = 1, 2, 3, ...$ kausal abhängig sein kann. Unter dieser Voraussetzung schreitet die Simulation in der Simulationszeit nur dann voran, wenn sicher ist, dass alle relevanten Ereignisse bekannt sind, um die Größe des nächsten Zeitintervalls sicher zu bestimmen. Das Zeitintervall muss so gewählt werden, dass alle Ereignisse exakt zu dem richtigen Zeitpunkt verarbeitet werden können. Ziel ist, dass kein Ereignis auftreten kann, dessen Ereigniszeit zum Zeitpunkt der Verarbeitung vor dem aktuellen, diskreten Zeitpunkt der Simulationszeit liegt und somit die Kausalordnung verletzt.
  Das Erfüllen der oben beschriebenen Eigenschaft konservativer Verfahren ist unter Umständen schwer realisierbar oder führt zu ineffizienten Algorithmen, da oftmals unnötige Leernachrichten zur Zusicherung der Kausalität übertragen werden müssen. Weiterhin sind diese Algorithmen stark *deadlock*-gefährdet.

- Optimistische Verfahren setzen voraus, dass eine Verletzung der Kausalität nur selten auftritt. Somit ist die Strategie vorteilhaft anzunehmen, dass ein gewisser Simulationsschritt sicher durchgeführt werden kann. Stellt sich diese Annahme als fehlerhaft dar, so muss das System den letzten korrekten Zustand wiederherstellen (so genanntes Rollback) und wieder von diesem Zustand erneut beginnen.
  Diese Verfahren sind recht einfach zu implementieren, sind unempfindlich gegenüber Deadlocks und zeichnen sich durch eine hohe Geschwindigkeit aus, wenn die Annahme der geringen Wahrscheinlichkeit für die Notwendigkeit eines Rollbacks korrekt ist, was in der Regel der Fall ist.
  Beim häufig verwendeten Time-Warp Verfahren wird z.B. zu bestimmten Zeitpunkten der aktuelle Zustand der Simulation gespeichert und alle folgenden Ereignisse vor der Verarbeitung in eine so genannte Anti-Ereignisliste gesichert. Sollte zur Simulationszeit t ein Ereignis in die Warteschlange eingegliedert werden, welches die Ereigniszeit $t-n;\ mit\ n = 1, 2, 3, ...$ aufweist, so wird die Simulation in den gespeicherten Zustand zurückversetzt, der vor $t - n$ gültig war. Zusätzlich wird die Simulationszeit korrigiert und anschließend mit der Verarbeitung der gesicherten Ereignisse aus der Anti-Ereignisliste von diesem Zeitpunkt an neu begonnen.
  Aus technischen Gründen und um die benötigte Speichermenge für alte Zustände zu reduzieren, wird in der Praxis häufig eine Rollback-Zykluszeit definiert, welche die zeitliche Distanz zwischen den Sicherungspunkten definiert. Alle Informationen eines vorher gespeicherten Sicherungspunktes werden somit verworfen, wenn dieser Zeitpunkt erreicht wird. Diese Zykluszeit darf somit nicht zu kurz gewählt werden, damit nicht doch ein Ereignis vor dem Zeitpunkt der Sicherung auftreten kann. Andererseits darf sie auch nicht zu lang sein, da ansonsten ein Rollback zu viel Zeit benötigen würde.

Aus den hier beschriebenen Gründen sind in der Praxis in den meisten Fällen optimistische Simulationsverfahren anzutreffen, da sie sich durch eine hohe Geschwindigkeit bei gleichzeitig einfacher Implementierung auszeichnen.

2.4.2.2 Zeitgesteuerte Simulation (Time-Driven Simulation)

Bei der zeitgesteuerten Simulation schreitet die Simulationszeit in festen oder variablen Inkrementen Δ voran. Nach jeder Zeiterhöhung treten alle aufgetretenden Ereignisse mit einem Zeitstempel t zwischen dem letzten und dem aktuellen Zeitpunkt ($t \in (t_{aktuell} - \Delta,\ t_{aktuell}]$) in prinzipiell willkürlicher Reihenfolge auf. Das Inkrement Δ muss somit klein genug gewählt werden, damit das auftretende Ereignis nur Ereignisse in der simulierten Zukunft, nicht aber andere Ereignisse des gleichen Zeitintervalls beeinflussen kann. Da bei der zeitgesteuerten Simulation – anders als bei der ereignisgesteuerten Simulation – nicht möglich ist, Totzeiten automatisch zu überspringen, darf das Intervall aber nicht zu klein gewählt werden, weil sonst die Simulationsgeschwindigkeit negativ beeinflusst wird. Diese Eigenschaft stellt einen wesentlichen Nachteil der zeitgesteuerten Simulation dar [47].

Auf die zeitgesteuerte Simulation wird später in Kapitel 4.2.4 noch genauer eingegangen.

2.4.3 Hybride Simulation

Je nach Literatur wird unter hybrider Simulation entweder die Simulation von Systemen bezeichnet, die gleichzeitig aus diskreten und kontinuierlich simulierten Komponenten bestehen [60], oder die Simulation von Systemen, die mittels optimistischer und konservativer Verfahren realisiert werden [47]. Bei der letzteren Variante wird zwischen horizontal- und vertikal hybriden Verfahren unterschieden. Bei den horizontal hybriden Verfahren koexistieren optimistische und konservative Ansätze. Bei den vertikalen Verfahren werden Ansätze realisiert, die zwischen der optimistischen und der konservativen Verfahren angesiedelt sind.

Im ersten Fall – der Kopplung von diskreten und kontinuierlichen Komponenten in der Simulation – muss diese Kopplung in geeigneter Weise realisiert werden und gliedert sich in folgende, qualitativ unterschiedliche Varianten:

- Die lose Kopplung, welche vorliegt, wenn zu festgelegten und konstanten Zeitpunkten ein Datenaustausch zwischen den Modellen auftritt. Als Beispiel ist hier ein Analog-/ Digitalwandler zu nennen, welcher zu gewissen Zeitpunkten den ermittelten Digitalwert ausgibt.
- Bei einer engen Kopplung führt die Überschreitung eines Schwellwertes zu einem Ereignis im diskreten Modell.
- Bei der internen Kopplung lassen sich diskrete und kontinuierliche Anteile im Modell nicht mehr voneinander trennen. Zur Beschreibung solcher Systeme sind geeignete Modellierungssprachen und Auswertealgorithmen notwendig.

Die wesentlichen Problempunkte bei diesen Kopplungen liegen zum einen in der schwierigen Vorhersage auftretender Ereignisse bei Überschreiten eines Schwellwertes, was gerade bei konservativen Algorithmen berücksichtigt werden muss. Zum anderen ist es nötig, eine geeignete Umsetzung zwischen analogen und digitalen Signalen zu gewährleisten, da z.B. der sprunghafte Wechsel von diskreten Signalen nicht der Realität entspricht. Zur Lösung dieser Problematik werden in der Regel entsprechende Umsetzungsmodelle (z.B. Tiefpässe, oder Funktionen mit speziellen Flankenformen) zwischen den Komponenten realisiert.

2.4.4 Cosimulation

Soll ein System simuliert werden, welches mittels eines Multi-Language Ansatzes beschrieben wurde (siehe hierzu auch Kapitel 2.3.2), so wird die Verwendung einer so genannten Cosimulation notwendig.

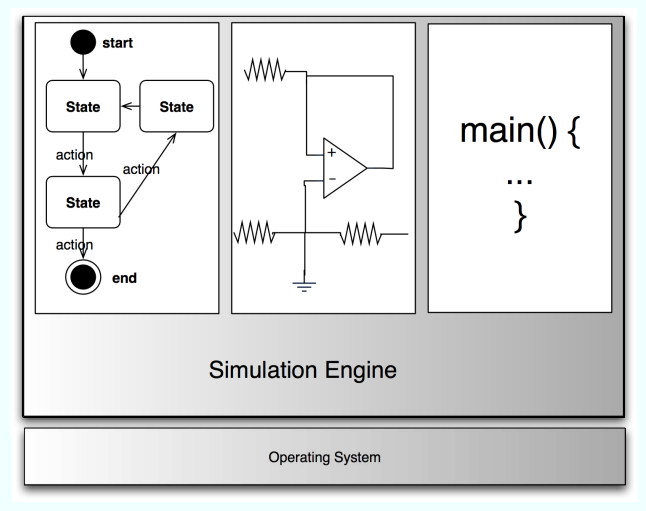
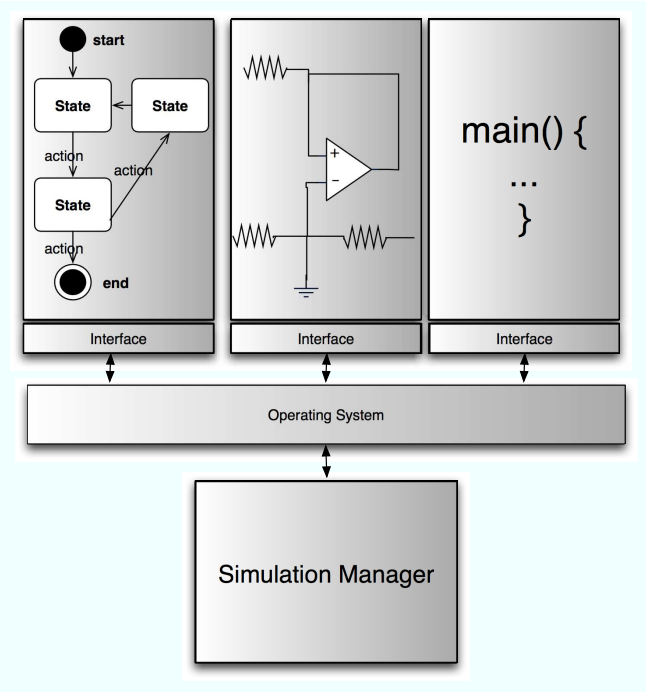


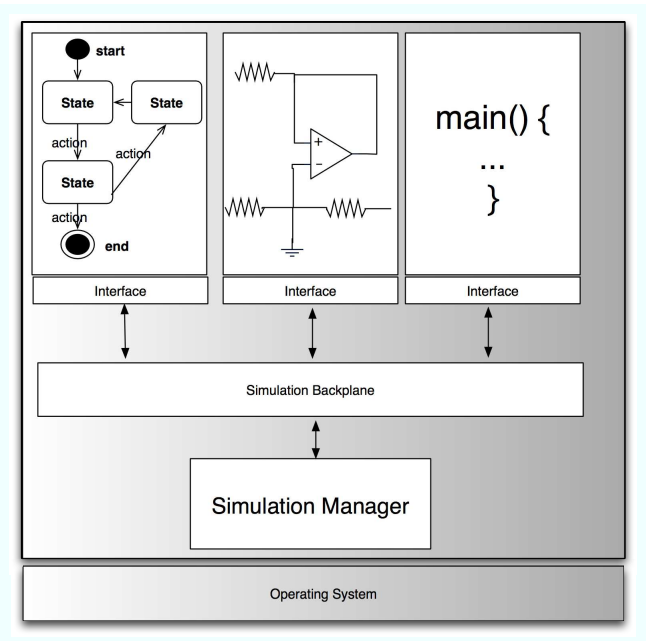
Abbildung 2.10: Single-Engine Simulation

Werden die verschiedenen Beschreibungssprachen in eine gemeinsame Zwischensprache umgesetzt, so kann das Gesamtsystem einfach mittels so genannter Single-Engine Cosimulation simuliert werden (siehe Abbildung 2.10), welche die verschiedenen Modelle in ein homogenes Simulationsmodell überführt und auf *einer* Simulations-Engine ausführt. Diese Simulationsart ist sehr schnell und effizient, da die verschiedenen Domänen optimal aufeinander abgestimmt werden können und – sollten analoge Komponenten existieren – eine Konvertierung der Signale zwischen diskreten und analogen Modellen automatisch realisierbar ist. Sollte die Abbildung einer beliebigen Sprache auf eine gemeinsame Zwischensprache aber nicht effizient möglich sein, so führt dies zur Verringerung der Ablaufgeschwindigkeit. Aufgrund der Tatsache, dass die verwendeten Zwischensprachen nicht standardisiert wurden und somit Tool- oder herstellerabhängig sind, führt dieser Ansatz zu einer aufwendigen Integration von Simulatoren und Modellen, die Fremdanbieter entwickelt haben. Somit bieten Simulatoren dieser Kategorie nur einen geringen Umfang an Domänen.

Wird keine gemeinsame Zwischensprache eingesetzt, so lässt sich das Konzept der Single-Engine Simulation nicht anwenden. In diesem Fall muss jedes Subsystem (bzw. jede Domäne) in eine eigene lauffähige Form überführt werden, welche mittels einer Multi-Engine Simulation simuliert wird. Diese lauffähigen Subsysteme sollen in diesem Zusammenhang als Simulations-Module bezeichnet werden. Zur Realisierung der Kopplung der Simulations-Module wird in heutigen Systemen eine so genannte Simulations-Backplane (siehe Abbildung 2.11 b) eingesetzt. Dieses Konzept ermöglicht eine deutlich höhere Simulationsgeschwindigkeit als bei einer Kopplung über das Betriebssystem (siehe Abbildung 2.11 a). Ein Simulations-Manager ist bei diesem Konzept für die Initialisierung der Module und für deren Synchronisierung und Kommunikation zuständig, denn nur der Manager besitzt das notwendige Wissen über die existierenden Verbindungen und die globale Simulationszeit.



(a) Multi-Engine Kopplung über Betriebssystem



(b) Multi-Engine Kopplung über Simulation Backplane

Abbildung 2.11: Multi-Engine Simulation

Aufgrund der Verwendung eines speziellen Interfaces zwischen den ausführbaren Simulations-Modulen und der Backplane bzw. dem Betriebssystem ist es einfach möglich, auch Modelle von Fremdanbietern an die Simulation anzuschließen, sogar, wenn sie nicht als Quellcode zur Verfügung stehen. Als Grundvoraussetzung ist lediglich die Möglichkeit des Austausches von Ereignissen und der Zeitsynchronisation über eine Programmierschnittstelle erforderlich [60].

2.4.5 Echtzeitsimulation

Soll eine Simulation mit einer echten Umgebung interagieren, so muss die prinzipielle Echtzeitfähigkeit der Simulation gewährleistet werden. Hierzu gibt es in der Praxis die Unterscheidung zwischen der so genannten weichen (soft real-time simulation) und der harten Echtzeitsimulation (hard real-time simulation). Gemeinsam ist beiden Formen die Notwendigkeit, dass eine Simulation so schnell durchzuführen ist, dass überhaupt eine Synchronizität zwischen der virtuellen und der realen Zeit möglich ist. Bei der weichen Echtzeitsimulation wird die Einhaltung bestimmter Zeitgrenzen lediglich im statistischen Mittel verlangt, was z.B. für Flugsimulatoren und andere, dem so genannten *Virtual Reality* Bereich zugehörige, Simulationen ausreicht. Bei der harten Echtzeitsimulation wird die garantierte Einhaltung bestimmter Zeitgrenzen verlangt, was somit eine zusätzliche und anspruchsvolle Anforderung bedeutet.

Für eine erfolgreiche harte Echtzeitsimulation müssen daher zusammenfassend die folgenden Eigenschaften garantiert werden:

- Die Simulationsgeschwindigkeit muss ausreichend hoch sein, damit der virtuelle Prototyp mit der geforderten Genauigkeit schneller als in Echtzeit simuliert werden kann.
- Die virtuelle Simulationszeit muss sich monoton und synchron zur Echtzeit erhöhen.
- Sollen reale Komponenten an der Simulation angeschlossen werden, so muss eine geeignete Schnittstelle zum Datenaustausch existieren.
- Alle virtuellen Komponenten müssen innerhalb fester Zeitgrenzen simuliert werden, wie es von der Definition von harter Echtzeit (hard real-time) entsprechend DIN-44300 gefordert wird.

Die erste Bedingung wird durch den Abstraktionsgrad der verwendeten Modelle – welcher direkten Einfluss auf die Genauigkeit der Modellierung hat – und der Rechenleistung des Simulationssystems bestimmt. Benötigt der Simulator die Zeit t_{real} um ein gegebenes Zeitintervall t_{sim} zu berechnen und t_{com} für die Kommunikation, so definiert sich der Realtime-Faktor R als $R = \frac{t_{sim}}{t_{real}+t_{com}}$, mit der Nebenbedingung $R \geq 1$ für Echtzeitsimulation. Da in der Simulation immer Zeit für die Kommunikation der Simulationskomponenten untereinander eingeplant werden muss und somit die Ausführungszeit t_{real}mit $t_{real} \leq t_{sim} - t_{com}$ notwendig wird, beschreibt dieser Faktor ebenfalls den minimalen Abstraktionsgrad und somit die Genauigkeitsgrenze, die bei einer bestimmten Rechner-Performance realisierbar ist (siehe hierzu auch 6).

Der zweite Punkt beschreibt eine Forderung bezüglich der Zeitsynchronisation. Nur zeitgesteuerte Algorithmen besitzen, im Gegensatz zu den ereignisgesteuerten Algorithmen, die hier geforderten Eigenschaften und sind für diese Art der Simulation geeignet.

Da Echtzeitsimulation nur bei einer Interaktion mit einer realen Umgebung sinnvoll ist, ergibt sich der dritte Punkt von selbst, wobei die Realisierung durchaus anspruchsvoll sein kann, wie in Kapitel 4.2.5 ausführlich diskutiert wird.

Der letzte Punkt betrifft nur die harte Echtzeitsimulation, bei der – wie bereits oben erwähnt – eine Einhaltung fester Zeitgrenzen verlangt wird. Dieser Punkt ist besonders wichtig, wenn die korrekte Funktion eines gemischt virtuell-realen Prototypen überprüft werden soll. Nur, wenn das Zeitverhalten der Echtzeitsimulation ausreichend genau gegenüber einem realen Pendant ist, sind die Simulationsergebnisse für den Entwickler reproduzier- und verwertbar.

2.4.5.1 Mixed-Reality Simulation:

Der Begriff „Mixed-Reality“ beschreibt in der Literatur grundsätzlich die Verbindung von virtueller- und „echter“ Realität zu einem Gesamtsystem. Diese Technologie wird bereits im Bereich der Simulation und Visualisation vielfältig eingesetzt, unter anderem in der Chemie [42], in der Ausbildung von Piloten (Flugsimulator) oder beim Training von Chirurgen, z.B. in der Mikrochirurgie. Diese Ansätze sollen dem Menschen Möglichkeiten bieten, Dinge zu begreifen, die normalerweise außerhalb ihrer Wahrnehmung sind, oder den Umgang mit Dingen und Situationen ermöglichen, die im Realfall eine Gefahr für den Lernenden oder für die Umwelt darstellen.

In den hier genannten Beispielen wird Mixed Reality als rückgekoppelte Interaktion (Force-Feedback) mit einer Umgebung verstanden, auf die der Mensch keinen direkten Zugriff hat oder bei der ein direkter Zugriff zu gefährlich wäre. Diese Methode wird in der Praxis ebenfalls unter dem Überbegriff „Virtual Reality“ subsumiert. Sie ist Bestandteil eines Entwurfsprozesses, bei dem ausschließlich das rein virtuelle System im Zentrum des Interesses liegt, wie z.B. bei der Verbindung von Molekülen in der Entwicklung von Medikamenten oder bei der Betrachtung eines Autodesigns in einem so genannten Cave.

Wird aber dieses gemischt virtuell-reale System selbst als das zu entwerfende Objekt verstanden, so kann es den problematischen Übergang von einem rein virtuellen Prototypen (siehe Abschnitt 2.2.5) zum realen Prototypen erleichtern. Ziel ist hier die Integration realer Komponenten, ohne die optimalen Möglichkeiten zur Beobachtung virtueller Systeme aufgeben zu müssen [34]. Diese Idee wird in dieser Arbeit tiefer verfolgt, indem der Ansatz des Virtual Prototyping mit dem des Rapid Prototyping verbunden wird, wie in Kapitel 3 näher dargelegt wird.

2.4.6 Emulation

Bei der Emulation handelt es sich genau genommen nicht um eine Simulationsmethode, sie ist aber mit der Simulation eng verwandt und soll der Vollständigkeit halber hier nicht unerwähnt bleiben.

Bei dieser Methode werden nicht Softwaremodelle erzeugt, die bestimmte Systeme (z.B. Mikrocontroller oder andere digitale Komponenten) auf einem Hostprozessor nachbilden, sondern es wird spezielle Hardware (typischerweise FPGAs) so programmiert, dass sie das Verhalten bestimmter Hardwarekomponenten nachbildet, also *emuliert*.

Der Vorteil liegt in der sehr hohen Ausführungsgeschwindigkeit gegenüber einer herkömmlichen Modellierung als Software-Modell. Der Nachteil liegt in der problematischen Beobachtbarkeit der intern im Modell ablaufenden Vorgänge und in den hohen Kosten solcher Systeme. Auch die

Modellierung gewisser Zeitmodelle ist schwierig, da die synchron getakteten Systeme nur selten in der Taktfrequenz beliebig änderbar sind.

Im praktischen Einsatz werden solche FPGA-Boards zusammen mit normalen Prozessorboards eingesetzt, um die oben beschriebenen Nachteile ausgleichen zu können. In diesem Fall werden meistens die digitalen Modellkomponenten für das FPGA synthetisiert und die softwarebasierten Modellkomponenten auf den Prozessoren ausgeführt.

2.5 Nutzung der Simulation beim Test von Prototypen

Ein traditioneller Ansatz zum Testen von Prototypen hat seinen Ursprung in der strikten Unterscheidung zwischen dem regelnden und dem geregelten Element oder einem Regler (Controller) und der zugehörigen Regelstrecke (Control Path). Wie bei der klassischen Regelungstechnik ist die Regelstrecke auf den Eingang des Reglers zurückgekoppelt und bildet somit einen Regelkreis (siehe Abbildung 2.12).

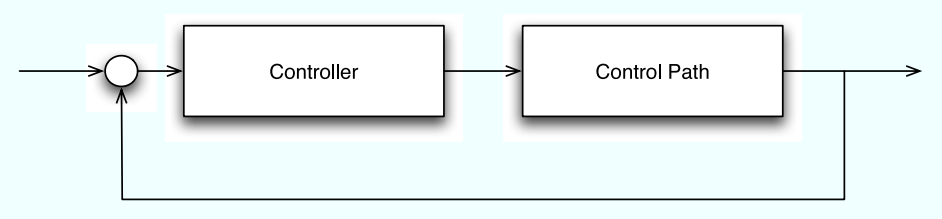


Abbildung 2.12: Regelkreis

Ein Regler ist in diesem Zusammenhang oftmals das zu testende System (SUT), kann aber auch ganz allgemein ein mathematischer Regelalgorithmus sein, der auf einer Protoypenumgebung ausgeführt wird. Die Regelstrecke selbst besteht möglicherweise aus einem oder mehreren Elementen (der Umgebung).

Entsprechend dem Ziel einer Testumgebung wird die Simulation eingesetzt, um entweder den komplexen Aufbau einer realen Testumgebung zu vermeiden oder um einen noch nicht fertiggestellten Prototypen zu überprüfen (siehe auch Kapitel 2.2.4)

Im Folgenden sollen die drei typischen Vertreter dieser Kategorie vorgestellt werden (siehe [58]).

- Hardware-in-the-loop (HIL):
  Wird die Regelstrecke als realer Regler an einer virtuellen Regelstrecke angeschlossen, so spricht man klassisch von einer Hardware-in-the-loop Simulation (siehe Abbildung 2.13).
  Ein typisches Beispiel für diesen Ansatz ist die Entwicklung von Raketensteuerungen. Hier simuliert ein Computer das Fluggebiet, die strömungstechnischen Eigenschaften der Rakete und ihre Flugbahn aufgrund der Stellwinkel des Leitwerks und der aktuellen Schubkraft des Antriebs. Der Regler beeinflusst die Einhaltung der Flugbahn und Geschwindigkeit für den korrekten Anflug auf das Ziel durch die Änderung des Schubs und der Änderung der Stellung des Leitwerkes.
  Ein weiteres klassisches Beispiel ist die Überprüfung einer realen ABS-Kontrolleinheit, wo

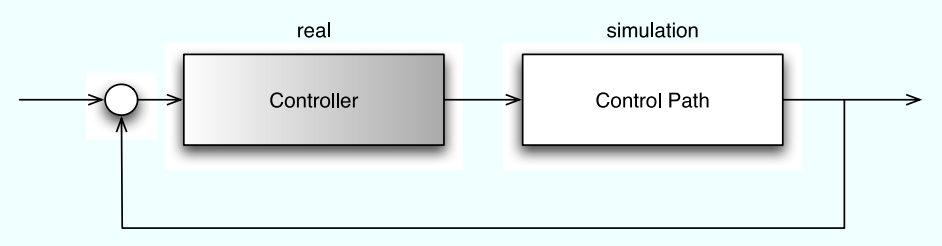


Abbildung 2.13: Hardware-in-the-loop

die Umgebung oder Regelstrecke die Simulation eines Fahrzeugs darstellt, die mit der Kontrolleinheit verbunden ist.

- Control prototyping:
  Eine weitere Möglichkeit besteht darin, den Regler zu simulieren und an eine reale Regelstrecke anzuschließen (siehe Abbildung 2.14). Dieser Ansatz kann z.B. in der Automobilindustrie vielfältig verfolgt werden, wenn ein ABS-Algorithmus überprüft werden soll. Hier wird nur der ABS-Regelalgorithmus auf einem speziellen Simulator ausgeführt und in ein echtes Automobil installiert, welches auf einer Teststrecke kontrollierten Tests ausgesetzt wird .

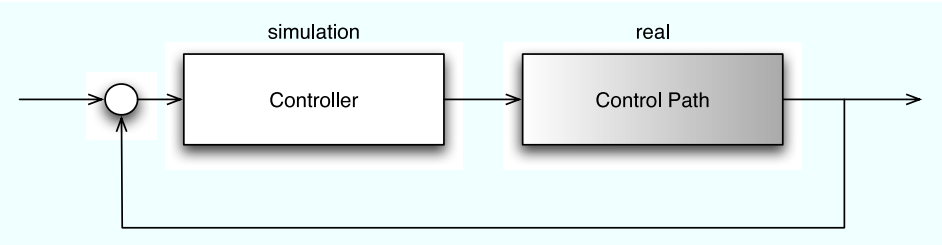


Abbildung 2.14: Control-prototyping

- Software-in-the-loop
  Wird die komplette Regelstrecke simuliert, so spricht man von einer Software-in-the-loop Simulation (siehe Abbildung 2.15).

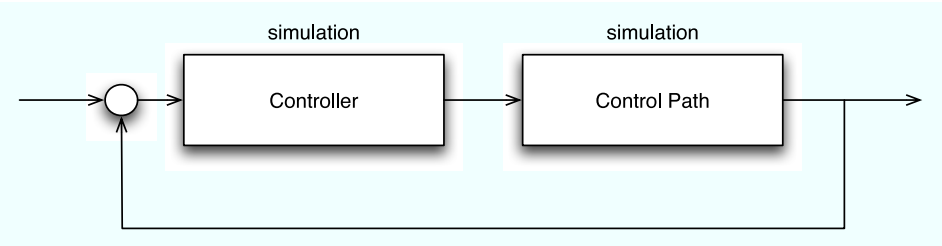


Abbildung 2.15: Software-in-the-loop

Heutige Systeme bestehen in der Regel aus vielen, gegenseitig rückgekoppelten Systemen. Somit ist die einfache Trennung zwischen Umgebung und dem regelnden Element nicht mehr aufrecht zu halten. Ein solches System entspricht eher der Abbildung 2.16, wobei die aktiven Elemente allgemein als Aktoren bezeichnet werden [31] und beliebig komplexe Graphen bilden können.

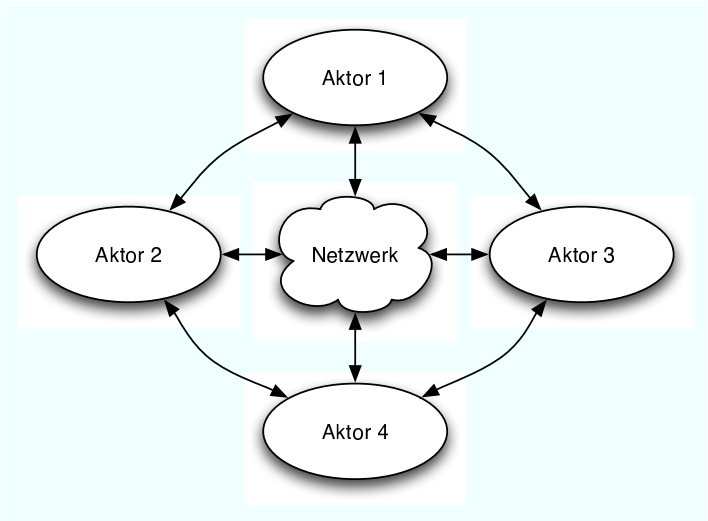


Abbildung 2.16: Beispiel eines komplexen rückgekoppelten Systems

Bei einem solchen System kann die Frage nicht mehr beantwortet werden, welches der Elemente eigentlich das regelnde oder das geregelte ist. Bei der Simulation besteht aber noch immer das Interesse, bestimmte Komponenten real und andere als Simulation darzustellen, was zu einem Mischsystem und somit zur Mixed-Reality-Simulation führt.

2.6 Diskussion und Zusammenfassung

Der in Abbildung 2.2 dargestellte optimale Entwurfsablauf skizziert zwei Stufen der Entwicklung:

1. Der Entwurf der Spezifikation, welcher keinen Bezug zu einer Implementation benötigt und
2. die Entscheidung und Implementation für eine Plattform, welche die in der Spezifikation definierten Randbedingungen (Constraints) einzuhalten hat.

In der ersten Stufe besteht das Hauptinteresse eines Entwicklers darin, mittels formaler Verifikation die Korrektheit der Spezifikation zu garantieren. Dies ist allerdings in vielen Fällen gerade für komplexe Systeme schwer zu realisieren, vor allem wenn es sich um Systeme handelt, die aus vielen verschiedenen Domänen (analog, digital, Software) bestehen. Somit muss sich der Entwickler in der Praxis mit einer Simulation behelfen, mit der er den spezifizierten Suchraum möglichst systematisch abdeckt.

Wäre die Welt perfekt, so könnte der Entwurf an dieser Stelle beendet werden. Leider gibt es vieles Gründe, warum eine vollautomatische Umsetzung der High-Level Spezifikation in eine reale

Implementation praktisch nicht möglich ist. Hierzu sind vor allem zwei Punkte herauszugreifen, die als besonders problematisch angesehen werden:

1. Es kann nicht ausgeschlossen werden, dass die Implementation Rückwirkungen auf das High-Level-Design hat. Zwar wird z.B. im Softwarebereich über targetspezifische Abstraktionsbibliotheken versucht, die Spezifikation korrekt auf die jeweilige Zielplattform abzubilden, indem die Eigenschaften gekapselt und vereinheitlicht werden. Allerdings kann bei eingebetteten Systemen, bei der die Hardware im größeren Maße Beschränkungen unterworfen sein kann (in ihren Eigenschaften und ihrer Leistung), nicht ausgeschlossen werden, dass bestimmte Spezifikationsanforderungen nicht realisiert werden können oder zu einer ineffizienten Nutzung der Hardware führen würden. Dies wäre z.B. der Fall, wenn Fließkommaberechnungen auf einer Hardware gefordert wären, die keine entsprechende Berechnungseinheit besäße, oder wenn ein Analog-/Digitalwandler mit einer bestimmten Genauigkeit verlangt wird, dieser aber nicht existierte.
   Um diese Probleme zu berücksichtigen, ist der Entwickler somit gezwungen, einen schrittweisen Top-Down-Entwurf von der Spezifikation zur konkreten Implementation durchzuführen und notfalls die Spezifikation iterativ zu ändern.

2. Als wesentlich problematischerer Grund für die unmögliche Realisierbarkeit einer vollautomatischen Implementation ist aber die Tatsache zu sehen, dass Codegeneratoren, HW-Synthese-Tools, Compiler, Bibliotheken und die Hardware prinzipiell als fehlerhaft angesehen werden müssen. Somit erfordert der Implementations- und Integrationsprozess ebenfalls hohe Aufmerksamkeit des Entwicklers und sollte in kleinen Schritten erfolgen.

Zur Realisierung der Implementation in kleinen Schritten gibt es, wie oben beschrieben, verschiedene Methoden, die für diese Aufgabe entwickelt wurden. Hierzu sind zwei wesentliche Methoden zu nennen, die sich nicht gegenseitig ausschließen, aber bisher nicht miteinander in einer Entwurfsmethode verbunden wurden: Das Rapid und das Virtual Prototyping. Während das Rapid Prototyping das Ziel verfolgt, möglichst früh die prinzipielle Umsetzbarkeit einer Spezifikation zu demonstrieren und diese gegenüber einer realen Umwelt zu überprüfen, versucht das Virtual Prototyping, das System mit seiner Umwelt im Computer zu modellieren und zu simulieren, um so zu einer korrekten Implementation zu kommen.

Wenn man sich beide Methoden genau anschaut, so kann man feststellen, dass die Schwächen der einen Methode in den meisten Punkten durch die Vorteile der anderen kompensiert werden:

Beim *Rapid Prototyping* besteht – je nach Realisierung – das Problem, dass die Spezifikation der Prototypen lediglich ausgeführt wird, aber keine Simulation der jeweils einzusetzenden Plattform stattfindet. Somit wird die Überprüfung des korrekten Timings oder der korrekten Systemperformance nur mit sehr viel Aufwand und evtl. hoher Ungenauigkeit möglich, wenn das Prototyping nicht schon auf der jeweiligen Zielhardware durchgeführt wird. Das würde aber der ursprünglichen Idee des *Rapid* Prototyping widersprechen. Die fehlende Virtualisierung der Umgebung kann sich bei komplexer Interaktion mit dieser als Problem darstellen, da für eine systematische und reproduzierbare Validierung die prinzipielle Zeitinvarianz eines Versuchs bei Vermeidung *ungewollter* Störeinflüsse unverzichtbar ist. Dies erfordert komplizierte und teure Testaufbauten, welche die Zeit bis zur Markteinführung (time-to-market) negativ beeinflussen.

In den hier genannten Punkten zeigt das *virtuelle Prototyping* deutlich seine Stärken: Die Berücksichtigung des Zielsystems ist schon im frühen Entwurfsprozess möglich, sofern ausreichend

validierte Simulationsmodelle zur Verfügung stehen. Die Fehleranalyse wird stark vereinfacht, da zu jedem Zeitpunkt das System angehalten und bis ins Detail analysiert werden kann. Die genaue Analyse der zeitlichen Grenzwerte ist ebenfalls jederzeit möglich, auch wenn noch Testroutinen und andere Elemente im System vorhanden sind, die später nicht mehr enthalten sein werden. Generell können alle Messeinflüsse exakt herausgerechnet werden, was einen erheblichen Vorteil gegenüber echten Systemen bedeutet. Da die Umgebung ebenfalls in der Simulation enthalten ist, ist die Forderung nach zeitlicher Invarianz und der Vermeidung ungewollter Störeinflüsse leicht zu realisieren und hilft somit die Kosten für einen Messaufbau und den für eine komplexe Messung nötigen Zeitaufwand zu vermeiden.

Der Nachteil des virtuellen Prototypen liegt in der Notwendigkeit der Verfügbarkeit ausreichend genauer Modelle, wodurch deren Komplexität deutlich erhöht und eine Validierung gegenüber dem realen Pendant notwendig wird. Der nötige Aufwand für die Erstellung solcher komplexen und genauen Modelle muss dem Aufwand für den realen Bau eines möglicherweise ungenügenden Prototypen und dem Aufwand einer komplexen, realen Testumgebung gegenüber gestellt werden. Je nach Aufgabe kann die Antwort hier unterschiedlich ausfallen.

Weiterhin ist natürlich nicht zu vergessen, dass auch das Ergebnis einer komplett virtuellen Entwicklung später als realer Prototyp einer Untersuchung und somit Messungen unterworfen werden muss, um so Fehler zu vermeiden, wie sie im Elch-Test bei der Mercedes A-Klasse berühmt wurden. Das Auto bestand diesen Test im Simulator, in der Realität kippte es auf die Seite. Dies wird umso komplizierter und teurer, je größer der Schritt von einem virtuellen zum realen Prototypen ist. Dies kann am Ende dazu führen, dass ein Testaufbau für einen realen Prototypen, der vorher komplett virtuell entwickelt wurde, genauso komplex und teuer ist, wie bei einer Entwicklung, bei der die Umgebung gleich real existiert. Das Ergebnis wäre, dass ein Teil der oben beschriebenen Vorteile des Virtual Prototyping wieder aufgehoben wird.

Somit stellt sich die Frage, wie die beiden Methoden miteinander kombiniert werden können, so dass möglichst die positiven Aspekte verstärkt und die negativen reduziert oder sogar aufgehoben werden. Die Lösung führt zur Entwurfsmethode des inkrementellen Entwurfablaufs. Diese Methode soll im folgenden Kapitel erläutert werden.

3 Lösungskonzept und Voraussetzungen

Wie im vorherigen Kapitel beschrieben wurde, existiert der Bedarf an einer Entwurfsmethode, die, ausgehend von einer abstrakten Spezifikation, inkrementell zu einem realen Prototypen führt. Dies soll unter Vermeidung von großen Entwicklungssprüngen und teuren, zeitaufwendigen und komplizierten Messungen an einem realen Prototypen ermöglicht werden. Weiterhin sollen alle komplexen Entwurfsschritte in einer virtuellen Umgebung erfolgen, da nur dort optimale Beobachtungsmöglichkeiten zur Verfügung stehen. Dies führt zu einem Entwurfsablauf, welcher sich – grob skizziert – folgerndermassen darstellt:

1. Erstellung einer abstrakten und ausführbaren Spezifikation.
2. Verifikation der abstrakten Spezifikation oder Validierung mittels Simulation.
3. Top-Down-Entwurf der Spezifikation, Modularisierung entsprechend der gewählten Zieldomänen.
4. Schrittweise Überführung des komplett virtuellen Prototypen in einen realen Prototypen.

Der 1. Punkt erfordert eine High-Level-Beschreibung des zu entwickelnden Systems, welche alle Forderungen und einzuhaltenden Bedingungen enthält. Diese Beschreibung muss vollständig sein und kann somit als Dokumentation und als Basis für die folgenden Schritte dienen. Für die Methoden zur Realisierung einer solchen Spezifikation wird an dieser Stelle auf andere Arbeiten verwiesen [7, 37, 53].

Dieser Entwurfsablauf soll im Folgenden weiter konkretisieret werden.

3.1 Der inkrementelle V/R-Entwurfsablauf

Ist die Spezifikation erstellt, so muss sie validiert oder verifiziert werden. Bei einer Validierung können Rapid-Prototyping-Systeme eingesetzt werden, falls die Überprüfung anhand einer realen Umgebung erfolgen soll. Wird dies nicht verlangt, so erfolgt die Validierung mittels einer zu Beginn möglicherweise einfachen und daher unpräzisen virtuellen Umgebung. Somit wird eine Überprüfung der Realisierbarkeit und der Korrektheit der Spezifikation ermöglicht. Treten hier Probleme auf, so muss die Spezifikation entsprechend angepasst werden.

Die Realisierung der Implementation beginnt mit der Aufteilung (der Partitionierung) des Gesamtsystems in einzelne Domänen. Anschließend folgt der Top-Down-Entwurf jeder einzelnen Domäne, bei dem mit zunehmender Genauigkeit der Implementation gleichzeitig ebenfalls die virtuelle Umgebung erweitert und präzisiert wird, um die Eigenschaften des gesamten virtuellen Systems zu konkretisieren. In einigen Fällen muss bei Erreichen einer Abstraktionsebene ebenfalls ein Wechsel

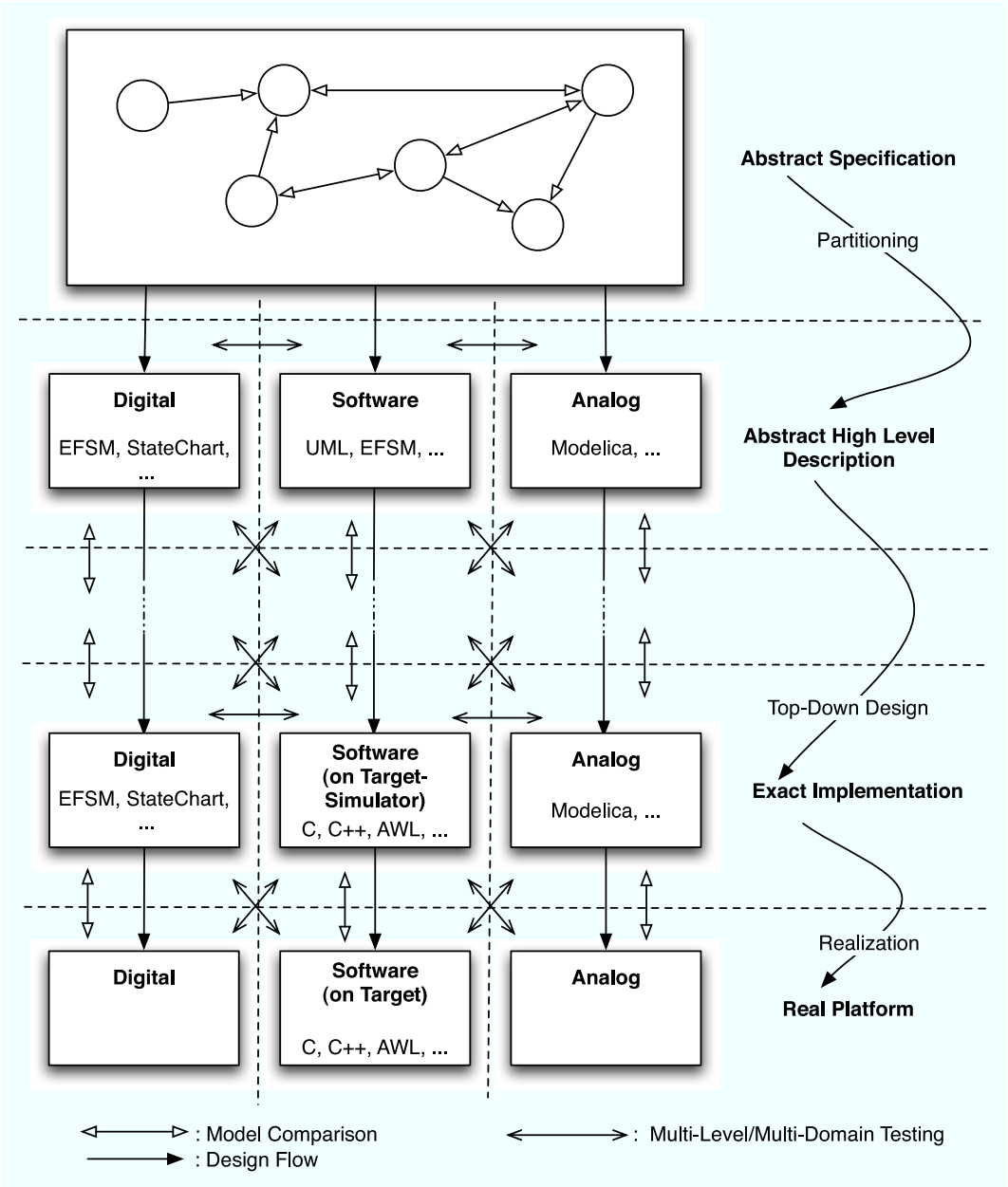


Abbildung 3.1: Top-Down Entwurfsablauf mit Multi-Domain Simulation

der Beschreibungsart innerhalb einer Domäne vollzogen werden, wie z.B. der Wechsel von einer Verhaltensbeschreibung mittels endlicher Automaten zu einer Beschreibung in C/C++ (siehe Abbildung 3.1). Jeder Wechsel dieser Art muss mittels automatischer Übersetzung durch Codegeneratoren oder – im schlimmsten Fall – durch manuelle Übersetzung erfolgen. In beiden Fällen muss dieser Prozess allerdings prinzipiell als fehlerbehaftet betrachtet werden!

Dem letzten Punkt – der schrittweisen Überführung des virtuellen Prototypen in die Realität – ist ebenfalls eine hohe Aufmerksamkeit zuzuschreiben. Hierbei wird das System, vom rein virtuellen Prototypen ausgehend, schrittweise über mehrere virtuell-reale Mischsysteme zum endgültigen realen Prototypen transformiert, indem die Verbindungs-Kanäle zwischen den Modellen an theoretisch beliebigen Stellen geschnitten werden, wie in Abbildung **3.2** schematisch dargestellt ist. Zur Verdeutlichung des Vorgangs ist in dieser Darstellung der virtuelle Prototyp inklusive seiner Umgebung als Aktor-Graph dargestellt. Die Verbindungen zwischen den Aktoren stellen die Kommunikationskanäle dar, die geschnitten werden, sollte ein Aktor in der Realität und einer in der virtuellen Umgebung miteinander kommunizieren.

Gelingt es, bei jedem Schritt die Äquivalenz des jetzigen und des vorherigen Modells zu gewährleisten, so kann die abschließende Validierung des vollständig realen Prototypen auf ein Minimum reduziert werden, um somit die hohen Testkosten des Rapid-Prototyping-Verfahrens zu vermeiden.

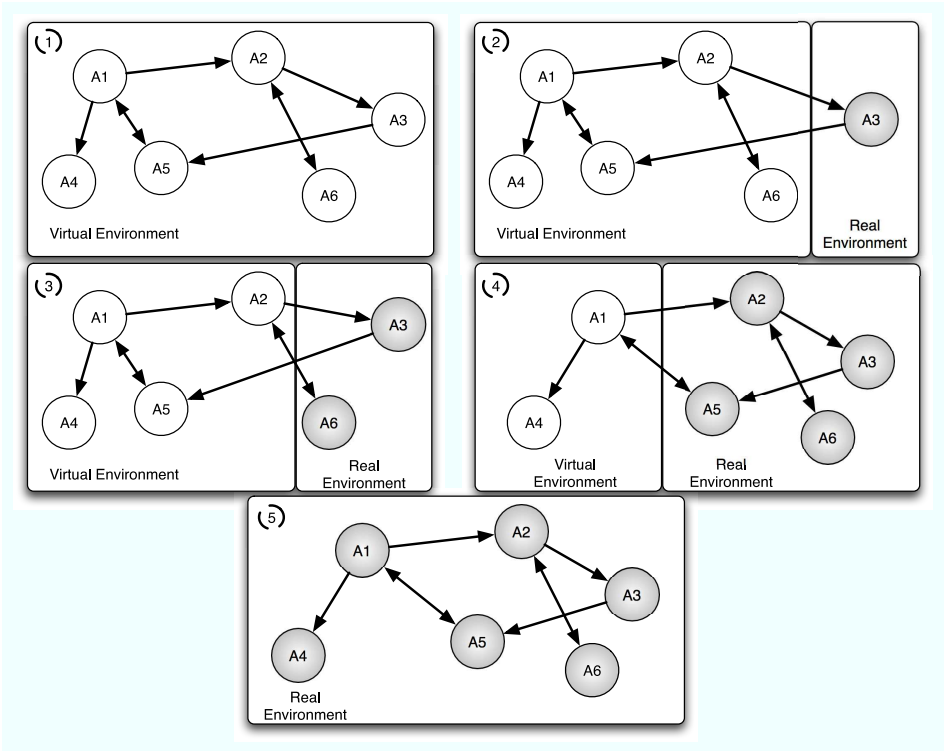


Abbildung 3.2: Inkrementeller Übergang von einem virtuellen zu einem realen Prototypen

Wie bei der Transformierung des virtuellen Prototypen in ein reales System ist auch beim Wechsel einer Beschreibungsart die Korrektheit des Vorgangs zu überprüfen, indem nach jedem Schritt ein Vergleich mit der vorherigen Modellbeschreibung erfolgt. Da ein transformiertes Modell nur schwer

direkt mit der vorherigen Variante verglichen werden kann (so genannter *White-Box Test*), wird die Äquivalenzanalyse an den Kommunikations-Kanälen der Modellgrenzen durchgeführt. Verhält sich das Modell bezüglich der Schnittstellenkommunikation zeitlich und funktional äquivalent zu einem anderen Modell, so sind die Modelle ebenfalls als äquivalent anzunehmen (so genanntes *Black-Box-Testen*). An dieser Stelle soll nicht unerwähnt bleiben, dass sich ein Black-Box-Test nicht eignet, um die Gleichheit zweier Modelle zu beweisen. Es ist lediglich in der Lage, innerhalb fest spezifizierter Grenzen eine Äquivalenzanalyse durchzuführen. Auch wenn dies gelingt, kann sich das Verhalten beider Modelle ausserhalb der Spezifikation komplett unterschiedlich verhalten. Aus diesem Grund ist bei den Tests die korrekte Auswahl der Testfälle sehr wichtig, was in der Praxis leicht gelingt, indem das Wissen des Entwicklers über das System in die Auswahl der Testfälle eingebracht wird (auch als Grey-Box-Test bezeichnet).

Dieses inkrementelle Vorgehen ist aber nur praktikabel, wenn das Modell schon zu einem frühen Entwurfszeitpunkt als korrekt oder ausreichend genau angenommen werden kann. Ist dies nicht der Fall, so ergibt sich die Gefahr, dass fehlerhafte Annahmen in der Modellierung von Schritt zu Schritt übertragen werden und erst bei dem abschließenden Vergleich mit dem realen System erkannt werden. Um dieses Problem in den Griff zu bekommen, muss mit Hilfe wechselseitiger Überprüfungen zwischen virtuellen und realen Modellen sicher gestellt werden, dass Modellierungsfehler zu einem möglichst frühen Zeitpunkt erkannt und korrigiert werden. Diese Methode soll im Folgenden *Cross Checking* genannt und näher beschrieben werden.

3.1.1 Modellvalidierung über Cross Checking

Die Funktionsweise des Cross- Checking soll an einem Beispiel erläutert werden:

Ein Entwickler soll die Software eines Speicherprogrammierbaren Systems (SPS) für eine Fabrikanlage entwickeln. Da zu Beginn diese Fabrikanlage noch nicht zur Verfügung steht oder ein direkter Zugriff auf die Anlage zu diesem Zeitpunkt zu gefährlich wäre, muss ein Modell dieser Anlange entwickelt werden, beispielsweise als UML-Statechart. Weiterhin soll das funktionale und zeitliche Verhalten der SPS-Steuerung zu Beginn ebenfalls als Statechart entwickelt werden, da eine Umsetzung auf hoher Abstraktionsebene das Verständnis der komplexen Abläufe erleichtert und somit beherrschbarer macht, als es bei einer sofortigen Programmierung im Assembler-ähnlichen AWL der Fall wäre. Erst anschließend soll das Programm in AWL-Code übersetzt werden, welches dann von der SPS direkt ausgeführt werden kann. In unserem Beispiel steht allerdings keine automatische Umsetzung von Statecharts in AWL zur Verfügung, wodurch das Risiko eines Implementationsfehlers bei der Übertragung der Steuerung besonders zu berücksichtigen ist.

Um diese Entwurfsaufgabe erfolgreich durchführen zu können, muss – wie oben erwähnt – parallel zur Programmierung der Steuerung auch die Umgebung und somit die Fabrikanlage entwickelt werden. Hierzu wird anhand der Konstruktionspläne der Anlage ein UML-Statechartmodell geschaffen, welches das zeitliche und funktionale Verhalten abbildet und die korrekten Sensorinformationen für die Steuerung erzeugt, bzw. auf Steuersignale korrekt reagiert. Obwohl das Modell mit bestem Wissen und hoher Sorgfalt erstellt wird, ist es an dieser Stelle unmöglich, sicher zu stellen, dass sich nicht irgendwelche Fehler im Modell befinden, so lange kein Abgleich mit der Realität stattgefunden hat. Die entwickelte Steuerung ist daher ebenfalls – da sie an diesem Modell getestet wird – als fehlerbehaftet anzusehen (siehe 3.3, 1. Schritt).

Die entwickelte Steuerung wurde gegenüber dem Modell der Fabrikanlage überprüft und muss jetzt gegenüber der realen Fabrikanlage ihre Funktionsfähigkeit beweisen. Hierzu wird im so genannten

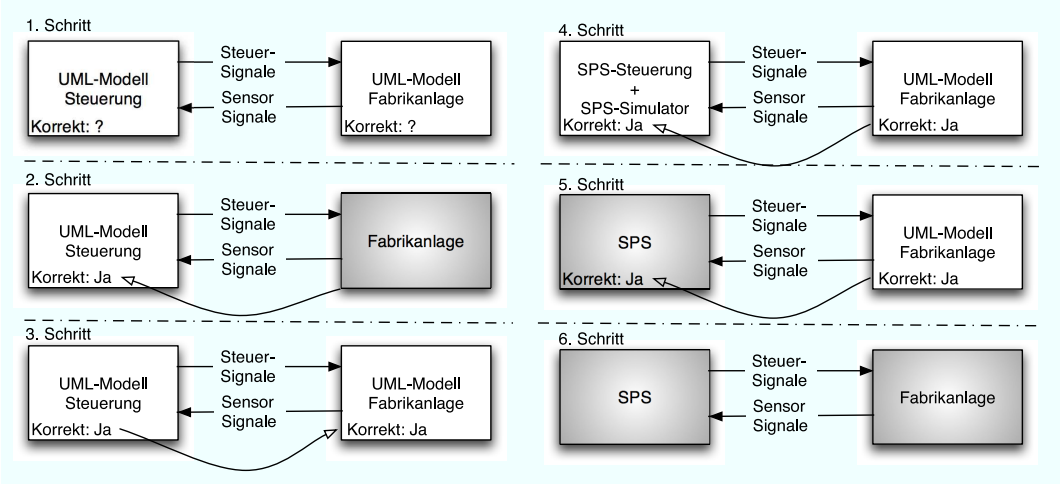


Abbildung 3.3: Beispiel des Cross Checking anhand der Steuerung einer Industrieanlage

Einrichtbetrieb die Verbindung zur Anlage aufgenommen, die Software systematisch getestet und beim Auffinden von Fehlern entsprechend korrigiert (siehe 3.3, 2. Schritt). An dieser Stelle hilft die Entwicklung als UML-Modell erheblich, da die Abläufe leichter zu verstehen sind und Fehler somit leichter diagnostiziert und korrigiert werden können. Am Ende kann die UML-Steuerung als korrekt validiert angesehen werden, wenn sie alle Tests bestanden hat.

Ziel der Entwicklung soll die Steuerung der Anlage durch eine SPS sein. Somit ist der Entwurfsablauf an dieser Stelle noch nicht abgeschlossen, da nur eine validierte Version als UML-Statechart entwickelt wurde. Gäbe es eine garantiert korrekte Umsetzung von einer Statechartbeschreibung zu AWL-Code, so wäre die Arbeit abgeschlossen. Da automatische und vor allem manuelle Übersetzungen fehlerhaft sein können, muss die Korrektheit der AWL-Beschreibung daher überprüft werden, bevor sie verwendet werden kann.

Prinzipiell könnte an dieser Stelle genauso vorgegangen werden, wie im Falle der UML-Steuerung: Die SPS würde schrittweise in die Anlage integriert und mittels Einrichtbetrieb überprüft. In diesem Fall hätten wir allerdings kaum einen Vorteil aus der vorherigen Überprüfung der UML-Version gezogen und müssten AWL-Code auf einer realen SPS an einer realen Fabrikanlage testen. Dies würde die Dauer des Entwurfsprozesses unnötig verlängern und wäre mit der traditionellen Rapid-Prototyping Methode vergleichbar. Ziel soll es aber sein, den virtuellen Entwurfsprozess weiter zu verwenden, um die Vorteile dieser Methode (gute Beobachtbarkeit, zeitliche Invarianz, hohe Testgeschwindigkeit, Vermeidung ungewollter Störeinflüsse) nutzen zu können.

Nachdem die UML-Steuerung erfolgreich validiert wurde, kann dieses Modell mit den zugehörigen, bei den Tests aufgezeichneten, Schnittstelleninformationen (Timing, Kommunikation) verwendet werden, um unsere UML-Version der Fabrikanlage ebenfalls zu validieren. Somit kann der benötigte Realitätsabgleich mit der realen Anlage einfach umgesetzt werden und dient als sicheres Fundament für die nächsten Entwicklungsschritte, die Programmierung der SPS-Steuerung in AWL (siehe 3.3, 3. Schritt).

Diese Implementierung wird in einem SPS-Simulator[1] ausgeführt, welches das funktionale sowie

[1] Hierbei handelt es sich um ein validiertes Modell aus der vorhandenen Bibliothek!

das zeitliche Verhalten der zu verwendenden realen SPS nachbildet. An dieser Stelle kann – neben der Validierung des Programms – genau überprüft werden, welches SPS-Modell eventuell für einen Einsatz in der Realität in Frage kommt oder ob das gewählte Modell bezüglich seiner Leistungsfähigkeit Probleme bereiten würde (siehe 3.3, 4. Schritt).

Nachdem das SPS-Programm anhand des überprüften Modells der Fabrikanlage validiert und die Entscheidung für eine zu verwendende SPS anhand der Performance-Analysen in der Simulation durchgeführt wurde, muss das Programm auf eine echte SPS geladen und ebenfalls anhand der virtuellen Fabrikanlage getestet werden. Dies garantiert, dass evtl. Modellierungsfehler in der virtuellen SPS nicht zu späteren Überraschungen führen (siehe 3.3, 5. Schritt).

Abschließend sollte die reale SPS anhand der realen Fabrikanlage korrekt funktionieren, ohne dass aufwendige Überprüfungen in dem realen System nötig wären (siehe 3.3, 6. Schritt).

Mit diesem Konzept können die Vorteile des Rapid Prototyping sowie des Virtual Prototyping miteinander effizient kombiniert werden unter Vermeidung der besprochenen Nachteile beider Methoden. Die wesentlichen Entwicklungsschritte und Tests finden innerhalb der virtuellen Teilumgebungen statt, was ihre einfache und schnelle Realisierbarkeit gewährleistet. Gleichzeitig ermöglicht eine ständige und gegenseitige Validierung der Modelle gegenüber der Realität die Vermeidung von hohen Kosten für komplexe abschließende Tests des vollständig realen Prototypen und der damit verbundenen Gefahr des zu späten Erkennens von Modellierungsfehlern.

3.2 Notwendige Voraussetzungen

Die oben beschriebene Entwurfsmethode benötigt eine Virtual-Prototyping-Simulationslösung, die das Konzept des Rapid Prototyping dahingehend erweitert, dass die Prototypen-Rechner nicht mehr nur das funktionale Verhalten eines Modells ausführen, sondern ein bezüglich des funktionalen sowie zeitlichen Verhaltens genaues Modell des geplanten Target-Systems. Soll Software entwickelt werden, so werden diese targetspezifischen Modelle eingesetzt, um sie auszuführen. Dies ermöglicht die Analyse der Laufzeiteigenschaften der Software bezüglich der gewählten Zielplattform, anstatt auf der Prototypenhardware lediglich die funktionale Überprüfung ohne Berücksichtigung der späteren Laufzeitumgebung durchzuführen.

Um die hier genannten Forderungen realisieren zu können, müssen zuvor einige grundlegende Voraussetzungen erfüllt sein. Hierzu gehören die Bedingung für die Durchführung einer Echtzeitsimulation, wie sie schon in Abschnitt 2.4.5 formuliert wurden. Diese sollen im Folgenden noch einmal aufgegriffen und für diesen Anwendungsfall konkretisiert und ergänzt werden:

- Eine geeignete Simulationsumgebung muss zur Verfügung stehen, die das Virtual Prototyping vollständig unterstützt. Es muss möglich sein, Modelle auf verschiedenen Abstraktionsebenen in ihrem funktionalen sowie Zeitverhalten zu beschreiben. Weiterhin muss die Simulationsumgebung möglichst exakte Modelle realer Komponenten zur Verfügung stellen (wie Mikroprozessoren, Busse usw.), die gegenüber den realen Pendants validiert wurden.

- Anders als bei rein virtuellen Prototypen muss die Simulation eines gemischten Prototyps, welcher aus realen sowie virtuellen Komponenten besteht, stetig und synchron zur realen Zeit in seiner virtuellen Simulationszeit fortschreiten.

- Die Simulationsumgebung muss in der Lage sein, die für den Entwicklungsprozess relevanten Teile des virtuellen Prototypen so schnell zu simulieren, dass eine korrekte Synchronisation der Laufzeiten zwischen realen und virtuellen Komponenten bei einem gemischten Prototypen gewährleistet bleibt. Dabei sind besonders die zusätzlichen Kommunikationslasten zu berücksichtigen.
- Je nach Anforderung muss harte oder weiche Echtzeit gewährleistet werden. Die Einhaltung muss überprüfbar sein.
- Die Verbindung von realen und virtuellen Komponenten zu einer gemischten Simulationsumgebung muss möglich sein. Die Einflüsse der Verbindungsschnittstellen dürfen nicht zu einer Verfälschung der Simulationsergebnisse führen. Neben der Möglichkeit eines Schnittes auf niedriger Abstraktionsebene (analoge/digitale Signale) ist ebenfalls eine geeignete Schnittmöglichkeit auf hoher Ebene (z.B. durch Bussysteme) vorzusehen.

Als Basis für die Erfüllung des ersten Punktes wurde das im Institut für Systems Engineering, System- und Rechnerarchitektur über viele Jahre entwickelte Simulationssystem ClearSim-MultiDomain gewählt, welches die zeitgenaue Simulation von virtuellen Prototypen ermöglicht, die entsprechend eines Multi-Language/Multi-Domänen Konzeptes realisiert werden [38, 49, 60]. Neben Modellierungssprachen wie UML-Statecharts und EFSMs zur Abbildung abstrakter Verhaltensmodelle, stehen bei diesem System ebenfalls konkrete Modelle realer Komponenten zur Verfügung, wie Mikrocontroller (Infineon C-167, C-505, ARM), Speicherprogrammierbare Systeme (SPS, Siemens-Simatik) und Bussysteme (wie dem CAN-Bus). All diese Modelle ermöglichen die Simulation nicht nur des funktionalen, sondern auch des Zeitverhaltens. Aufgrund der hohen Ausführungsgeschwindigkeit von ClearSim-MultiDomain erschien dieses System als am erfolgversprechendsten um die oben beschriebenen Forderungen umzusetzen. Weiterhin bestand ein kompletter Zugriff auf den Quellcode des Simulators, was die notwendigen Modifikationen für die Realisierung der weiteren Anforderungspunkte ermöglichte.

Die Forderung nach einer hohen Simulationsgeschwindigkeit ergibt sich aus der grundsätzlich geforderten Zeitsynchronität zwischen dem virtuellen und der realen Teil des Gesamtsystems. Kann ein Simulationsmodell nicht schnell genug ausgeführt werden, so ergibt sich hieraus ein permanentes Nachlaufen und somit inakzeptable Reaktionszeiten auf Ereignisse, die das virtuelle System erreichen, bis hin zu einem funktional fehlerhaften Verhalten. Die Synchronität zwischen virtueller und realer Zeit bedeutet zusätzlich, dass die virtuelle Zeit nicht beliebig in die Zukunft und schon gar nicht in die Vergangenheit springen darf. Dies hat Konsequenzen bezüglich der zu verwendenden Simulationsalgorithmen: weg von einem eventgesteuerten hin zu einem zeitgesteuerten Verfahren, welches – bei einer hier verwendeten diskreten Simulation – die Synchronität an diskreten Zeitpunkten gewährleistet (siehe hierzu Kapitel 4.2.4).

Weiterhin muss garantiert werden, dass die Reaktionszeiten auf äußere Ereignisse vorhersagbar bleiben. Dies stützt die Forderung nach einer harten Echtzeitsimulation, wobei in einigen Fällen durchaus eine weiche Echtzeitsimulation befriedigende Ergebnisse liefern mag. Da aufgrund der Komplexität eines hier modellierten Systems kaum eine mathematische Berechenbarkeit für die maximalen Zeitschranken möglich ist, muss eine inkrementelle Annäherung an den Optimalwert möglich sein. Dies kann durch eine Protokollierung der auftretenden Zeitkonstanten während der virtuellen Simulation geschehen, die anschließend im gemischten System weiterhin ihre Gültigkeit besitzen, da die virtuellen Modelle nicht verändert werden, was in Kapitel 4.2.4 näher erläutert wird.

Der letzte Punkt – die Realisierung einer Schnittstelle zwischen virtuellen und realen Systemteilen auf verschiedenen Abstraktionsebenen – beschreibt die Notwendigkeit nach einer physikalischen Verbindung dieser Welten, die z.B. über Digital-/Analogwandler und digitale I/O-Schnittstellen realisiert wird. Dass das Einbringen einer Schnittstelle mit zusätzlichen Latenzzeiten behaftet ist, lässt sich aus technischer Sicht nicht vermeiden, da die Wandlung von Events zu physikalischen Signalen immer Zeit benötigt. Diese Verzögerung muss vom Entwickler berücksichtigt und kritisch beurteilt werden (siehe hierzu Kapitel 8).

Wird ein System so geschnitten, dass im virtuellen sowie im realen Teil des Systems unterschiedliche Abstraktionsebenen aufzufinden sind, bedingt dies für die zeitgenaue Echtzeitsimulation eine besondere Berücksichtigung. Diese ist begründet in der Tatsache, dass ein Anstieg in der Abstraktion immer mit dem Verlust an Informationen verbunden ist, die aber eventuell wichtig für die korrekte Simulation sind. Schnitte dieser Art können nicht vermieden werden, wenn die verlangte Echtzeitfähigkeit eine Simulation auf niedrigerer Abstraktionsebene verbietet. Dieser Punkt wird in Kapitel 4.2.5 näher beleuchtet.

Abschließend soll ein wichtiger Aspekt erwähnt werden, der nur indirekt als eine der Voraussetzungen für die Realisierung der hier beschriebenen Entwurfsmethode bezeichnet werden kann, aber aus der Forderung nach harter Echzeitfähigkeit erwächst: Die Simulationsumgebung muss in ein hart-echtzeitfähiges Betriebssystem eingebettet werden. Da viele verschiedene Betriebssysteme existieren, ist es sinnvoll, eine Abstraktionsschicht einzuführen, welche die leichte Anpassung an die verschiedenen Systeme ermöglicht (siehe 2.1.4 und 4.2.1). In diesem Zusammenhang ist ebenfalls in dieser Abstraktionsschicht eine Middleware zu integrieren, die eine komfortable Kommunikation zwischen harter Echtzeitsimulation und nicht echtzeitfähigen Funktionen des Betriebssystems ermöglicht, um solche Funktionen wie Dateizugriffe oder Netzwerkkommunikation ohne Gefährdung der Echtzeitfähigkeit der Simulation zu ermöglichen. Auf diesen Punkt wird in Kaptitel 4.2.3 genauer eingegangen.

Im nächsten Kapitel soll die Realisierung der hier beschriebenen Voraussetzungen im Detail erläutert werden.

4 Realisierung der inkrementellen Entwurfsmethode

Wie im vorherigen Kapitel ausführlich erläutert wurde, stand als Basis für das zu entwickelnde Simulationssystem ClearSim-MultiDomain zur Verfügung. Da es sich hierbei um eine komplexe, in C++ geschriebene Software handelt, die ohne Berücksichtigung von Echtzeiteigenschaften entwickelt wurde, musste sie an vielen grundlegenden Punkten geändert und erweitert werden. Eine wesentliche Herausforderung bei den Änderungen bestand in der Wiederverwendung möglichst vieler der über Jahre entstandenen Programmquellen und Modelle, ohne sie in ihrer grundlegenden Struktur ändern oder sogar komplett neu schreiben zu müssen.

Zur Einführung soll im nächsten Abschnitt zunächst das Ausgangssystem beschrieben werden, gefolgt von einer ausführlichen Darlegung der Änderungen und Erweiterungen in dem System.

4.1 Cosimulationsumgebung ClearSim-MultiDomain

Stark vereinfacht besteht ClearsSim-MultiDomain aus einem Simulationsmanager (im Folgenden *Simulationskernel*, oder kurz *Kernel* genannt), einem oder mehreren Simulationseinheiten (*als Simulationsmodule* bezeichnet) und einer Nachrichtenkommunikation mit generischem Interface (*UPSI)* über das, neben dem reinen Datenaustausch, auch die zeitliche Synchronisation zwischen den Simulationskomponenten realisiert wird. Die Simulationsmodule und die zwischen ihnen aufgebauten Nachrichtenkanäle bilden einen gerichteten Aktor-Graph (siehe auch Abschnitt 2.1.3).

Das ganze System wird über eine Abstraktionsschicht gegenüber dem Host-Betriebssystem gekapselt, um eine leichte Portierbarkeit zu erreichen.

Auf die hier genannten Teilsysteme der Simulationsumgebung soll im Folgenden eingegangen werden, beginnend mit dem Simulationskernel und den Simulationsmodulen.

4.1.1 Der Simulations-Kernel und die Simulationsmodule

Der Simulations-Kernel hat die Aufgabe, den Datenaustausch und die zeitliche Synchronisation zwischen den Simulationsmodulen zu gewährleisten. Zur Koordinierung der Kommunikation zwischen den Simulationsmodulen wird der Kernel mit einer *Netzliste* beaufschlagt, welche die Verbindungen (oder *Kanäle*) zwischen den Simulationsmodulen beschreibt. Die Simulationsmodule selber sind ausführbare, nachrichtenbasierte Modelle beliebiger Art (in C, C++ oder Java programmiert). Sie werden vom Kernel gesteuert, indem sie Nachrichten von ihm erhalten, die entweder Daten oder Kommandos darstellen, wie z.B. die Aufforderung, einen gegebenen virtuellen Simulationszeitraum zu simulieren. Das Konzept ist mit dem Prinzip eines kooperativen Multithreadings zu vergleichen, in dem der Prozessor durch das Betriebssystem an einen Thread

übergeben wird, bis dieser wieder „freiwillig“ von diesem freigegeben wird (siehe Abbildung 4.1 und 4.2 (a)).

Während die Simulationsmodule den vom Kernel diktierten Simulationszeitraum simulieren, können jederzeit Ereignisse für andere angeschlossene Module auftreten, die sofort dem Kernel zur Weiterverarbeitung übergeben werden. Bei zueinander rückgekoppelten Modulen ist zu berücksichtigen, dass ausgesandte Ereignisse ein Folgeereignis in einem angeschlossenen Modul hervorrufen können, das wiederum für die eigene Simulation relevant sein könnte. Aus diesem Grunde muss das sendene Modul sofort nach dem Versand stoppen und dem Kernel-Scheduling die Entscheidung überlassen, welches Modul als nächstes zum Simulieren aufgefordert wird [12] (siehe Abbildung 4.2 (b)). Ist das Modul nicht rückgekoppelt, so muss es trotz Aussendung von Ereignissen bis zum Erreichen der vorgeschriebenen Endzeit die Simulation durchführen.

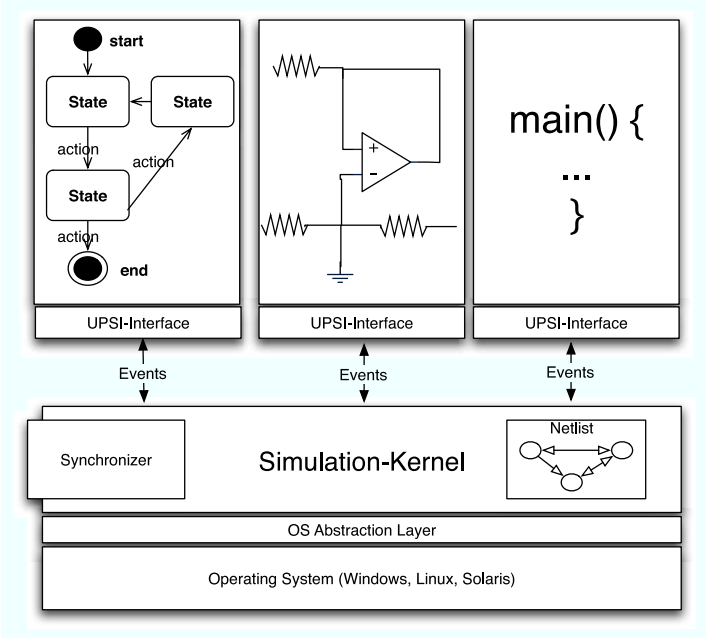
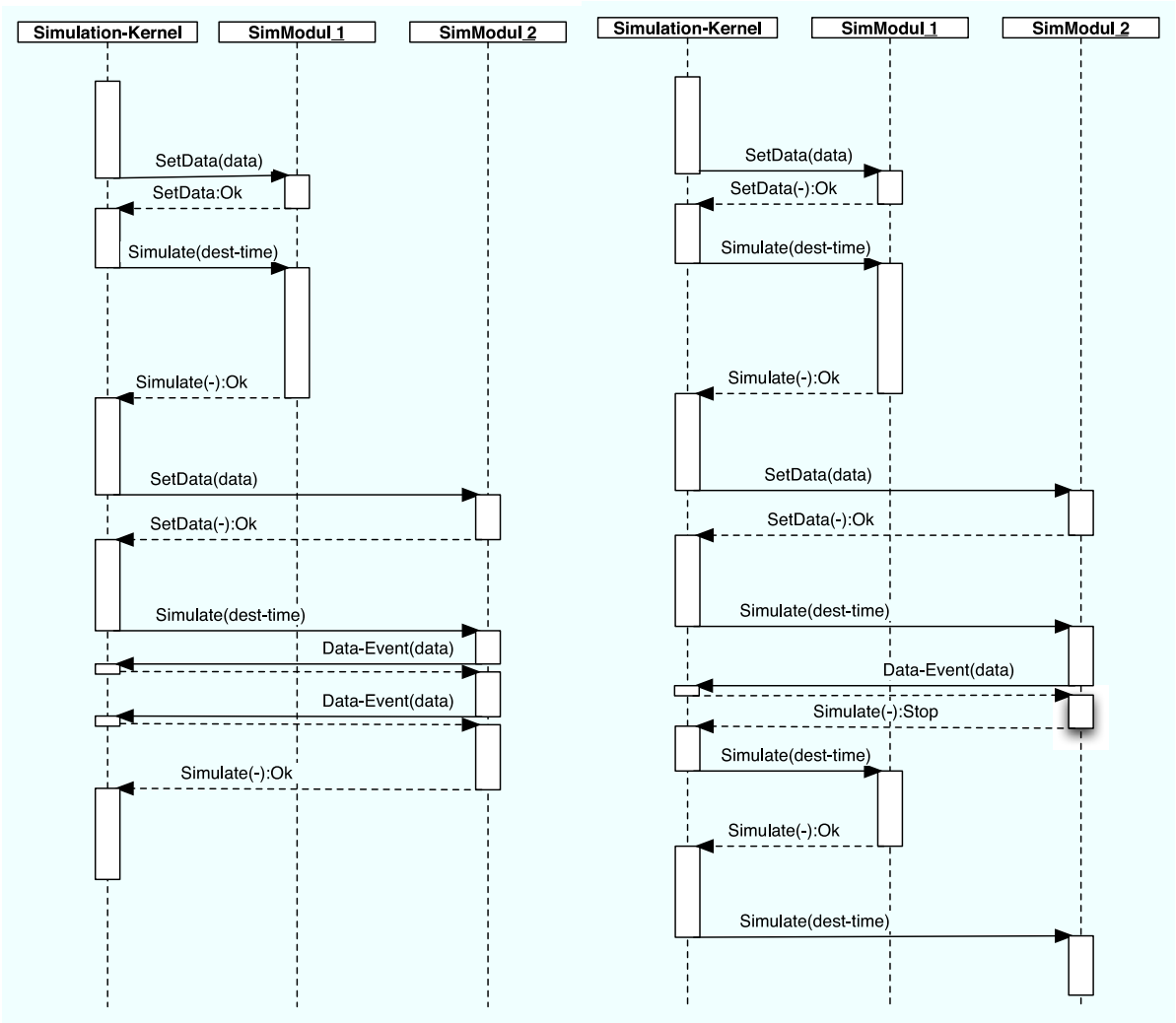


Abbildung 4.1: Grundsätzlicher Aufbau von ClearSim-MultiDomain

Die zeitliche Synchronisation erfolgt mittels einer zentralen Eventliste im Kernel durch einen so genannten *Synchronizer*. Dieser enthält den Simulationsalgorithmus und ist zur Startzeit der Simulation flexibel auswählbar. Er stellt somit die zentrale Komponente dar, die das zeitliche Verhalten des Simulators definiert. Bei rein virtueller Simulation wird ein ereignisgesteuerter Synchronizer verwendet, bei dem die fortschreitende Simulationszeit durch die in der Eventliste befindlichen Ereignisse definiert wird. Hierbei wird von dem Algorithmus automatisch dafür gesorgt, dass Simulationsmodule nur Ereignisse erhalten, deren Zeitstempel identisch mit dem der internen Uhr des empfangenden Moduls sind. Sollte das Ereignis aus der „Vergangenheit“ des Moduls stammen, wird – sofern das Modul einen entsprechenden Mechanismus unterstützt – ein Rollback, gefolgt von der Simulation zur korrekten Zeit, durchgeführt. Kommt ein Ereignis aus der virtuellen Zukunft, so wird das als Ziel definierte Simulationsmodul bis zu dieser Eventzeit simuliert, bevor ihm das Ereignis übergeben wird. Somit handelt es sich hierbei um die Realisierung eines optimistischen Time-Warp Verfahrens (siehe hierzu auch 2.4.2.1 und [60]). Da nicht alle Simulationsmodule die



(a) Scheduling nicht rückgekoppelter Simulationsmodule (b) Scheduling rückgekoppelter Simulationsmodule

Abbildung 4.2: Beispiele des Kernel-Scheduling

Rollbackfunktion beherrschen, besitzt der eventgesteuerte Algorithmus zusätzlich einen Mechanismus, um die hierheraus resultierenden Ungenauigkeiten zu minimieren. Hierbei wird die maximale Zeitdifferenz zwischen den Simulationsmodulen auf einen bestimmten Wert begrenzt. Dieses Vorgehen ist mit einem äquidistanten, zeitlichen Synchronisationspunkt vergleichbar und erinnert in diesem Aspekt an einen zeitgesteuerten Mechanismus, wie er in Abschnitt 2.4.2.2 definiert wurde.

Dieses Konzept ermöglicht eine zeitlich und funktional genaue und schnelle Simulation virtueller Prototypen, die aus vielen einzelnen Simulationsmodulen bestehen, welche jeweils für eine gewisse Teildomäne des Systems stehen, wie z.B. Mikrocontroller, Busse, analoge sowie digitale Komponenten, usw.

Zur Verbindung der Simulationsmodule mit dem Kernel wird ein generisches Interface eingesetzt, welches im nächsten Abschnitt näher beschrieben werden soll.

4.1.2 Das „Universal Portable Simulation Interface“ (UPSI)

Um die Integration prinzipiell beliebig gearteter Simulationsmodelle, inklusive kommerzieller Simulatoren, in die Gesamtsimulation eines virtuellen Prototypen zu ermöglichen, wurde das Universal Portable Simulation Interface (kurz: UPSI) entwickelt [60].

Als Grundvoraussetzung für die Integrierbarkeit in eine virtuelle Simulationsumgebung ist generell die Verfügbarkeit eines Interfaces zur Realisierung des Datenaustausches und zur zeitlichen Synchronisation zu nennen. Somit muss ein generisches Interface genau diese Minimalforderungen an einen Simulator oder ein Simulationsmodul stellen. Ist dieser in der Lage, diese Forderungen zu erfüllen, so steht prinzipiell der Integration nichts mehr im Wege. Je nach Art des Simulators sind allerdings unterschiedliche Ausprägungen der UPSI-Schnittstelle zu verwenden. Hierzu ist an erster Stelle zwischen aktiven und passiven Simulatoren zu unterscheiden:

- Lässt sich ein Simulator, von aussen im zeitlichen Ablauf steuern, so handelt es sich aus Sicht von ClearSim-MultiDomain um einen passiven Simulator. Der Simulationskernel kann also den Simulator komplett steuern, wie es bei den für ClearSim selbstprogrammierten Simulationsmodulen üblich ist.
- Ist ein zu integrierender Simulator nicht von aussen (im oben definierten Sinne) kontrollierbar, so handelt es sich um einen so genannten aktiven Simulator. In diesem Fall muss der Simulationskernel das Fortschreiten der Simulationszeit der aktiven Komponente als Maßstab für das Fortschreiten der Gesamtsimulation zu Grunde legen. Dies kann im Fall zu großer vorgegebener Zeitschritte unter Umständen zu Ungenauigkeiten bei der zeitlich korrekten Übertragung von Ereignissen zwischen den Simulationsmodulen führen. Auf dieses Problem soll hier allerdings nicht weiter eingegangen werden.

Neben der Unterscheidung zwischen aktiven und passiven Simulatoren ist noch die Art der Ausführung zu unterscheiden. Dies hat direkten Einfluss auf die zur Wahl stehenden Kommunikationsmöglichkeiten:

- Aktive Simulatoren werden als eigenständige Prozesse ausgeführt. Die Kommunikation mit dem Simulationskernel findet dabei über Shared-Memory oder Netzwerk-Sockets statt.

- Passive Module können prinzipiell als Prozesse, Threads oder einfach als Shared-Libraries realisiert werden. Die Shared-Library ist die häufigste Variante, bei der die Kommunikation mittels einer funktionalen Bindung (Kommunikation über Funktionsaufrufe) realisiert wird. Hierbei handelt es sich um die effizienteste Art der Bindung – sie stellt somit den Standardfall dar. In Sonderfällen, wenn z.B. eine Verteilung der Simulationslast auf verschiedene Prozessoren (Symmetrical Multiprocessing) oder sogar unterschiedliche Rechner (Parallel Execution) verlangt wird, so kann die Kommunikation im ersten Fall über Shared-Memory oder Netzwerk-Sockets und im zweiten Fall nur über Netzwerk-Sockets erfolgen.

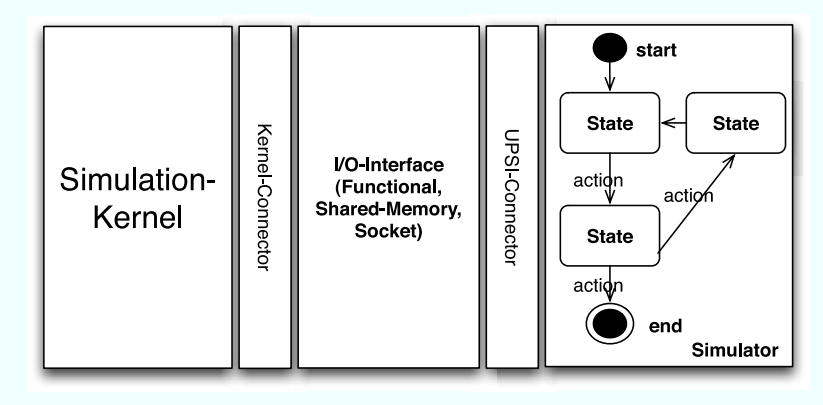


Abbildung 4.3: UPSI I/O Abstraktionsschicht

Ob ein Simulator aktiv oder passiv ist, wird vom Entwickler festgelegt und kann nicht geändert werden, wenn kein Zugriff auf den Quelltext der Implementation besteht. Lediglich die Art der Kommunikation kann im oben beschriebenen Rahmen durch Wahl der UPSI-Variante gewählt werden. Dies kann prinzipiell ohne Änderung am Simulator selbst durchgeführt werden, da die verschiedenen Variationen vom UPSI-Interface gekapselt werden (siehe Abbildung 4.3). Bei der Auswahl ist aus Performancegründen die folgende Reihenfolge zu empfehlen:

1. Die funktionale Kopplung ist mit dem geringsten Overhead und den kleinsten Latenzzeiten behaftet und ist somit immer zu bevorzugen, wenn es möglich ist. Im folgenden Teil der Arbeit wird lediglich diese Variante eingesetzt.

2. Die Shared-Memory Kommunikation zeichnet sich ebenfalls durch geringe Latenzzeiten aus. Da die Kommunikation aber über Semaphoren abgesichert sein muss, ist ein höherer Verwaltungs-Overhead zu verzeichnen.

3. Die Kommunikation über TCP/IP ist aufgrund der hohen Netzwerk-Latenzzeiten und des Verwaltungsaufwands beim Zerlegen und anschließendem Zusammensetzen von Nachrichten in netzwerkkonforme Formate (Byte-Arrays) als langsamste Kommunikationsform anzusehen.

Im nächsten Abschnitt soll die Betrachtung von ClearSim-MultiDomain mit der letzten Komponente, der Betriebssystem-Abstraktionsschicht, abgeschlossen werden.

4.1.3 Die Betriebssystem-Abstraktionsschicht

ClearSim-MultiDomain enthält eine Bibliothek, die eine Abstraktionsschicht zwischen dem Host-Betriebssystem und dem Simulations-Kernel mit seinen Simulationsmodulen realisiert. Als Grundlage dient hier das „write once, compile anywhere“ Konzept[1]. Es ist dadurch möglich, ein Simulationsmodul zu schreiben, welches ohne Änderung des Sourcodes auf den unterstützten Betriebssystemen wie Linux, Sun-Solaris und Windows kompilierbar ist. Kompilierte Module sind allerdings nicht unabhängig von der Plattform und somit nicht universell einsetzbar.

Diese Abstraktionsschicht ermöglicht einem Entwickler, ClearSim-MultiDomain sehr schnell auf die verschiedenen Betriebssysteme zu portieren, was für den Umbau zu einer echtzeitfähigen Simulationsumgebung eine wesentliche Hilfe darstellte, wie im folgenden Abschnitt dargelegt wird.

4.2 ClearSim-MultiDomain: gemischt virtuell-reale Systemsimulation

Wie im Abschnitt 3.2 beschrieben, waren einige Voraussetzungen zu erfüllen, um das in dieser Arbeit umzusetzende Konzept auch praktisch realisieren zu können. Wie dort erwähnt wird, war die erste Forderung nach einer geeigneten Simulationsumgebung mit der Nutzung von ClearSim-MultiDomain erfüllt. Somit basiert die folgende Entwicklung auf diesem System, welches mit dem in Abbildung 4.1 auf Seite 48 dargestellten strukturellen Aufbau vorlag.

Um eine saubere Trennung zwischen dem Simulationssystem und der notwendigen Echtzeitumgebung zu ermöglichen, wurde die Implementierung einer Middleware zur Vermittlung zwischen diesen Systemen realisiert, wie in Abschnitt 4.2.3 detailliert nachzulesen ist. Diese Middleware sollte weiterhin die Portierbarkeit auf verschiedene Echtzeitbetriebssysteme ermöglichen, um in Zukunft auch andere Plattformen unterstützen zu können.

Aufgrund der Tatsache, dass das verwendete Echtzeitbetriebsystem das Fundament für die hier durchzuführende Simulation darstellte und die Realisierung der Middleware im Detail ebenfalls von dieser Entscheidung abhing, soll im nächsten Abschnitt dieser Punkt weitergehend erläutert werden. Anschließend wird das Konzept der harten echtzeitfähigen Umgebung beschrieben, gefolgt von der Einführung, dem Aufbau und den Eigenschaften der realisierten Middleware. Am Ende soll mit der Beschreibung der Eigenschaften der zeitgesteuerten Systemsimulation und der Beschreibung der Schnittstelle zwischen virtuellen und realen Komponenten das Kapitel abgeschlossen werden.

4.2.1 Echtzeitbetriebssystem RTAI/Linux

Zu Beginn dieser Arbeit musste die Fragestellung beantwortet werden, welches Echtzeitbetriebssystem optimal für die zu erfüllenden Voraussetzungen sei. Da all die existierenden Varianten bezüglich ihrer Echtzeitfähigkeiten auf dem zu verwendenden Intel-PCs ähnliche Ergebnisse lieferten, wurden zusätzlich andere Kriterien zu Rate gezogen. Besonders zu berücksichtigen war die nötige Unterstützung des bereits existierenden und in C++ geschriebenen Simulators.

[1] Im Gegensatz zum „Compile once, run anywhere“ Konzept, dass durch die Java Laufzeitumgebung bekannt wurde!

Da C++ zu Beginn der Arbeit von keiner der existierenden Systeme offiziell unterstützt wurde, musste eine Veränderung der Echtzeitumgebung in Betracht gezogen werden, was zwangsläufig die Auswahl in Richtung quelloffener Lösungen einschränkte. Die in Frage kommenden Kandidaten waren somit RT-Linux und RTAI/Linux . Beide Lösungen erweitern den Linux-Kernel durch eine Hardware-Abstraktionsschicht, welche die auftretenden Interrupts virtualisiert und den Linux-Kernel als Idle-Task im Real-Time Scheduler ausführt (siehe Abbildung 4.4) [8].

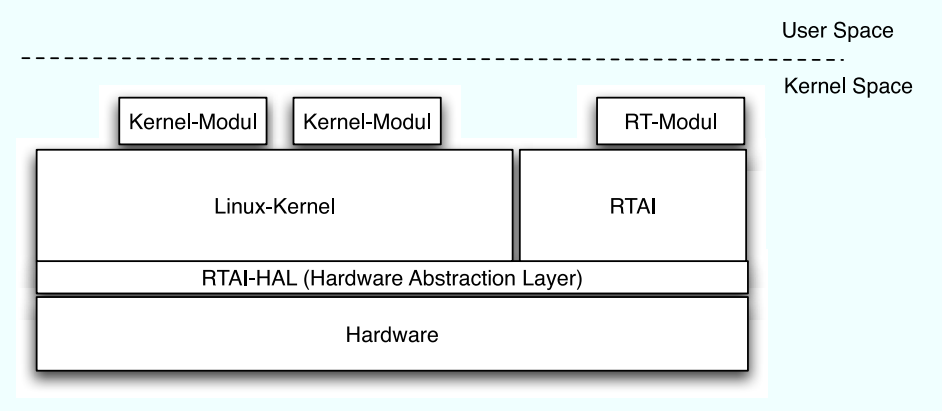


Abbildung 4.4: Prinzipieller Aufbau von RTAI/Linux

Beide Systeme ermöglichen die Ausführung von Echtzeitprozessen im Adressraum des Linux-Kernels (dem so genannten Kernel-Space), indem die vom RT-Scheduler auszuführende Software als Kernelmodul kompiliert und somit dynamisch eingebunden werden kann. RTAI/Linux unterstützt mittels der Erweiterung „LXRT“ zusätzlich die Option, Echtzeitprozesse ebenfalls im so genannten User-Space auszuführen zu können (siehe Abbildung 4.5). Nur der User-Space ermöglicht die Ausführung komplexer C++ Programme in einer Echtzeitumgebung, da Linux-Kernelmodule prinzipiell nur aus C-Programmen erzeugt werden können. Dieser Unterschied gab den Ausschlag zur endgültigen Entscheidung, RTAI/Linux einzusetzen.

Diese Entscheidung führte zu dem Entwurf einer echtzeitfähigen Simulationsumgebung, wie im nächsten Abschnitt erläutert wird.

4.2.2 Hart-Echtzeitfähige Umgebung

Eine zu berücksichtigende Folge aus der Entscheidung zur Nutzung von RTAI/Linux war die in Abschnitt 2.1.4 schon diskutierte Problematik der Dualität dieser Umgebung. Der Entwickler hat dafür Sorge zu tragen, dass innerhalb der harten Echtzeitumgebung nicht versehentlich Betriebssystemzugriffe erfolgen, die den Linux-Kernel aufrufen und nicht das Real-Time Subsystem (im weiteren als RT-Kernel bezeichnet). Da der Simulationskernel und die zugehörigen Simulationsmodule von ClearSim-MultiDomain nicht unter dem Blickpunkt einer möglichen Ausführung unter harten Echtzeitbedingungen entwickelt wurden und somit beliebige Bildschirmausgaben, Datei- sowie Netzwerkzugriffe erlaubt waren, musste eine Lösung geschaffen werden, die diese Zugriffe korrekt gestaltet. Zur Reduzierung des Aufwands musste eine geeignete Lösung diese Umgestaltung möglichst automatisch oder zumindest ohne wesentliche Neuprogrammierungen ermöglichen (siehe Abbildung 4.6).

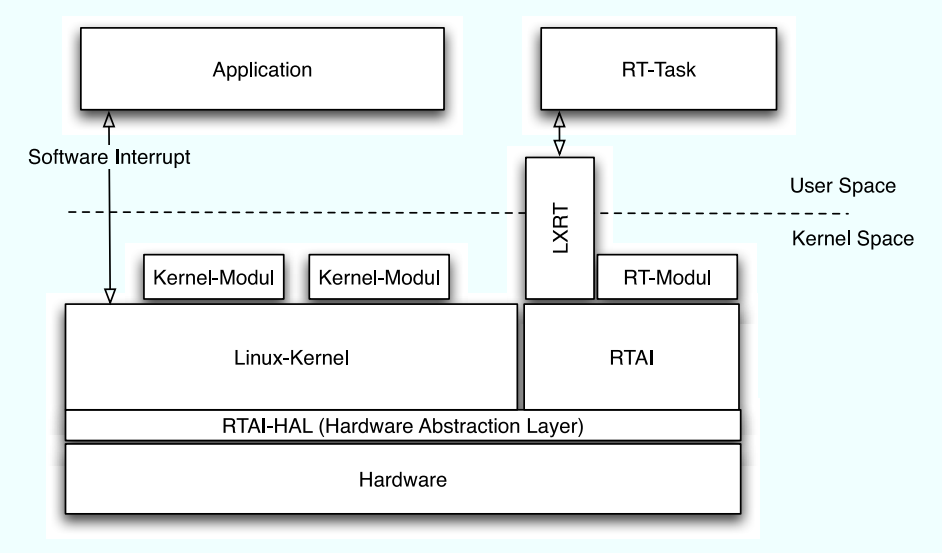


Abbildung 4.5: Aufbau der Echtzeiterweiterung RTAI/Linux mit LXRT

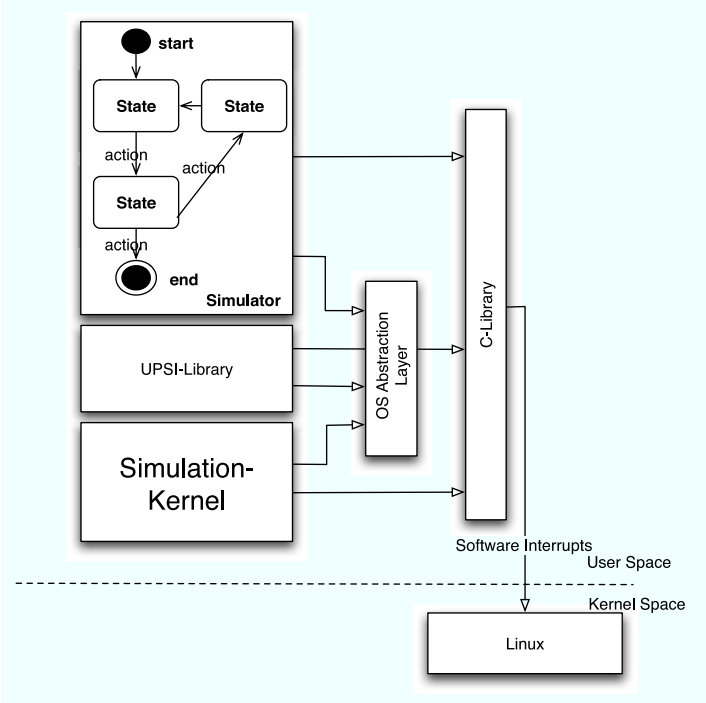


Abbildung 4.6: Darstellung der Systemzugriffe durch ClearSim-MultiDomain

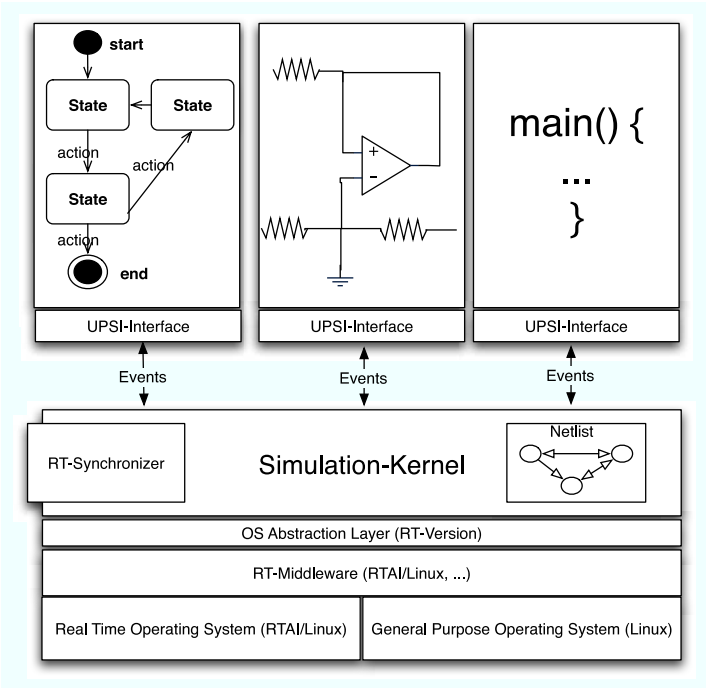


Abbildung 4.7: Integration von ClearSim-MultiDomain in ein RT-Betriebssystem

Auch wenn die Entscheidung für ein Echtzeitbetriebssystem getroffen wurde, schien es angebracht, die Unterstützung anderer Betriebssysteme für die Zukunft weiterhin möglichst einfach realisierbar zu halten, was zu einer zusätzlichen Forderung nach der Kapselung der Funktionen von RTAI/Linux durch eine Abstraktionsschicht führte.

Da schon eine Betriebssystem-Abstraktionsschicht in ClearSim-MultiDomain existierte, war es naheliegend, diese so zu erweitern, dass sie nicht mehr direkt auf das Host-Betriebssystem aufsetzt, sondern stattdessen auf eine geeignete Zwischenschicht (im Folgenden als Middleware bezeichnet), die für eine korrekte Zuweisung der Aufrufe zur Echtzeiterweiterung oder dem Linux-Kernel sorgt (siehe Abbildung 4.7).

Auf de ersten Blick mag die Entscheidung zum Einsatz einer zusätzlichen Schicht verwundern. Um bei der hier zu beherrschenden Komplexität die Wartbarkeit und Erweiterbarkeit des Systems nicht zu opfern, musste eine logische Trennung zwischen den hochsprachlichen und simulationsbezogenen Aspekten des Simulators sowie den Basiselementen des Betriebssystems realisiert werden. Dieses Konzept entspricht in seiner Grundidee einer Microkernel-Architektur, wie sie bei den verschiedensten Betriebssystemen zur Verbesserung der Portierbarkeit und Erhöhung der Modularität eingesetzt wird. Um z.B. eine andere Echtzeiterweiterung für Linux zu unterstützen, wäre es ausschließlich nötig, die neue und relativ kleine RT-Abstraktionsschicht anzupassen, anstatt die komplexe OS-Abstraktionsschicht komplett verändern zu müssen.

Wie oben erwähnt, bestand die Herausforderung in der korrekte Portierung der existierenden Simulationsumgebung, so dass die Aufrufe des Betriebssystems nur noch ohne Gefährdung der Echtzeiteigenschaften durchgeführt werden (siehe Abbildung 4.8) .

In diesem Zusammenhang war die Änderung und Erweiterung der ursprünglich in ClearSim-Multi-

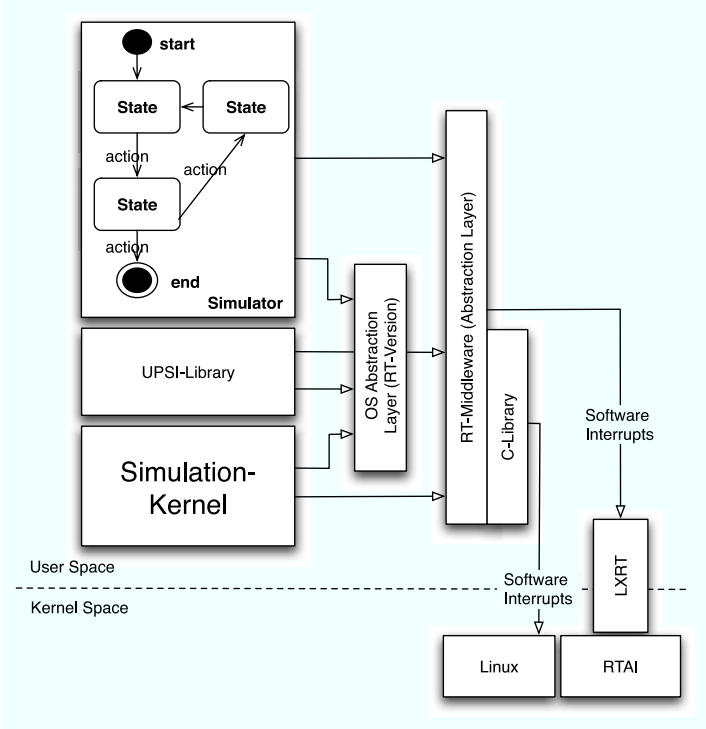


Abbildung 4.8: Betriebssystemzugriffe werden korrekt über die RT-Middleware geleitet.

Domain vorhandenen Abstraktionsschicht alleine nicht ausreichend, um eine vollständige Anpassung des Programmcodes an ein Echtzeitbetriebssystem zu vollziehen, da nicht alle Zugriffe über diese Schicht geleitet wurden. Gefahrenpunkte waren hier Zugriffe auf die C-Library, die von den Programmierern bisher ohne Einschränkung nutzbar war. Zusätzlich gefährdete die Anforderung von Speicher über die C-Funktionen „malloc()“ und „free()“, sowie über die C++ Operatoren „new“ und „delete“ das Echtzeitsystem, da der so angeforderte Speicher im Normalfall ebenfalls über die C-Library abgewickelt wird. All diese Zugriffe erzeugen Betriebssystemzugriffe, die zu nicht deterministischem Zeitverhalten des Echtzeitsystems führen können und somit die geforderten Zeitschranken gefährden.

All diese problematischen Funktionen und Operatoren sollten möglichst automatisch auf die Middleware abgebildet und somit echtzeitkonform realisiert werden. Hierzu wurden – soweit möglich – die Fähigkeiten des C++ Präprozessors genutzt:

- C-Funktionen wurden über spezielle Header-Files auf die korrekten, echtzeitfähigen Funktionen umgeleitet, wie z.B. „printf()“ zu „RTLib_IO_Printf()“ oder „malloc()“ zu „rt_malloc()“
- C++ Klassen, die Datei oder Netzwerkzugriffe ermöglichen, wie z.B. die Stream-Klassen, wurden neu implementiert und ebenfalls über spezielle Header-Files durch die herkömmlichen ersetzt.
- Alle implementierten C++ Klassen im Simulator müssen von einer speziellen Klasse abgeleitet werden, die eine echtzeitkonforme Realisierung von „new“ und „delete“ an ihre Nachkommen vererbt. Die Veränderung der Klassen muss manuell erfolgen.

- Die Klassen der beliebten Standard Template Library (STL) verwenden ein Allocator-Template für die Anforderung von Speicher. Dieses wurde durch die eigene, echtzeitkonformen Version ersetzt.

Mittels dieser Mechanismen war es möglich, ClearSim-MultiDomain, mit ungefähr 30.000 Zeilen Code im Basissystem, innerhalb weniger Monate komplett zu einem echtzeitkonformen Simulationssystem zu wandeln, nachdem die Grundvorraussetzungen dafür geschaffen wurden. Da die automatischen Änderungsmechanismen nicht in allen Fällen greifen, war bei einer Portierung die manuelle Begutachtung der Software weiterhin nötig, konnte aber auf bestimmte bekannte Problempunkte reduziert werden. Zusätzlich wurden spezielle Debugging-Tools verwendet, welche z.B. den Speicherzugriff im System protokollieren und somit den Entwickler erkennen lassen, ob an irgendeiner Stelle versehentlich auf den nichtdeterministischen Speicher zugegriffen wird.

Die Umstellung selbst eines so komplexen Simulationsmoduls wie des SPS-Simulators mit 19.000 oder der Mikrocontroller C505 und C167, mit jeweils mehr als 9000 Zeilen Programmcode waren jeweils innerhalb von zwei bis drei Manntagen erfolgreich durchzuführen und zu validieren, was die Leistungsfähigkeit dieses Systems eindrucksvoll demonstriert.

Der so modifizierte Code greift auf die Middleware zu, die im nächsten Abschnitt im Detail beschrieben werden soll.

4.2.3 Middleware für allgemeine Echtzeitbetriebssysteme

Wie in den vorherigen Abschnitten beschrieben, mussten alle Zugriffe des Linux-Kernels und der verwendeten Simulationsmodule auf den RT-Kernel umgeleitet werden, damit sie keine Gefährdung der vorgegebenen Zeitschranken darstellten. Gleichzeitig sollte eine Abstraktion der verwendeten Echtzeiterweiterung RTAI/Linux dafür sorgen, dass in Zukunft bei Bedarf leicht eine Portierung auf ein anderes System möglich bleibt.

Um dies zu gewährleisten, wurden zu Beginn grundlegende Funktionen der Echtzeitumgebung identifiziert, die einerseits nicht zu speziell waren, so dass sie von bestimmten zukünftigen Echtzeitumgebungen eventuell nicht zur Verfügung gestellt werden können, andererseits aber umfangreich und komplex genug, um so die existierende Codebasis von ClearSim mit möglichst wenig strukturellen Änderungen auf die neue Middleware abzubilden. Somit entstand eine Lösung, welche die folgenden Bereiche umfasst:

- Systeminitialisierung
- Semaphoren-Handling
- Thread-Handling
- I/O-Management (Datei-, Socket-, Bildschirmzugriff)
- Memory-Management

Der Bereich „Systeminitialisierung“ umfasst die grundlegenden Bereiche, die zur Einrichtung und zum Betrieb eines Echtzeitsystems nötig sind. Hierzu gehört im Grunde auch die Aktivierung

der Timer und die in RTAI/Linux implementierte Umschaltung zwischen harten und weichen Echtzeitprozessen, was nicht bei allen Echtzeitbetriebsystemen nötig ist und somit in diesen Fällen einfach durch leere Funktionen ersetzt werden kann.

Das „Semaphoren-Handling“ und „Thread-Handling“ stellt jeweils Funktionen zur Verfügung, die alle Echtzeitsysteme zur Verfügung stellen müssen. Leider besitzen die meisten vorhandenen Umgebungen hierzu mehr oder weniger stark unterschiedliche Lösungen, die sich zwar zumeist an POSIX [54] anlehnen, aber sich im Detail deutlich unterscheiden. Somit generalisiert die hier eingeführte Middleware die verschiedenen Varianten zu denjenigen Standardfällen, die ClearSim-MultiDomain benötigt und von allen existierenden Echtzeitumgebungen zur Verfügung gestellt werden können.

Das I/O-Management und Memory-Management wurde von RTAI/Linux und anderen Umgebungen nicht oder teilweise in einer hier ungewünschten Form zur Verfügung gestellt. Beide sind aber essentiell für die Lauffähigkeit von ClearSim-MultiDomain. Aus diesem Grunde musste eine allgemeingültige Lösung gefunden werden, die einerseits komfortabel nutzbar, aber andererseits auch auf jedem System realisierbar ist. Auf diese Punkte soll in den nächsten Abschnitten detailliert eingegangen werden.

4.2.3.1 Realisierung des I/O-Managements in der RT-Middleware

Die verfügbaren Betriebssystemerweiterungen sind teileweise sehr limitiert, was die Interaktion mit nicht echtzeitkonformen Teilen des Systems – wie in diesem Fall dem Linux-Kernel – anbelangt. Im Gegensatz zu homogenen Echtzeitumgebungen werden in einer Echtzeiterweiterung oftmals lediglich Ausgaben für Debuggingzwecke standardmäßig unterstützt, keinesfalls aber Datei- oder Netzwerkzugriffe[2]. Da Programme, die auf diesen Erweiterungen laufen sollen, ebenfalls die Möglichkeit einer Visualisierung oder Protokollierung ihres Ablaufes benötigen, wird allerdings mindestens eine einfache Schnittstelle zur Verfügung gestellt, die eine sichere – also blockadefreie – Kommunikation ermöglicht. Dabei handelt es sich in der Regel entweder um so genannte „Pipes“ für byteweise Kommunikation oder um so genannte „Mailboxen“ zum Austausch von Nachrichten.

In der Praxis wird von Entwicklern eine individuelle und einfache Möglichkeit realisiert, um den geforderten Datenaustausch auf der Basis der jeweils existierenden Schnittstelle zu gewährleisten. Da die Portierung komplexer C++ Systeme nicht möglich wäre, ohne eine I/O-Möglichkeit zu schaffen, die derjenigen einer normalen Betriebssystemumgebung zumindest ebenbürtig ist, musste hier eine Lösung konzipiert werden, die alle üblicherweise verfügbaren Funktionen zur Verfügung stellt, aber ebenfalls auf die existierende Kommunikationsmöglichkeit[3] aufsetzt.

Die Lösung für solche Probleme ist die Einrichtung eines so genannten Stellvertreters (Proxy), wie er auch für ähnliche Situationen vielfach eingesetzt wird (z.B. [35, 29]). Ein Proxy ist ein Entwurfsmuster (Design Pattern), gehört zu der Gruppe der Strukturmuster (Structural Patterns) und verlagert die lokale Kontrolle auf ein anderes Objekt (im Folgenden als I/O-Handler bezeichnet) [27]. In diesem Fall muss dieser I/O-Handler gegenüber der Real-Time-Umgebung das Verhalten der benötigten I/O-Funktionen nachbilden, als handele es sich um einen normalen Zugriff auf die C-Library, und auf der Linux-Seite den Datenaustausch mit der Echtzeiterweiterung und dem Linux-System sicherstellen (siehe Abbildung 4.9). Dies wird intern durch eine Zustandsmaschine ermöglicht, die in einem eigenen Thread läuft, welcher in seiner Priorität kurz über dem

[2] An dieser Stelle sollen die existierenden Implementierungen von hart echtzeitfähigen Netzwerken ausgeklammert werden!

[3] Bei RTAI/Linux: Mailboxen

zugehörigen Echtzeitprogramm (hier dem Simulator) angesiedelt ist, um Prioritätsinversionen zu verhindern (siehe Abbildung 4.10).

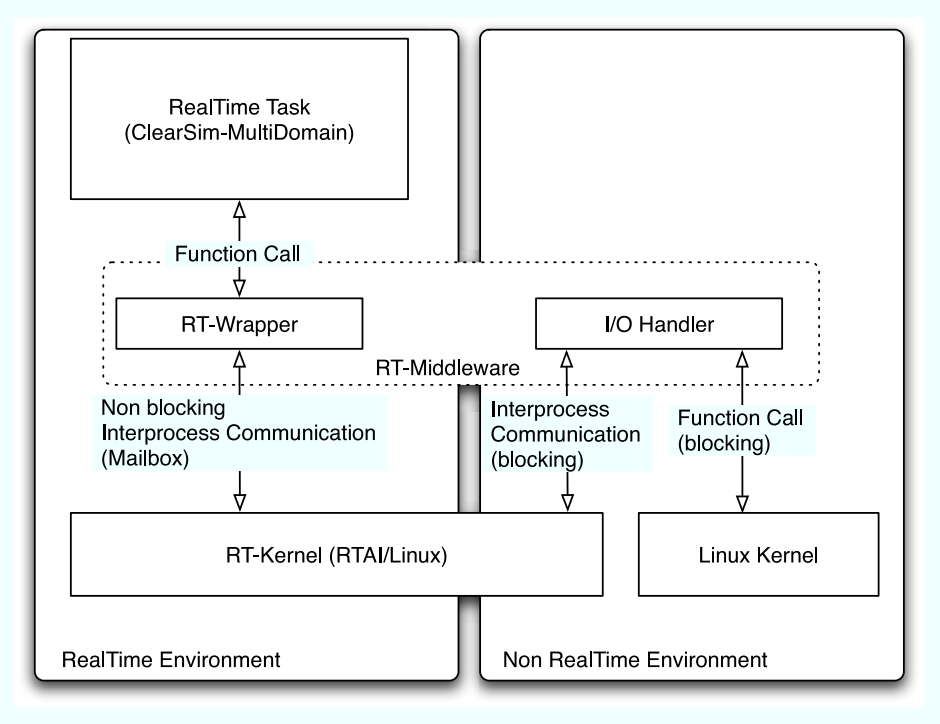


Abbildung 4.9: Prinzipieller Aufbau der RT-Middleware

Die Kommunikation zwischen der harten Echtzeitumgebung und dem Thread des I/O-Handlers erfolgt nun folgendermaßen, wie an einem einfachen Beispiel dargestellt werden soll (siehe hierzu auch Abbildung 4.11 und Abbildung 4.12) :

Sollen Daten aus der Echtzeitumgebung an Linux übergeben werden (z.B. um sie in eine Datei zu schreiben), so muss die Datei zuerst geöffnet werden. Hierzu wird die Funktion

```
int RTLib_IO_OpenFile (RT_HANDLE *handle, const char *filena-
me, const unsigned int direction, int my_prio);
```

aufgerufen. Diese Funktion erzeugt nun eine Datenstruktur, in der die hierfür relevanten Daten, inklusive einer neu erstellten Mailbox, eingetragen werden und fügt sie in eine verkettete Liste ein, die gemeinsam mit dem Thread des I/O-Handlers verwaltet wird. Diese Datenstruktur wird in Zukunft für alle Zugriffe auf die Datei verwendet und wird als Zeiger über den Parameter „handle“ zurückgegeben. Abschließend wird mittels einer Signalisierungs-Semaphore der I/O-Handler über eine Änderung informiert. Der I/O-Handler wird nun automatisch die neue Datenstruktur auslesen und das Öffnen der Datei in der Linuxumgebung durchführen. Somit ist asynchron die gewünschte Datei geöffnet worden. Wichtig zu erwähnen ist die Tatsache, dass eine Blockade des I/O-Handlers keine Auswirkungen auf das Echtzeitverhalten des Programms hat, wie ja gefordert wird!

Das Schreiben der Daten erfolgt nun mit der Funktion

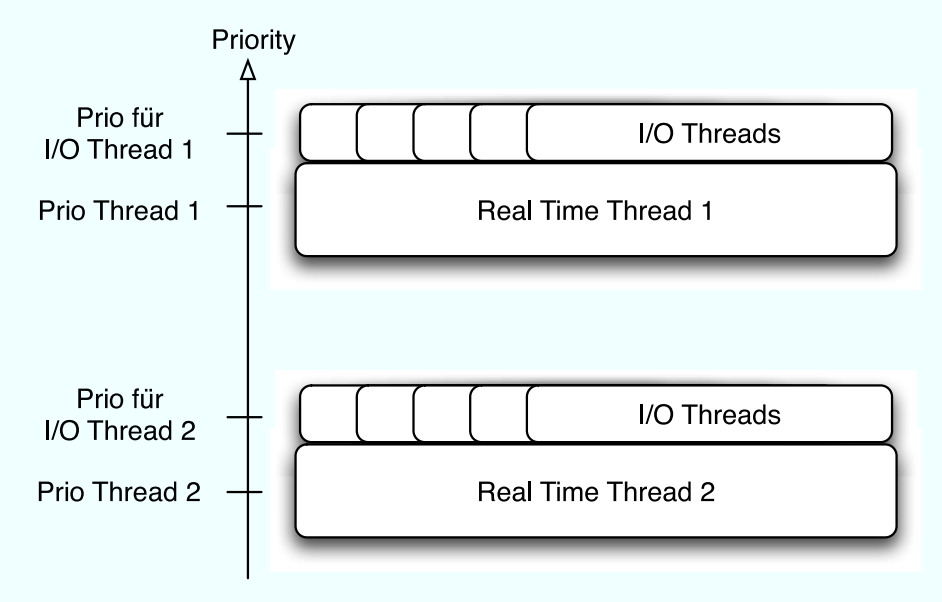


Abbildung 4.10: Prioritäten bei der Real-Time Middleware

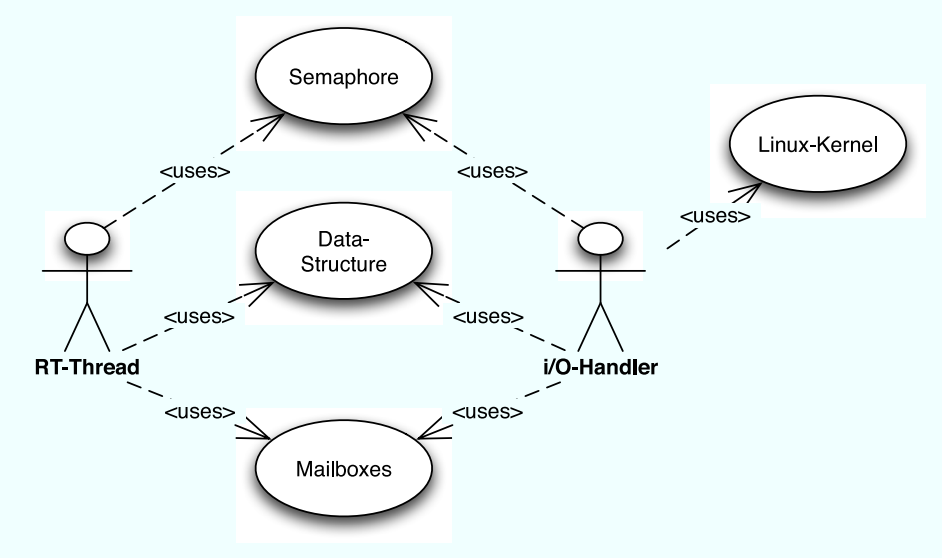


Abbildung 4.11: Gemeinsame Elemente zwischen RT-Thread und I/O-Handler

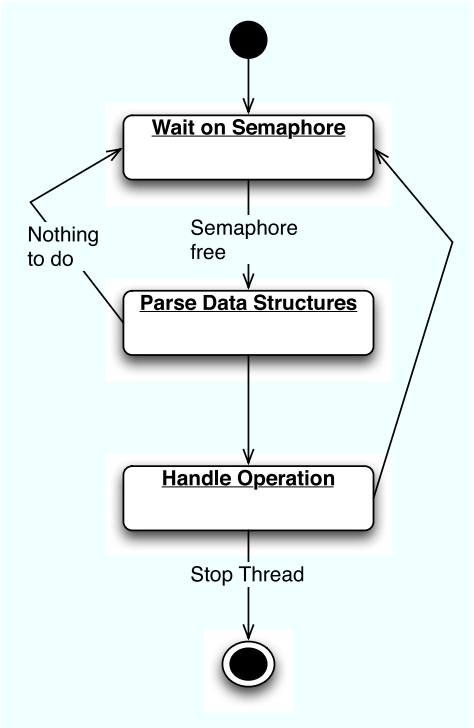


Abbildung 4.12: Darstellung des prinzipiellen Ablaufs des I/O Handlers

```
int RTLib_IO_FPrintf (RT_HANDLE handle, const char *format,...);
```

welches die Datenstruktur verwendet, die über den Zeiger „handle“ adressierbar ist. Sollte der I/O-Handler noch nicht die Möglichkeit gefunden haben, die Datei zu öffnen, so ist dies kein Problem, das zu einer Gefährdung der Zeitschranken führt. Da von dieser Funktion blockadefrei in die bereits existierende Mailbox geschrieben wird, kann dessen Überfüllung schlimmstenfalls zu Datenverlust führen.

In diesem Beispiel wird die Problematik bei der Umstellung von synchronen zu asynchronen Funktionen demonstriert, was in diesen Fällen zu einem unterschiedlichen funktionalen Verhalten des Simulators führen würde, sollte es im Programmcode nicht berücksichtigt werden:

Sollte z.B. eine Datei, auf die zugegriffen werden soll, nicht beschreibbar sein, so würde dies im synchronen Fall sofort zu einem Fehler in der Open-Funktion führen, auf die der Programmcode des Simulators sofort reagieren würde. Im asynchronen Fall wird ein solches Problem erst sehr viel später sichtbar werden, was im Programm entsprechend berücksichtigt werden muss.

Dieser funktionale Unterschied ist aufgrund der notwendigen Entkopplung nicht zu vermeiden und ist bei der Portierung entsprechend zu berücksichtigen. Da aber die Simulationsmodule nur an wenigen Stellen Funktionen dieser Art ausführen und diese leicht zu identifizieren sind, ist der hieraus resultierende Aufwand bei der Umstellung eher gering.

Bei dem Auslesen von Daten aus einer Netzwerkverbindung oder Datei ist zwischen notwendigen und optionalen Leseoperationen zu unterscheiden: Beim optionalen Lesen kann ein Programm

weiter arbeiten, auch wenn keine Daten zur Verfügung stehen. Dies wird durch eine blockadefreie Funktion ermöglicht, bei der lediglich abgefragt wird, ob Daten zur Verfügung stehen. Im zweiten Fall benötigt das Programm unbedingt gültige Daten, damit der Programmablauf überhaupt durchgeführt werden kann. In diesem kritischen Fall macht es keinen Sinn, blockadefreie Leseoperationen zu verwenden. Stattdessen stellt die Middleware Operationen mit einer maximalen Blockadezeit (einem Timeout) zur Verfügung. Wird diese Zeit erreicht und der Vorgang abgebrochen, so bleibt in der Regel nur noch die geordnete Beendigung der Simulation, da ein Weiterlaufen keinen Sinn mehr macht. Abgemildert wird dieses Problem, indem die meisten notwendigen Leseoperationen in der Initialisierungsphase des Simulators durchgeführt werden, um z.B. Konfigurationsdateien einzulesen. Da nur in der Simulationsphase die Blockadefreiheit garantiert werden muss, kann in diesem Fall die vorliegende Implementierung unverändert übernommen werden.

4.2.3.2 Memory Management in einer RT-Umgebung

Eine weitere Notwendigkeit zur erfolgreichen Ausführung einer komplexen Software – wie ClearSim-MultiDomain – in einer Echtzeitumgebung ist die Notwendigkeit zur dynamischen Anforderung von Speicher. Konsequente Echtzeit-Evangelisten verlangen, dass ein Programm unter harten Echtzeitbedingungen zur Laufzeit keine Speicherforderungen stellen darf, da sie möglicherweise nicht erfüllbar sind und somit zum Fehlverhalten oder Abbruch führen würden. Für Systeme, die lebenserhaltene Funktionen zur Verfügung stellen, wäre ein Ausfall sicherlich eine Katastrophe. Nur stellen solche Systeme eher eine Ausnahme dar, für die natürlich besondere Regeln gelten müssen.

In der Praxis sind alle Echtzeitbetriebssysteme in der Lage, auf irgendeine Weise dynamischen Speicher zur Verfügung zu stellen, da ansonsten gerade komplexere Echtzeitprogramme nicht realisierbar wären. Da der Speicher natürlich limitiert ist, muss sichergestellt werden, dass auf eine unerfüllte Speicheranforderung im geeigneten Maße reagiert werden kann.

Soll in einer harten Echtzeitumgebung dynamisches Speichermanagement zur Verfügung gestellt werden, so muss eine Anforderung blockadefrei und innerhalb einer deterministischen Zeitschranke beantwortet werden, was von nicht echtzeitfähigen Betriebssystemen natürlich nicht zu erwarten ist [51]. Zur Realisierung von – in diesem Zusammenhang – korrektem Memory-Management, welches auch bei gleichzeitiger Anforderung mehrere Prozesse die geforderten Bedingungen erfüllt, existieren verschiedene so genannte optimistische und konservative Verfahren [24]. Da die Realisierung optimistischer Algorithmen sehr aufwendig sind, werden in den meisten Fällen die folgenden konservativen Varianten implementiert:

1. Statischer Memory-Handler: Bei dieser Variante wird vor dem Echtzeitlauf eine ausreichende Menge Speicher vom System reserviert und von der Auslagerung auf einen Massenspeicher ausgenommen. Anschließend wird dieser Speicher in Blöcke unterteilt und bei Bedarf an die Echtzeitprozesse übergeben.

2. Dynamischer Memory-Handler: Es wird zu Beginn ein Memory-Pool einer gewissen Größe reserviert. Wird durch die Speicheranforderungen eine gewisse untere Marke des freien Speichers erreicht, so wird zu einem sicheren Zeitpunkt neuer Speicherplatz vom General-Purpose-Betriebssystem geholt.

Beide Varianten haben natürlich ihre Vor- und Nachteile: Während die erste Variante tendenziell immer zu viel Speicher anfordert, da vor Ablauf eines Programms oftmals die benötigte Speichermenge nicht vorhersagbar ist und somit immer möglichst das theoretische Maximum reserviert wird, stellt sich bei der zweiten Variante die Frage, wann denn ein sicherer Zeitpunkt für die Anforderung neuen Speichers wäre. Dies verlangt die präemptive Integration des Algorithmus in den Idle-Prozess des Echtzeitbetriebssystems, da nur in diesem Fall dieser Vorgang gefahrlos durchgeführt werden kann. Dies kann allerdings theoretisch zu Engpässen kommen, wenn das System sehr dynamischen Speicheranforderungen unterworfen ist und somit nicht genügend Zeit erhält, den Speicherpool rechtzeitig aufzufüllen.

Da zu Beginn der Arbeit keine befriedigende dynamische Speicherverwaltung in RTAI/Linux integriert war und erst im laufe der Arbeit ein dynamischer Handler hinzugefügt wurde, entstand ein einfacher, statischer Memory-Handler und wurde in die Middleware integriert. Somit kann auch auf Echzeiterweiterungen, die keinen ausreichenden Mechanismus für die Speicherverwaltung zur Verfügung stellen, eine korrekte Funktion der Simulation gewährleistet werden. Ein weiterer Vorteil der hier gewählten Lösung ist, dass sie komplett im User-Space implementiert wurde und somit unabhängig vom eingesetzten Betriebssystem ist. Weiterhin konnte der Memory-Handler auf das typische anforderungsverhalten des Simulators optimiert werden und kann somit sehr schnell den geforderten Speicher liefern (typisches Zeitverhalten: O(1), im worst-case Fall: O(n)) .

4.2.3.3 Transparente RT-Abstraktionsschicht

Für Linux ohne Echtzeiterweiterung wurde eine transparente Version der RT-Middleware implementiert. Da alle Echtzeitfunktionen über diese Abstraktionsschicht auf die Standard-C-Library abgebildet werden, ist es nun möglich, eine beliebige Echtzeitsimulation ohne Neukompilierung auf jedem Linux-System auszuführen. Dies geschieht natürlich unter Verlust der Echtzeiteigenschaften, indem die Simulation so schnell abläuft wie es die Rechenleistung des verwendeten Systems ermöglicht. Da die Middleware als Plugin (Shared-Library) in den Simulator integriert wird, kann ohne Neukompilierung die Abstraktionsschicht ausgetauscht werden (siehe Abbildung 4.13 im Gegensatz zu Abbildung 4.7 auf Seite 55).

Neben der Ausführung von Echtzeitsimulationen auf gewöhnlichen Linux-Systemen ermöglicht diese spezielle Version der Middleware das zeitweise Ausschalten der Nutzung des Echtzeit-Subsystems, was für Debugging-Zwecke sehr nützlich sein kann. Da es bei einer fehlerhaften Programmierung in einer Echtzeitumgebung schnell zu Blockierungen (Deadlocks) kommt, die, aufgrund der hieraus resultierenden totalen Verklemmung des Systems, nur schwer aufgespürt werden können, eignet sich diese Version besonders für die Kontrolle der funktionalen Korrektheit der jeweiligen Implementation. Da der Simulator nun komplett in der unveränderten Umgebung läuft, muss nicht auf komfortable Testmöglichkeiten, mittels üblicher Debugger-Tools, verzichtet werden. Erst nachdem die funktionale Korrektheit der Implementation überprüft wurde, wird die Echtzeitsimulation auf dem wirklichen Echtzeitsystem gestartet.

Mit dieser Lösung konnte der Portierungsvorgang – im Geiste dieser Arbeit – ebenfalls inkrementell durchgeführt werden und so deutlich beschleunigt werden.

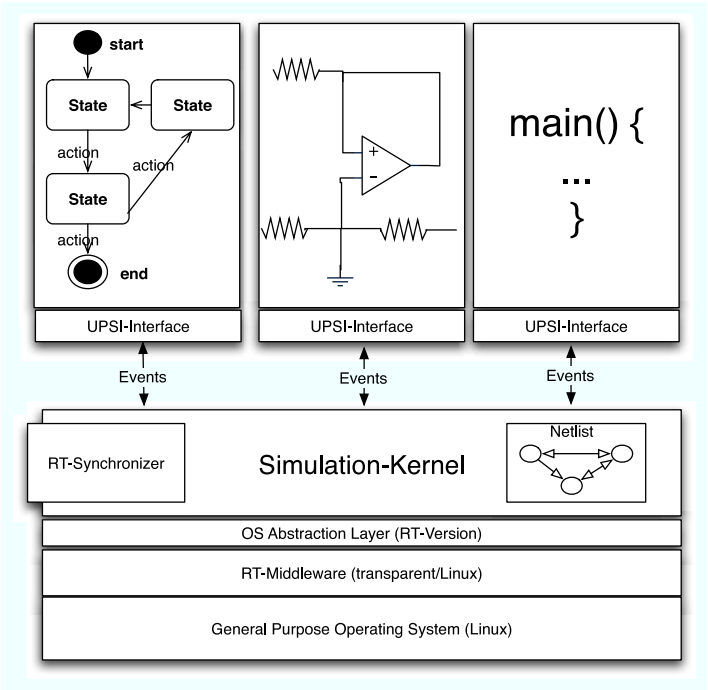


Abbildung 4.13: Simulationsumgebung mit transparenter RT-Middleware

4.2.4 Zeitgesteuerte Systemsimulation

Wie schon in Abschnitt 2.4.2.1 beschrieben wurde, haben ereignisgesteuerte Simulationsalgorithmen die Eigenschaft, dass sie auf der virtuellen Zeitachse Sprünge sowohl in die Zukunft als auch in die Vergangenheit durchführen können. Dies verletzt die in Abschnitt 3.2 formulierte Forderung nach Zeitsynchronität mit der Realität.

Aus diesem Grunde wurde – zusätzlich zum existierenden eventgesteuerten Algorithmus – ein zeitgesteuerter RT-Synchronizer entwickelt, der das Fortschreiten der Simulation in äquidistanten Zeitschritten ermöglicht. Dies geschieht unter Nutzung von zyklisch eingeplanten Echtzeit-Threads, die von der Echtzeitumgebung zur Verfügung gestellt werden. Dies ermöglicht die exakte Einhaltung der vorgegebenen Zeitintervalle (siehe auch Abschnitt 2.4.2.2).

Da es sich bei ClearSim-MultiDomain um eine zeitdiskrete Simulation handelt, wird die geforderte Zeitsynchronität mit der Realität an vorgegebenen diskreten Zeitpunkten am Ende eines jeweiligen Intervalls sichergestellt. Da alle in der Simulation befindlichen Simulationsmodule zu diesem Zeitpunkt zueinander zeitlich synchron sind, ist es problemlos möglich, ohne zusätzliche Synchronisationsmechanismen zwischen ihnen Daten auszutauschen, da alle zu diesem Zeitpunkt erzeugten Events in ihrer Eventzeit auf jedenfall identisch mit allen individuellen Modellzeiten sind. Vor oder nach diesem Zeitpunkt führen die Simulationsmodule individuell und somit asynchron die Simulation des gegebenen Zeitintervalls durch.

Da die Simulationsmodule für einen eventbasierten Algorithmus programmiert wurden, können sie zu jedem Zeitpunkt – und nicht nur zum Zeitpunkt der Zeitsynchronität – Ereignisse erzeugen und Informationen anfordern, was unweigerlich zu Problemen führt, die korrekt behandelt werden

müssen:

- Ereignisse die erzeugt werden, werden in ihrer Ereigniszeit auf den nächsten Synchronisationspunkt verschoben an dem sie ausgeliefert werden (siehe Abbildung 4.14).

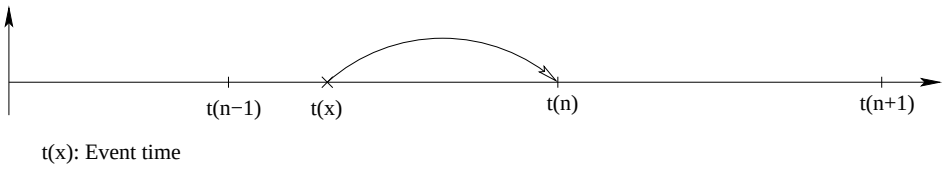


Abbildung 4.14: Verzögerung der Ereignisse zum nächsten zeitsynchronen Punkt

- Vor dem Beginn eines neuen Simulationsintervalls werden alle Ausgangsports, die möglicherweise abgefragt werden könnten, gesichert. Fordert ein Simulationsmodul von einem dieser Ports Informationen an, so wird diese Anfrage mit einer der vorgespeicherten Informationen beantwortet, wobei die Ereigniszeit auf den Zeitpunkt der Anfrage verschoben wird (siehe Abbildung 4.15).

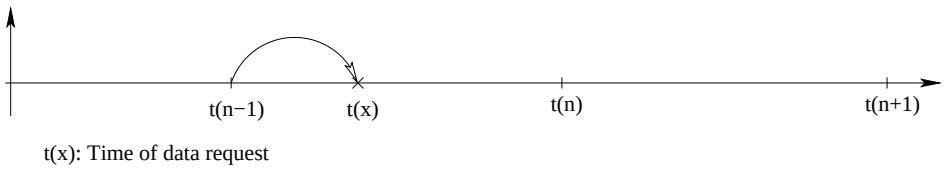


Abbildung 4.15: Verzögerung der vorgespeicherten Ereignisse zum Zeitpunkt der Anforderung

Mit diesem Vorgehen werden die beschriebenen Probleme gelöst, wenn auch mit unvermeidlichen Verzögerungen der auftretenden Ereignisse von einem Zyklus und bei rückgekoppelten Modulen von 2 Zyklen.

Die einzige Möglichkeit, um die hier beschriebenen Verzögerungen zu minimieren ist die Verkürzung der Zykluszeit, was aber unweigerlich eine Verlängerung der Simulationszeiten bedeutet und somit bei der Echtzeitsimulation die Einhaltung der maximalen Zeitschranken, und somit einen Echtzeitfaktor $R < 1$ gefährdet, wie in Abbildung 4.16 gezeigt wird.

Der Entwickler muss somit das Optimum zwischen Einhaltung des Echtzeitverhaltens und der Minimierung der Verzögerung in der Simulation erreichen, was nur iterativ am komplett virtuellen Prototypen erfolgen kann. Um dies zu erleichtern, wurde eine umfangreiche Protokollfunktion implementiert, die für jeden Simulationszyklus die Information über die realen Laufzeiten der jeweiligen Simulationsmodule und der Kommunikation in eine Protokolldatei speichert. Somit ist es leicht möglich, sich iterativ dem Optimum zu nähern, worauf später detaillierter eingegangen werden soll.

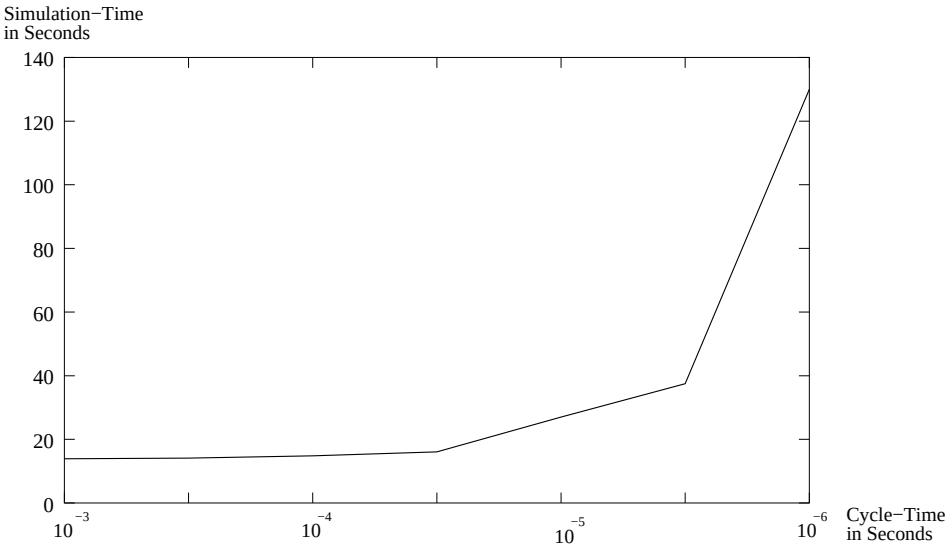


Abbildung 4.16: Einfluss der Zykluszeit auf die Simulationszeit

4.2.5 Schnittstelle zwischen virtuellem und realem System bei zeitgenauer Systemsimulation

Um die Simulation gemischter Prototypen, bestehend aus virtuellen und realen Teilsystemen, zu ermöglichen, muss eine Kopplung realer Komponenten an die Simulation ermöglicht werden. Hierzu wird der virtuelle Prototyp geschnitten und einige der virtuellen Komponenten durch reale Pendants ersetzt. Der Schnitt des Systems sollte aufgrund der unvermeidlich auftretenden Kommunikationslatenzen nur an losegekoppelten Stellen durchgeführt werden, da dort pro Zeiteinheit am wenigsten Nachrichten ausgetauscht werden. Somit werden bevorzugt die Kanäle zwischen den Aktoren (den Simulationsmodulen) für eine Trennung ausgewählt werden. Da ein Kanal bei ClearSim-MultiDomain lediglich abstrakte Ereignisse (Events) überträgt, die in der Realität in dieser Form nicht existieren, hat eine geeignete Schnittstelle die Aufgabe, diese Ereignisse in die korrespondierende „Form“ der Realität zu wandeln.

Da ein Ereignis auf niedriger Abstraktionsebene z.B. eine Spannungsänderung oder auf hoher Abstraktionsebene eine komplette Nachricht bedeuten kann, aber auf der realen Seite in der Regel nur Spannungen und Ströme auftreten, liegt die Herausforderung bei der korrekten Übersetzung zwischen den beiden Abstraktionsebenen. Grundsätzlich stellt ein Ereignis immer eine Veränderung eines bestehenden Zustands dar und beinhalten eine Eventzeit, die den Zeitpunkt der Erzeugung definiert. Wird ein Ereignis aufgrund einer Veränderung – z.B. einer Spannung – in der realen Umgebung erzeugt, so wird diese Ereigniszeit auf die aktuelle Simulationszeit gesetzt. Ereignisse, die das virtuelle Umfeld verlassen werden gesendet, wenn ihre Ereigniszeit gleich derjenigen der Schnittstelle ist, als handelte es sich bei einer Schnittstelle um ein Simulationsmodul.

Modelle, die angesichts der begrenzten Rechenzeit heutiger Computer in Echtzeit simulierbar sind, stellen meist Varianten höherer Abstraktion dar, da eine niedrige Abstraktionsebene zu viel Rechenzeit verlangen würde. Bei einem Mikrocontroller bedeutet es z.B., dass nicht die VHDL-Beschreibung des originalen Controllers simuliert wird, sondern sein – in einer Hochsprache formuliertes – funktionales Verhalten. Um eine zeitgenaue Simulation zu ermöglichen, wird zusätzlich ein

Modell integriert, welches das zeitliche Verhalten beschreibt und somit, zusammen mit dem funktionalen Verhaltensmodell einer vollständigen und ausreichend genauen Modellierung des Gesamtverhaltens führt, bei gleichzeitig sehr hoher Simulationsgeschwindigkeit [11, 64]. Ähnliches gilt für die Modelle von Bussen, wie dem in ClearSim-MultiDomain implementierten CAN-Bus, der ebenfalls aus einer funktionalen Verhaltensbeschreibung und einem Zeitmodell besteht. Bus-Modelle stellen somit in dem hier beschriebenen Zusammenhang einen Sonderfall dar, da sie aufgrund ihrer aktiven Arbitrierungseigenschaften als Simulationsmodul implementiert werden, und nicht – wie vielleicht intuitiv vermutet – einen Aktor-Kanal zwischen anderen Simulationsmodulen darstellen. Soll ein System nun z.B. an einem CAN-Bus geschnitten werden, so wird ein Simulationsmodul und somit ein funktionales und zeitliches Modell auf hoher Abstraktionsebene geschnitten.

Die zwei Möglichkeiten – der Schnitt eines Kanals und der Schnitt eines Bus-Modells – sollen nun im Folgenden beschrieben werden:

- Wird das System an einem Kanal geschnitten, so wird das in der virtuellen Umgebung liegende Ende mit einer geeigneten Schnittstelle verbunden (im Folgenden als V/R-Interface bezeichnet), welche die Wandlung zwischen Ereignissen und – in der Regel – Spannungen durchzuführen hat. Soll eine Übertragung von Ereignissen höherer Abstraktion realisiert werden (z.B. CAN-Nachrichten), so ist eine softwarebasierte Wandlung der Nachrichten in Spannungsmuster in der Praxis nicht möglich, da die genaue Einhaltung der jeweiligen Signalpegel und Zeiten in Echtzeit auf dem verwendeten Industrie-PC nicht realisierbar wäre. Hier ist eine hardwarebasierte Lösung zu verwenden. In jedem Fall ist aber eine gewisse zusätzliche Latenzzeit für die Übertragung der Information über die Schnittstelle und für die Umwandlung in ClearSim-Events zu verzeichnen. Diese Latenzzeiten müssen ausreichend klein sein, um das Simulationsergebnis nicht zu verfälschen.

- Bei dem Schnitt eines Bus-Modells wird das Simulationsmodul direkt mit dem V/R-Interface verbunden. Auch in diesem Fall ist zur Garantierung der Echtzeitfähigkeit der Simulation eine hardwarebasierte Wandlung der abstrakten Nachrichten in Spannungsimpulse notwendig. Der gesamte Bus besteht somit aus einem virtuellen und einem realen Teil, was unvermeidbare Folgen für das funktionale und das zeitliche Modell hat, wie am Beispiel des CAN-Busses erläutert werden soll:
  Das zeitliche, sowie das funktionale Modell benötigen zur genauen Berechnung der Übertragungszeiten und für die Arbitrierung eine vollständige Sicht auf den gesamten Bus. Diese wird nun durch die Schnittstelle zum realen Teil des Busses eingeschränkt:

  - Eine reale CAN-Nachricht muss erst komplett empfangen worden sein, bevor sie zu einem Ereignis (Event) gewandelt werden kann und somit im virtuellen Teil des Busses sichtbar wird.
  - Der Übergang von einer niedrigen zu einer hohen Abstraktionsebene ist immer mit einem Detailverlust der übertragenden Information verbunden. Im umgekehrten Fall müssen Informationen hinzugefügt werden, die eventuell nicht zur Verfügung stehen. In unserem Beispiel ist ein Verlust der detaillierten Zeitinformationen über den realen Teil des Busses zu verzeichnen, welche aber in dem Zeitmodell des virtuellen Busses für die Berechnung des Bus-Timings weiterhin benötigt werden.

  Beide Fälle führen zu einem gewissen Grad an Ungenauigkeit. Im ersten Fall erscheinen Ereignisse, die in der virtuellen Umgebung genau gleichzeitig auftreten würden, als nacheinander

auftretend. Dies kann bei der Arbitrierung des Busses die Folge haben, dass eine im rein virtuellen Fall verdrängte Nachricht im gemischt virtuellen-/realen System doch gesendet wird und umgekehrt.

Im zweiten Fall benötigt der Algorithmus des Zeitmodells Informationen, die vom V/R-Interface über ein Teilsystem zur Verfügung gestellt werden müssen, in das kein oder nur ein ungenügender Einblick besteht. Dies kann über ein eigenes Zeitmodell im Interface erfolgen, das mittels Beobachtung des Busgeschehens, wie es in der Regelungstechnik als Zustandsraumregler ähnlich angewendet wird [44], möglichst genaue Informationen liefert.

Die hier beschriebenen Ungenauigkeiten erscheinen auf den ersten Blick als Schwäche der gemischt virtuellen-/realen Simulation, da ein Fehlverhalten durch diese Verfälschungen nicht ausgeschlossen werden kann. Beim genaueren Hinsehen wird allerdings recht schnell festgestellt, dass diese Eigenschaft eine Art Vorbote der im realen System vorzufindenden Ungenauigkeiten und Indeterminismen darstellt und somit hilft, die Korrektheit der Implementation zu prüfen. Gerade die in der Simulation eines virtuellen Prototypen gewollte Eigenschaft der zeitlichen Invarianz, also dem immer gleichen Verhalten bei jedem Simulationslauf, kann zur Folge haben, dass Fehler im Design des Systems nicht erkannt werden, weil sie zufällig nicht Teil der verwendeten Testfälle sind. Diese Probleme äußern sich dann später im realen System z.B. als so genannte Timingprobleme, deren Ursache meist nur schwer zu finden ist.

Der aufgrund seiner zeitlichen Invarianz systematisch überprüfbare virtuelle Prototyp erhält somit bei dem Übergang zum realen System ebenfalls schrittweise diejenigen Eigenschaften, die ein reales System ausmachen: Wird z.B. eine CAN-Nachricht mit höchster Priorität an einen Infineon C-167 geschickt, so schwankt die reale Übertragungszeit und es kann lediglich mit Worst-Case Zeiten gearbeitet werden. Die Arbitrierung auf dem BUS kann bei 1 MHz zwischen 0 und 157 μs dauern (näheres hierzu in Abschnitt 5.2), da die aktuell auf dem BUS befindliche Nachricht abgewartet werden muss. Anschließend wird die Nachricht übertragen und vom empfangenden CAN-Bus Controller gespeichert, was eine Zeit zwischen 47 μs und 157 μs benötigt. Die Auslösung eines Interrupts auf dem Mikrocontroller, die zum Auslesen des CAN-Controllers führt, dauert zwischen 250 und 500 ns, gefolgt von ca. 800 ns zur Übertragung der Nachricht in den Prozessor-Core, falls der verwendendete XBUS nicht anderweitig belegt ist. Somit ist die gesendete Nachricht nach maximal 315,3 μs am Ziel angekommen, kann aber auch nur 48,05 μs benötigen [5].

Trotz allem wird aber – zumindest im virtuellen Systemteil – die optimale Beobachtbarkeit erhalten und ermöglicht auftretende Probleme zu analysieren und zu lösen. Das Ziel muss ein reales System sein, welches zeitliche Ungenauigkeiten im üblichen Rahmen problemlos toleriert, ansonsten wäre es in der Praxis kaum sicher funktionsfähig.

Was somit bleibt, ist die Erkenntnis, dass die unvermeidlichen Ungenauigkeiten unterhalb einer gewissen akzeptablen Schwelle bleiben müssen, damit die daraus erfolgenden Einflüsse auf das System nicht zu stark das funktionale und zeitliche Verhalten stören und somit die Ergebnisse der Simulation ihre Aussagefähigkeit verlieren. Dieses Problem soll im nächsten Kapitel detaillierter beleuchtet werden, indem anhand der realen Implementierung genau diese Verzögerungen gemessen und beurteilt werden.

5 Validierung des Konzeptes

Bevor die in dieser Arbeit geschaffene Simulationsumgebung praktisch eingesetzt werden konnte, musste sie einer Validierung unterzogen werden. Ziel dieser Validierung war einerseits die Versicherung, dass der Simulator wirklich frei von Programmzeilen war, welche die Echtzeiteigenschaften gefährden. Andererseits mussten die Laufzeiteigenschaften analysiert werden, um die Einsetzbarkeit für verschiedene in Frage kommende Nutzungsszenarien abschätzen zu können. Zu diesen Eigenschaften gehören einerseits die Einflüsse des verwendeten Host-Systems und somit Linux mit der Echtzeiterweiterung RTAI auf einem Industrie-PC. Andererseits mussten die Einflüsse des Interfaces zwischen realem- und virtuellem Systemteil auf die Simulation analysiert werden.

Im folgenden Abschnitt werden die Ergebnisse der Validierung der Echtzeitumgebung dargelegt, gefolgt von der Validierung des Interfaces.

5.1 Validierung der Umgebung für die Echtzeitsimulation

Um Modelleinflüsse bei den Analysen weitestgehend auszuschließen oder zumindest einschätzbar zu machen, wurden diese Versuche nicht an realitätsnahen Modellen durchgeführt, sondern an synthetischen Zustandsmaschinen, die möglichst gleichmäßige Laufzeiteigenschaften aufweisen (siehe Abbildung **5.1)**. Die hier verwendeten Zustandsmaschinen führen prozessorlastige Berechnungen aus, wenn sie im Besitz eines „Tokens“ sind. Dieses „Token“ wird nach der Berechnung an die andere Maschine übergeben und springt somit immer hin und zurück. Bei der durchgeführten Berechnung handelt es sich um eine einfache Fließkomma-Division, die über eine Schleife 20.000 mal ausgeführt wird. Aufgabe dieser Berechnung ist dabei, sicher zu stellen, dass die auftretenden Simulationszeiten gegenüber denen der Kommunikationszeiten dominant sind, wie es auch bei realen Simulationsmodulen der Fall wäre. Durchgeführt wurden die Versuche auf einem Industrie-PC von Inova mit einem Pentium 4 Prozessor, der mit 2 GHz getaktet war.

Sollte das Simulationsverhalten von dem erwarteten Zeitverhalten abweichen, so resultieren diese Effekte aufgrund der hier getroffenen Modellwahl mit Sicherheit aus Einflüssen der Simulationsumgebung selbst. Als Ursachen dieser Einflüsse können folgende Möglichkeiten identifiziert werden:

1. Einflüsse aufgrund von Implementationsfehlern, wie z.B. direkte Zugriffe auf Ressourcen, die nicht vom Echtzeitbetriebssystem verwaltet werden und somit zu nichtdeterministischen Verzögerungen führen.
2. Einflüsse des Echtzeitbetriebssystems und der verwendeten Hardware, wie z.B. Bus- bzw. Cache-Latenzen.

Um diese Einflüsse sichtbar zu machen, wurde das Linux-System mit verschiedene Lasten beaufschlagt, während die Simulation auf der Echtzeiterweiterung lief. Eine geeignete Last hat für

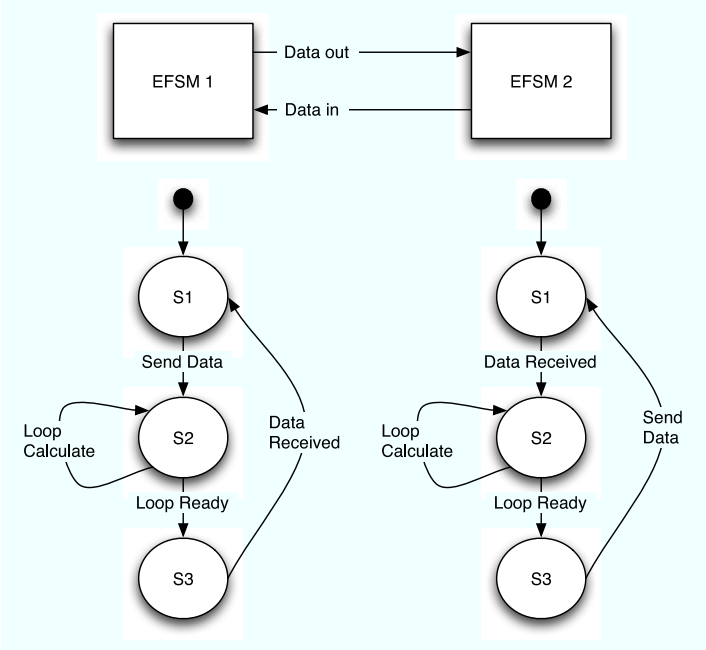


Abbildung 5.1: Verwendete Zustandsmaschine für die synthetischen Tests

eine hohe Anzahl von Prozessor- und I/O-Zugriffen zu sorgen und möglichst grosse Teile des verfügbaren Speichers zu binden. Da der Gnu C-Compiler üblicherweise als Benchmark eingesetzt wird, um die Gesamtleistung eines Systems unter besonderer Berücksichtigung der Speicher- und Festplatten-Performance zu berücksichtigen, wurde dieser auch in diesem Fall eingesetzt. Als geeignete Last zeigte sich die Kompilation des Linux-Kernels, wobei durch den Aufruf des Kommandos „make“ mit dem Parameter „-j“ für jeden unabhängigen Source-Zweig ein eigener GCC-Compiler gestartet wurde. Somit konkurrierten viele hundert Compiler um den Speicher und führten massive Festplattenzugriffe durch. Dies führte dazu, dass die Faktoren Cache-Verdrängung, Auslagerung von laufenden Prozessen in die Auslagerungsdatei und Bus-Latenzen maximiert wurden.

Durch eine im Simulator integrierte Profiling-Funktion wurden die realen Laufzeiten der Simulationsmodule sowie der Kommunikation untereinander für jeden Simulationszyklus protokolliert und mittels einer Visualisierungssoftware als Diagramme sichtbar gemacht. Entsprechend des zu erwartenen Modellverhaltens müssten die Diagramme vollkommen gleichmäßige Laufzeiten zeigen, da jeder Simulationszyklus immer die gleiche Rechenzeit benötigt. Die oben beschriebenen Einflüsse würden sich durch Ungleichmäßigkeiten in Form von Spikes in diesen Diagrammen darstellen, da zeitweise ein ungewöhnlich großer Zeitbedarf für die Simulation eines Zeitschrittes festzustellen wäre.

Da die beiden oben aufgezählten Einflussfaktoren jeweils als Spikes erscheinen, musste zusätzlich die Frage geklärt werden, ob es möglich ist, diese anhand der Diagramm-Bilder voneinander zu unterscheiden. Dazu wurden zusätzliche Diagramme von Testsimulationen erzeugt, die mit der nicht echtzeitfähigen Variante der Simulation durchgeführt wurden[1] und anschließend mit denen

[1] Die Simulatoren unterscheiden sich nur in der Wahl der Middleware: D.h., die Nicht-Echtzeitsimulationen wurden mit der transparenten Version durchgeführt (siehe 4.2.3.3).

der Echtzeitfähigen verglichen, wobei ebenfalls die Messungen einmal bei Leerlauf von Linux und einmal bei hoher Last entstanden.

Die Abbildung 5.2 a) zeigt die Simulation auf einer nicht echtzeitfähigen Version des Simulators, während Linux unbelastet läuft. Abbildung b) zeigt die gleiche Simulation unter Vollast. Im Vergleich hierzu ist auf Abbildung 5.3 a) die gleiche Simulation auf der echtzeitfähigen Variante von ClearSim-MultiDomain im Leerlauf und bei Abbildung b) unter Vollast dargestellt. Bei diesen Diagrammen wurde auf der Abszisse die Simulationszeit aufgetragen (hier 5 Sekunden), wobei die Ordinate die Gesamtzeit eines Simulationsschrittes darstellt, die aus der Summe der Simulationszeiten der jeweiligen Zustandsmaschinen und der Kommunikation berechnet wurde. Die Zykluszeit war in allen Beispielen auf 1 ms eingestellt und stellte somit die maximal erlaubte Zeitschranke dar.

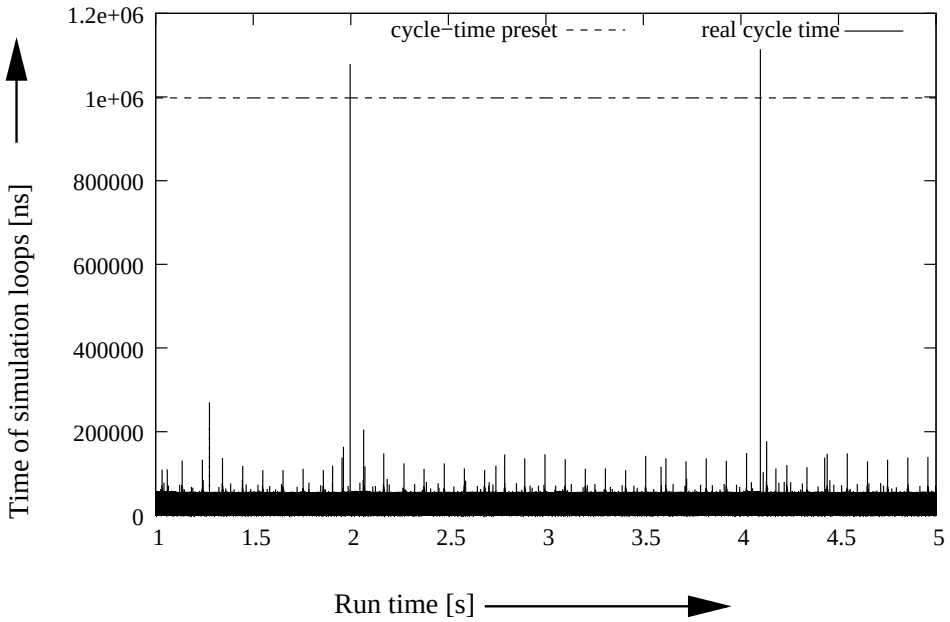
Anhand dieser Diagramme ist zu erkennen, dass Zugriffe auf Ressourcen des Linux-Kernels – vor allem bei hoher Belastung – zu Verzögerungen führen, die sich als sehr hohe und unregelmäßigen Peaks darstellen. Sogar auf dem unbelasteten System lassen sich bereits Peaks erkennen, welche die maximal zulässigen Grenze der eingestellen Zykluszeit von 1 ms deutlich überschreiten.

Jede Überschreitung der Zykluszeit würde den Verlust der Zeitsynchronität mit der Realität und inakzeptable Reaktionszeiten auf äußere Ereignisse bedeuten. Somit ist das nicht echtzeitfähige Linux-System bereits ohne Belastung nicht in der Lage, die Forderung nach Einhaltung der maximalen Zeitschranken einzuhalten. Unter Vollast wurden diese Zeitgrenzen dann um mehr als das 180-Fache überschritten.

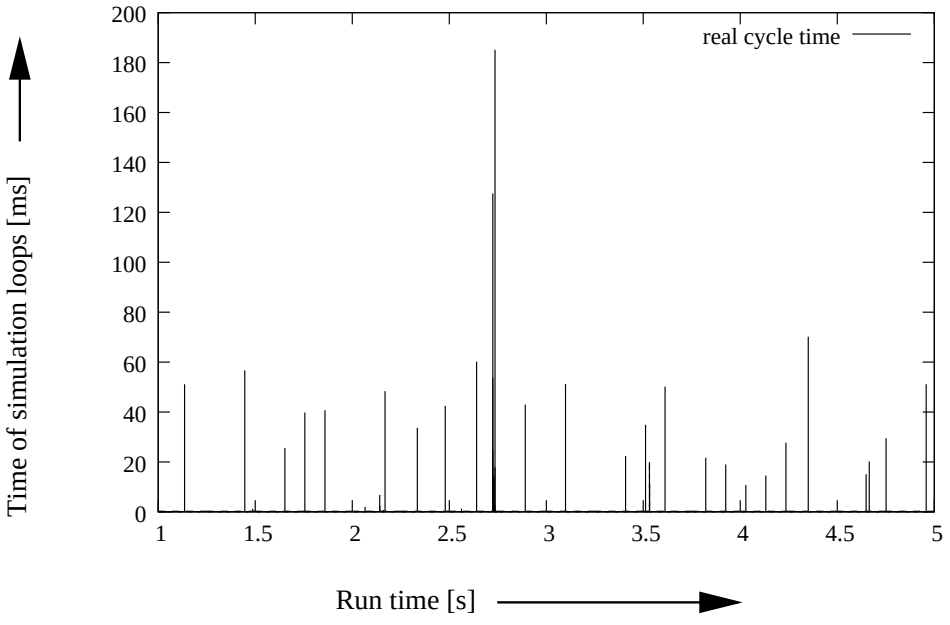
Bei der Simulation auf der echtzeitfähigen Variante von ClearSim-MultiDomain (Abbildung 5.3) zeigte sich stattdessen, dass die Verzögerungen im Vergleich sehr viel geringer ausfielen, wobei die kleinen, regelmäßigen Peaks in Bild (a) auf die Kommunikationsvorgänge der Zustandsmaschinen untereinander zurückzuführen, und somit modellbedingt waren. Die Abbildung **5.4** zeigt zur Verdeutlichung die Differenzen zwischen den beiden Messungen, mit einem Maximum von 88,152 μs und einem Mittelwert von ungefähr 46 μs.

Aufgrund der deutlich geringer ausfallenden Differenzen gegenüber der Ausführung ohne Echtzeitumgebung ist es weitestgehend auszuschließen, dass die hier ersichtlichen Einflüsse von einem Implementationsfehler herrühren. Auch die Verhinderung der Auslagerung des genutzten Echtzeit-Speicherpools, das von dem selbstimplementierten Echtzeit-Speicherhandler verwaltet wurde, konnte mit diesem Versuch erfolgreich demonstriert werden. Als Erklärungen bleiben somit lediglich unvermeidliche Verzögerungen aufgrund der auftretenden Verdrängung aus dem Cache und Kollisionen bei Buszugriffen (Bus-Locking), die zwangsläufig relativ klein sind. Neben diesen Hardware-Ursachen sind ebenfalls Software-Ursachen zu nennen, wie nicht präemptive Software-Sequenzen und abgeschaltete Interrupts im Echtzeit-Kernel. Vor allem in modernen Mikroprozessoren, die stark geschwindigkeitsoptimiert wurden und neben Caches ebenfalls spekulative Ausführung und Pipelines verwenden, werden die der Hardware zugeordneten Einflüsse besonders stark bemerkbar und intensivieren sich sogar bei hohen Taktfrequenzen, wobei die softwarebasierten Ursachen bei Erhöhung der Taktfrequenz erwartungsgemäß reduziert werden [56].

Die hier dargestellten Diagramme zeigen die Ausführungszeiten der jeweiligen Simulationszyklen. Was in dieser Betrachtung nicht enthalten ist, ist der so genannte *Scheduling Jitter*, der durch Störungen bei der Einplanung und Ausführung des zyklischen Echtzeitprozesses auftritt. Der Echtzeit-Scheduler wird zwar durch einen sehr genauen Timer gesteuert, aber die Aufrufe des Prozesses können ebenfalls durch die oben genannten Ursachen verzögert werden [57].

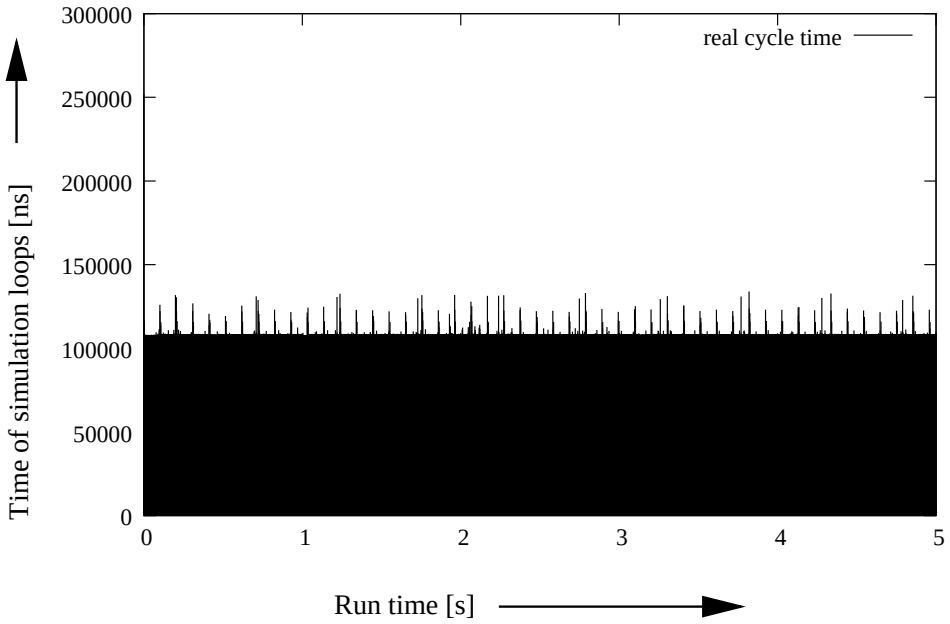


(a) Linux im Leerlauf, Zykuszeit 1 ms

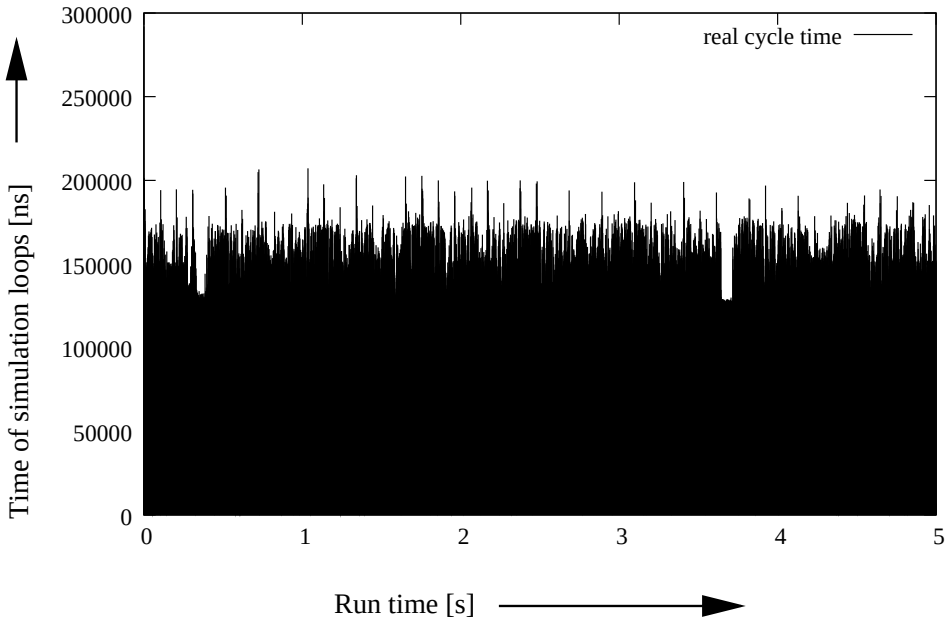


(b) Linux unter Vollast, Zykluszeit 1 ms

Abbildung 5.2: Simulation in der nicht echtzeitfähigen ClearSim-Umgebung



(a) RTAI mit Linux im Leerlaufbetrieb, Zykluszeit 1 ms



(b) RTAI mit Linux unter Vollast, Zykluszeit 1 ms

Abbildung 5.3: Simulation von ClearSim-MultiDomain auf der Echtzeiterweiterung

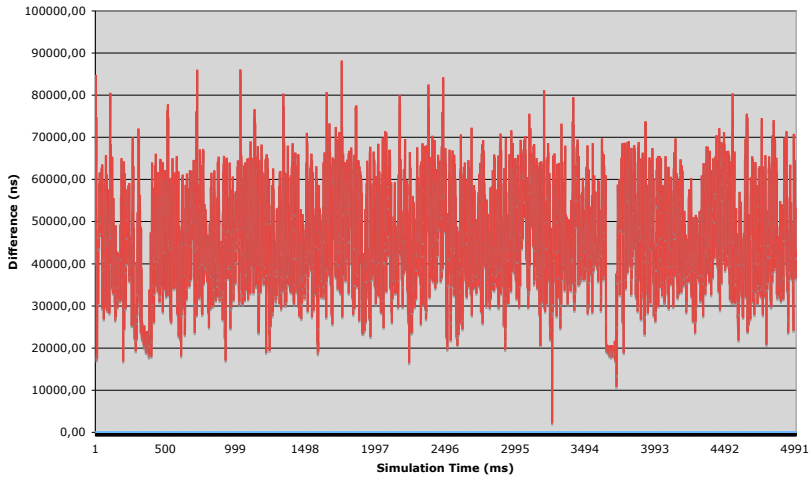


Abbildung 5.4: Differenz zwischen den Messungen von Abbildung 5.3

Um diese Art Jitter zu messen, wurde ein zyklisch eingeplanter Echtzeitprozess geschrieben, der an einem digitalen Ausgang ein Rechtecksignal erzeugt, indem bei jeder Ausführung das Ausgangssignal invertiert wird. Die hier auftretenden Verzögerungen des Prozesses waren somit direkt an dem digitalen Ausgang abzulesen. Ähnliche Verzögerungen sind bei der Echtzeitsimulation ebenfalls zu erwarten, da dort der gleiche Scheduling-Mechanismus angewendet wird.

Bei der oben schon erwähnten Kompilation des Linux-Kernels mit dem Parameter „-j“ lag der gemessene Jitter bei ungefähr 44μs (+/- 22μs). Wird auf der grafischen Oberfläche ein Fenster verschoben, konnte sogar, aufgrund der hohen Bus-Benutzung, ein maximaler Jitter von ca. 78μs (+/- 39μs) gemessen werden. Diese Zeiten sind unabhängig von der verwendeten Zykluszeit.

Wie die Werte zu beurteilen sind, wird in der Zusammenfassung dieses Kapitels näher diskutiert, nachdem im folgenden Abschnitt die Validierung des Interfaces zwischen dem virtuellen- und dem realen System besprochen wurde.

5.2 Validierung des Interfaces zwischen dem virtuellen und realen System

Wie im vorherigen Kapitel beschrieben wurde, ist eine niedrige Übergangszeit zwischen dem virtuellen- und dem realen System wesentlich für eine korrekte Simulation des Gesamtsystems. Latenzzeiten sind prinzipiell allerdings nicht zu vermeiden und müssen vom Entwickler berücksichtigt werden.

Die Größe der Latenzzeit hängt maßgeblich von der Realisierung des Interfaces und dem Abstraktionsgrad der übertragenen Information ab. Für die Übertragung einer Information aus dem realen

Teilsystem in die Simulation und umgekehrt setzt sich prinzipiell die Gesamtzeit aus der Summe der folgenden Einzelzeiten zusammen:

- Die Zeit zum Wandeln der Signale zwischen analoger- und digitaler Repräsentation.
- Die Übertragungszeit zwischen Hardwarebaustein und V/R-Interface.
- Die Zeit zur Übertragung der Daten von dem V/R-Interface zum empfangenden Simulationsmodul und umgekehrt.

Die Wandlung der Signale erfolgte über Erweiterungskarten, die über einen PCI-Bus mit dem verwendeten Industrie-PC verbunden wurden. Somit hing die Geschwindigkeit bei der Wandlung der Signale maßgeblich von der Leistungsfähigkeit dieser Karten ab:

Digital I/O: Bei digitalen I/O-Karten ist diese Zeit durch die so genannte Schaltzeit vorgegeben. Sie definiert die benötigte Zeit für einen Wechsel von einen Zustand zum anderen. Die Schaltspannung wechselte bei der hier verwendeten Karte[2] zwischen 0 V und 24 V, wobei laut Datenblatt ein Wert von ca. 20 μs für die Eingänge und typischen 200 μs (bis max. 400 μs) für die Ausgänge angegeben wurde. Reale Messungen konnten diese Zeiten bestätigen, wobei die Schaltzeit für die Ausgänge jeweils mit einem Delay von 200 μs und einer realen Signalwechselzeit bei steigender Flanke von ca. 44 μs und bei fallender Flanke von ca. 34 μs gemessen wurden. Dieser Delay von jeweils 200 μs ist durch eine Eigenschaft der Karte begründet, die ohne Detailkenntnisse der Hardware-Realisierung nicht erklärbar war. Für kürzere Schaltzeiten besteht die Möglichkeit aus einer große Anzahl von verfügbaren Karten auszuwählen, wobei z.B. bei Hochgeschwindigkeitskarten Schaltzeiten von bis zu 10 ns durchaus realisierbar sind.

Analog I/O: Für Ausgänge, die mit Digital-/Analog Wandlern bestückt sind, ist die so genannte Wandlungszeit (Conversion Time) relevant, die angibt, wieviel Zeit die Wandlung einer anliegenden Spannung in einen digitalen Wert (oder umgekehrt) in Anspruch nimmt. Bei der im Industrie-PC eingesetzen Karte wird z.B. eine Zeit von 33 μs laut Datenblatt angegeben. Zusätzlich wird für den Eingang noch die so genannte Sampling-Rate definiert, die angibt, wie häufig eine anliegende analoge Spannung aktualisiert wird. Sie betrug laut Datenblatt ca. 150 kHz. Da diese Schnittstelle für praktische Versuche nicht verwendet wurde, wurden keine Messungen zur Feststellung der realen Werte durchgeführt.

CAN-BUS: Zur Konvertierung der CAN-Nachrichten wurde ein handelsüblicher CAN-Bus Controller eingesetzt. Da die auf dem realen Bus als Spannungsimpulse übertragenen Nachrichten erst komplett empfangen werden, bevor sie an die Simulation weiter gegeben werden können, wird die beim CAN-Bus auftretende Latenzzeit maßgeblich durch die verwendete Busgeschwindigkeit definiert und stellt somit eine nicht unterschreitbare Größe dar. Da ein CAN-Bus mit maximal 1 Mbit/s betrieben wird, benötigt, bei einer Nachrichtenlänge von 47 bis 131 Bit zuzüglich der möglicherweise hinzugefügten Bit-Stuffing Bits auf dem physikalischen Layer von max. 26 bit, die Übertragung mindestens 47 μs und maximal 157 μs. Bei der Verwendung eines geringeren Bus-Taktes erhöht sich diese minimale Latenzzeit entsprechend.

[2] Inova ICP-DIO-0

Die Übertragungszeit zwischen den Hardware-Erweiterungen und dem V/R-Interface im Simulator wurden ebenfalls über Messungen ermittelt. Hierzu wurde mit einem Signalgenerator ein Rechtecksignal an den Eingang der Digital-I/O-Karte angelegt und über eine dafür geschriebene Software wieder über den digitalen Ausgang ausgegeben. Das Ausgangs- und Eingangssignal wurde mit einem Speicheroszilloskop dargestellt und die Zeitdifferenz zwischen den Signalen gemessen. Die gemessene Differenz ist auf das zweimalige Durchlaufen des PCI-Busses (32-Bit, 33MHz), sowie der verschiedenen Softwareschichten zwischen dem RT-Kernel und dem V/R-Interface im User-Space zurückzuführen und beträgt ca. 76 μs. Bei einer angenommenen Schaltzeit des Einganges von 20 μs führt das zu einer Übertragungszeit von aufgerundeten 30 μs.

Obwohl dieser Wert für digitale Signale gemessen wurde, ist er ebenfalls für analoge Werte vollkommen vertrauenswürdig, da die Nutzdaten in beiden Fällen mit jeweils 16 bit übertragen werden, was problemlos innerhalb eines PCI-Bus Taktes möglich ist. Beim CAN-Bus werden ungefähr 81 bit pro Nachricht übertragen, was somit eine minimal längere Übertragungszeit zur Folge hätte. Da der CAN-Bus auf dem hier verwendeten Industrie-PC aufgrund fehlender Treiber für Linux nicht eingesetzt wurde, konnten genaue Werte für die Übertragung nicht ermittelt werden. Bei einem CAN-Bus-basierten Versuch auf einer anderen Hardware (siehe hierzu auch Abschnitt 7.1) wurden Übertragungszeiten von ungefähr 1 ms gemessen, da die Karte dort sehr langsam über den Parallelport des Rechners angeschlossen wurde [19].

Sind die Daten im V/R-Interface angelangt, werden sie in ein ClearSim-Event umgewandelt und an den Simulations-Kernel übergeben, der sie anschließend an die angeschlossenen Simulationsmodule überträgt. Da diese Nachrichten korrekt in das vom zeitgesteuerten Simulationsalgorithmus vorgegebenen Scheduling-Raster (entsprechend Abschnitt 4.2.4) eingefügt werden müssen, tritt an dieser Stelle eine maximale Verzögerung von einem Zyklus auf.

Im nächsten Abschnitt sollen die gemessenen Werte beurteilt und eingeordnet werden.

5.3 Zusammenfassung

Wie bereits erwähnt, hängt die Nutzbarkeit dieser Simulationsumgebung von den zu erwartenden Verzögerungen ab. Diese Verzögerungen setzen sich aus den folgenden Punkten zusammen und müssen je nach Anwendungsfall bewertet werden:

1. Die minimal zu erreichende Zykluszeit in der Simulation definiert die Verzögerungen bei der Übertragung der Ereignisse über den zeitgesteuerten Simulationskernel. Sie setzt sich aus der benötigten Zeit für die Berechnung der Simulationsmodule und den Verzögerungen durch das Echtzeitsystem zusammen.
   Im Abschnitt 5.1 konnte gezeigt werden, dass die messbaren Schwankungen bei der Ausführung der jeweiligen Simulationszyklen nicht auf Implementationsfehler zurückzuführen sind. Es wurden Verzögerungen von maximal 90 μs bei hoher Belastung des Simulationssystems gemessen, die von einem Entwickler zu berücksichtigen sind. Diese Zeit ist immer als Spielraum zwischen der größten benötigten und der maximalen Zyklusdauer einzuplanen, um die harte Echtzeitfähigkeit der Simulation zu garantieren.
   Die oben durchgeführte Echtzeitsimulation mit einer Zykluszeit von 1 ms zeigt z.B. deutlich, wie stark die Dauer eines Zeitschritts praktisch reduzierbar ist. Sollte die benötigte Reaktionszeit auf Ereignisse eine geringere Zykluszeit verlangen, so könnte dort gefahrlos

ein theoretisches Minimum von ungefähr 250 μs gefunden werden, ohne die vorgegebenen Zeitgrenzen zu überschreiten.

2. Die Latenzmessungen an der Schnittstelle zum realen System zeigen, dass die Verzögerungen an drei Stellen auftreten:

   a) Die eingesetzten Schnittstellenkarten definieren maßgeblich, wie schnell Informationen die Grenze des virtuellen Systems überschreiten können. Auf niedriger Abstraktionsebene – wie bei analogen oder digitalen Signalen – entscheidet lediglich der finanzielle Aufwand, wie nahe man den physikalisch machbaren Grenzen kommt. Bei Informationen höherer Abstraktion – hier bei dem CAN-Bus – ist die minimale Verzögerung durch die Tatsache bedingt, dass eine komplette Nachricht in einem Stück an die Simulation übergeben werden muss. Somit ist die Übertragungszeit über den Bus die Konstante, die nicht unterscheidbar ist. Dies ließe sich lediglich ändern, indem die Simulation selbst auf geringerer Abstraktionsebene realisiert wäre, um direkt auf eine analoge oder digitale Schnittstelle auf dem Bus zugreifen zu können. Aufgrund der hierfür nötigen Rechenzeit und der bereits beschriebenen Verzögerungen wäre dies als Echtzeitsimulation auf den heute verfügbaren Rechnern allerdings nur schwer realisierbar.

   b) Wie oben beschrieben erzwingt die Zykluszeit des Simulationskernels die maximale Übertragungslatenz zwischen den Simulationsmodulen und somit ebenfalls die Übertragungszeit zwischen dem V/R-Interface[3] und dem empfangenden oder sendenden Simulationsmodul. Diese Zeit kann mit einfachen Möglichkeiten reduziert werden, indem das V/R-Interface nicht als eigenständiges „Simulationsmodul“ an den Simulationskernel angeschlossen ist, sondern direkt in das Modul integriert wird, das mit der Außenwelt in Kontakt treten soll. Somit wird eine Verarbeitung unabhängig von der globalen Zykluszeit ermöglicht. Dieses Konzept zur Verringerung der Latenz bei eng gekoppelten Simulationsmodulen konnte bereits erfolgreich erprobt werden [19]. Da diese Art der engen Kopplung gleichzeitig den Verlust von Flexibilität und Wiederverwendbarkeit bedeutet, sollte dieser Ansatz nur eingesetzt werden, wenn die benötigte Übertragungszeit dies erzwingt. Für die durchgeführten Anwendungsfälle war das nicht der Fall.

   c) Die übrige Verzögerung, bei der Übertragung von der Schnittstellenkarte zum V/R-Interface, beruht auf der Zeit, welche die Daten bei der Übertragung über den PCI-Bus und anschließend bei der Überführung vom Kernel-Space in den User-Space benötigen. Sie wurde in dieser Arbeit mit ungefähr 30 μs gemessen. Auf die Bus-Latenz hat der Entwickler keinen Einfluss, da sie maßgeblich von der eingesetzten Rechnerarchitektur abhängt. Theoretisch könnte lediglich der Software-Einfluss reduziert werden, indem die Prozesse im Kernel-Space, anstatt im User-Space ablaufen. Dies ist aber mit der hier verwendeten Simulationsumgebung aufgrund der in Kapitel 4 beschriebenen Gründe nicht möglich und wurde somit auch nicht weiter untersucht.

3. Bei der Betrachtung des gemessenen Jitters unter Vollast von max. 78 μs fällt sofort auf, dass diese Werte deutlich höher liegen als es in den verschiedenen Veröffentlichungen mit 1-10 μs dargelegt wurde [1, 8, 57]. Die dort dokumentierten Versuche wurden mit – aus heutiger

[3] Das ebenfalls als Simulationsmodul in die Simulation eingebunden wird.

Sicht – langsam getakteten Pentium[4] Prozessoren (200-400 MHz) im Kernel-Space durchgeführt. Neuere Untersuchungen auf aktuellen Prozessoren zeigen dagegen Werte, die mit den hier präsentierten Messungen vergleichbar sind [46] und bestätigten die Vermutung, dass die modernen PC-Architekturen bezüglich der Latenzzeiten und dem Jitter deutlich schlechtere Eigenschaften besitzen. Zusätzlich zeigen diese Messungen, wie sich die Ausführung des Echtzeitprozesses im User-Space, aufgrund des notwendigen Wechsels des Prozessors zwischen Kernel- und Userkontext, gerade bei modernen Prozessoren negativ auswirkt.

Abschließend bleibt die Frage, welche realen Systeme mit einer Echtzeitsimulation unter den hier beschriebenen Einflüssen und Randbedingungen korrekt und sicher simuliert werden können. Um diese Frage zu beurteilen, sollen die Randbedingungen typischer Anwendungsszenarien zum Vergleich herangezogen werden, die dem Autor bekannt sind:

- Ein typisches ABS-System der Firma WABCO verlangt 3 ms von der Messung der Drehgeschwindigkeit an den Radsensoren, bis zur Änderung des hydraulischen Drucks auf den Bremsen [40].
- Ein Bremssystem für Flugzeuge begnügt sich mit einer Zykluszeit von 5 ms [11].
- Die Firma DSpace beschreibt in ihren Veröffentlichungen, dass 1 ms eine typische Zykluszeit für Hardware-in-the-loop Versuche darstellt [2].
- Die Simulation eines Motor-Modells mit Kraftstoffeinspritzung zum Anschluss an einem Steuergerät konnte mit einer Zykluszeit von minimal 5 ms erfolgreich realisiert werden [39].

Anhand dieser Beispiele kann abgeleitet werden, dass eine minimalen Reaktionszeit auf äussere Ereignisse von ca. 1 ms - 5 ms für reale Anwendungsfälle durchaus ausreicht. In diesem Zeitfenster muss, neben der Simulation der Modelle, die Kommunikation durchgeführt werden. Gehen wir von einer analogen Sensormessung und von einer digitalen Ansteuerung eines Aktuators aus, dann würde bei dem hier ausgemessenen Industrie-PC eine I/O-Verzögerung von insgesamt ca. 293 μs[5] zu verzeichnen sein. Weiterhin muss ein Sicherheitsbereich von ca. 90 μs zur Einhaltung der Echtzeitfähigkeit einkalkuliert werden. Insgesamt sind somit ca. 383 μs Overhead einzuplanen. Ist eine Reaktionszeit auf äussere Ereignisse von 1 ms gefordert, bleiben somit maximal ca. 617 μs zur Berechnung aller Modelle und der Kommunikation innerhalb des Simulators.

Der Entwickler steht an dieser Stelle vor der Frage, wie komplex seine Modelle maximal sein dürfen, um in der zur Verfügung stehenden Zeit ein korrektes Ergebnis zu liefern, was aufgrund der vielfältigen Einflussfaktoren leider analytisch nicht berechenbar ist, wie Kapitel 6 noch einmal detailliert diskutiert wird. Somit bleibt nur die iterative Annäherung an ein Optimum. Um dies zu erleichtern, besitzt der Simulations-Kernel einen Mechanismus, um die jeweiligen Ausführungszeiten der Simulationszyklen exakt zu messen und in eine Datei zur späteren Analyse zu protokollieren, wie es z.B. exemplarisch mit den Diagrammen von Abbildung 5.3 und im Kapitel 7 anhand zweier Beispiele demonstriert wird. Die Ausgabe der Protokollinformationen erfolgt über die RT-Middleware und führt somit nicht zu Rückwirkungen, welche die Echtzeitfähigkeit des Systems gefährden.

[4] Welcher Prozessortyp genau verwendet wurde, wurde nicht immer dokumentiert. Oftmals handelte es sich um ein Pentium-II.

[5] analog: $33\mu s + 30\mu s$, digital: $200\mu s + 30\mu s$

Der gemessene Scheduling Jitter von maximal ca. +/- 39 μs wurde bisher in dieser Betrachtung nicht berücksichtigt, da sich dieser nicht als ständige Verzögerung, sondern als dynamische Schwankung der eingeplanten Zykluszeiten manifestiert. Bei einer Zykluszeit von 1 ms bedeutet der hier gemessene Wert, dass eine Ungenauigkeit von ca. 4 % zu verzeichnen ist, die bei weiterer Verringerung der Zykluszeit zunehmend dominant wird.

Anhand dieser Ungenauigkeit und der Zeitverzögerungen an den I/O Schnittstellen zeigen sich deutlich die Grenzen des hier beschriebenen Systems. Während die I/O-Verzögerungen der verwendeten Schnittstellenkarten noch prinzipiell reduzierbar sind, zeigen sich der Scheduling-Jitter und die Einflüsse des Prozessors und des Betriebssystems bei der hier verwendeten Systemarchitekur als nicht wesentlich verbesserungsfähig. Schnellere Prozessoren ermöglichen ausschließlich die Erhöhung der Modellkomplexität bei gleichbleibender Zykluszeit. Eine Reduzierung der Zykluszeit deutlich unter 1 ms wird aber aufgrund der zunehmend dominant werdenden Störeinflüsse, wie Cache-, Bus- und Pipeline-Latenzen, auch in Zukunft wenig realistisch bleiben, da die Systeme hauptsächlich hinsichtlich des Datendurchsatzes optimiert werden, was sich leicht anhand der mehr als 20-stufigen Pipeline eines Pentium 4 Prozessors erkennen lässt.

Diese Erkenntnis hat deutlichen Einfluss auf die realen Schnittmöglichkeiten des virtuellen Prototypen, wie sie in Abschnitt 3 beschrieben werden:

Während das Schneiden von Bussen durchaus realisierbar ist, wird es wohl auch in Zukunft kaum möglich sein, z.B. eine Taktleitung oder die Kommunikationsleitungen eines Prozessor-Cores mit seiner Peripherie zu schneiden, da dort unrealistisch kurze Zykluszeiten zur korrekten Einhaltung der Reaktionszeiten nötig wären.

Anhand der hier beschriebenen Anwendungsfälle und den vorliegenen Messungen ist aber festzuhalten, dass unser Simulationssystem für eine korrekte Simulation bis zu einer minimalen Zykluszeit von ca. 1 ms einsetzbar ist, da die Ungenauigkeiten in einem tolerierbaren Rahmen bleiben oder mit entsprechenden Maßnahmen in einen akzeptablen Bereich gebracht werden können. In diesem Zusammenhang wäre z.B. die in dem hier verwendeten Industrie-PC eingesetzte Digital-I/O-Karte für den jeweiligen Anwendungsfall kritisch zu hinterfragen und eventuell durch ein schnelleres Modell auszutauschen.

Eine korrekte Simulation verlangt zusätzlich zu den hier diskutierten Einflüssen die Ausführung der Simulationsmodule innerhalb der definierten Zeitschranken, was von der Komplexität der zu berechnenden Modelle und der Rechenleistung des verwendeten Prozessors abhängt. Auf diesen Aspekt soll in dem nächten Kapitel noch einmal eingehend eingegangen werden, bevor in Kapitel 7 anhand von Anwendungsbeispielen die praktische Einsatzfähigkeit demonstriert wird.

6 Vorhersage der maximalen Modellkomplexität für eine harte Echtzeitsimulation

Im vorherigen Kapitel wird eine einfache iterative Vorgehensweise vorgestellt, die dem Entwickler ermöglicht, mittels Testsimulationen und der Protokollierung realer Modell-Ausführungszeiten eine klare Aussage zu erhalten, welche minimale Zykluszeit zu wählen ist, um eine harte Echtzeitsimulation zu garantieren. Diese Methode konnte in den hier dargebotenen Anwendungsbeispielen (siehe Kapitel 7) seine praktische Tauglichkeit beweisen.

Unglücklicherweise ermöglicht diese Methode nur eine Aussage, welche Zykluszeiten bei bereits existierenden Modellen zu wählen ist. Gewünscht ist in diesem Zusammenhang aber ebenfalls eine Aussage, wie komplex ein Modell praktisch sein darf, damit eine gewünschte Zykluszeit erreicht wird. Wäre die Zykluszeit in Abhängigkeit einer geforderten Komplexität bekannt, könnte im Vorfeld gegebenenfalls ein entsprechend leistungsstarker Simulationsrechner beschafft oder das Vorhaben von vornherein als unrealisierbar erkannt werden.

In diesem Kapitel soll die theoretische Realisierbarkeit dieser Vorhersage diskutiert werden, wobei sich die Überlegungen auf den hier relevanten Anwendungsfall stützen, bei dem ein Host-System (der Simulationsrechner) einen virtuellen Prototypen ausführt, welcher aus einem oder mehreren Target-Systemen (z.B. Mikrocontroller, Speicherprogrammierbare Steuerung (SPS), aber auch Zustandsmaschinen) besteht.

Im nächsten Abschnitt soll mit der Einführung der grundlegenden Zusammenhänge begonnen werden, wobei wir die Betrachtung zur Verdeutlichung auf einen einfachen Fall reduzieren, in der nur ein Target-System auf einem Host simuliert wird und die Programmlaufzeit vereinfachend als Maß für die Komplexität eines Modells angesehen wird.

6.1 Grundlegende Zusammenhänge

Auf einem realen System, wie z.B. einem Mikrokontroller oder einer SPS, wird immer eine gewisse Zeit für die Verwaltung des Systems benötigt, die an dieser Stelle als *Overhead* bezeichnen werden soll. Dieser Overhead kann z.B. in Form der durch das Betriebssystem verbrauchten Rechenzeit oder – wie bei der SPS – als benötigte Zeit zur Erstellung des Prozessabbildes (die Übernahme der Sensordaten zu Beginn und die Ausgabe der Steuerinformationen zum Ende eines Programmdurchlaufs) auftreten. Diese Zeit steht für die Ausführung des eigentlichen Nutzprogramms nicht zur Verfügung. Im Folgenden soll die Ausführung des Nutzprogramms als *Nutzlast* bezeichnet werden.

Entsprechend dieser Betrachtung kann in der Simulation die Belastung eines Target-Systems als *Target-Nutzlast* L_T bezeichnet werden, während als *Grundlast* $t_{GL,T}$ der Verbrauch an Rechenzeit

zu definieren ist, der ohne Ausführung eines Nutzprogramms auftritt. Ob diese Grundlast real wirklich messbar ist, soll an dieser Stelle nicht weiter verfolgt werden.

Bei dem Host-System sieht die Unterscheidung zwischen Nutzlast und Overhead ähnlich aus:

Als Overhead sind hier das Host-Betriebssystem und der Simulations-Kernel zu nennen, wobei die Ausführung des Target-Simulationsmodells als Nutzlast zu bezeichnen ist. Analog zum Target soll die Nutzlast als L_H und die Grundlast als $t_{GL,H}$ bezeichnet werden. Weiterhin soll im Folgenden mit $t_{L,H}$ die Zeit beschrieben werden, welche auf dem Host-System die Ausführung der Target-Grundlast (bei $L_T = 0$) benötigt.

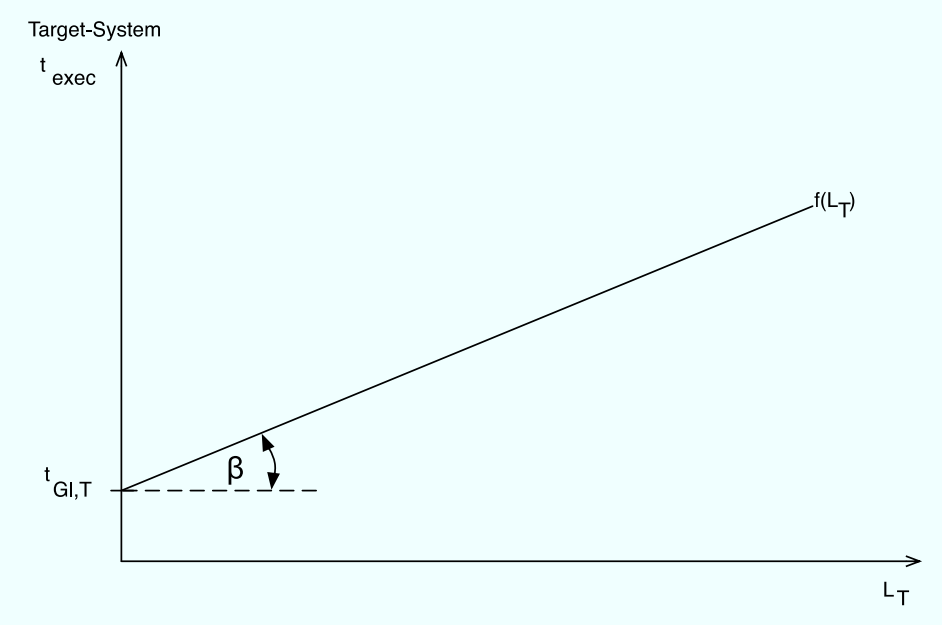
L_T und L_H stehen zueinander in einer funktionalen Beziehung, die von der Realisierung des ausgeführten Modells abhängt: So, wie die Änderung einer Last auf dem realen Target-System (L_T) die reale Ausführungszeit ($t_{exec,T}$) beeinflusst, wird ebenfalls in der Simulation die reale Ausführungszeit ($t_{exec,H}$) erhöht oder verringert. Aufgrund der unterschiedlichen Einflüsse einer Target-Laständerung auf die Ausführungszeiten von Host und Target führt dies zu unterschiedlichen Steigungen der jeweiligen Lastkurven ($f(L_T) \neq f(L_H)$). Wird von einem linearen Zusammenhang zwischen Last und Ausführungszeit ausgegangen, so kann die Steigung der resultierenden Lastgeraden als Winkel angetragen werden (hier: α und β), wie in Abbildung 6.1 (a) und (b) zusammenfassend grafisch dargestellt wird.

Eine Echtzeitsimulation ist nur möglich, so lange die hier dargestellte Gerade des Host-Systems unterhalb derjenigen des Target-Systems bleibt. Entsprechend ist der Schnittpunkt beider Geraden als Echtzeitgrenze zu definieren, da an dieser Stelle $\frac{t_{exec,T}}{t_{exec,H}} = 1$ gilt.

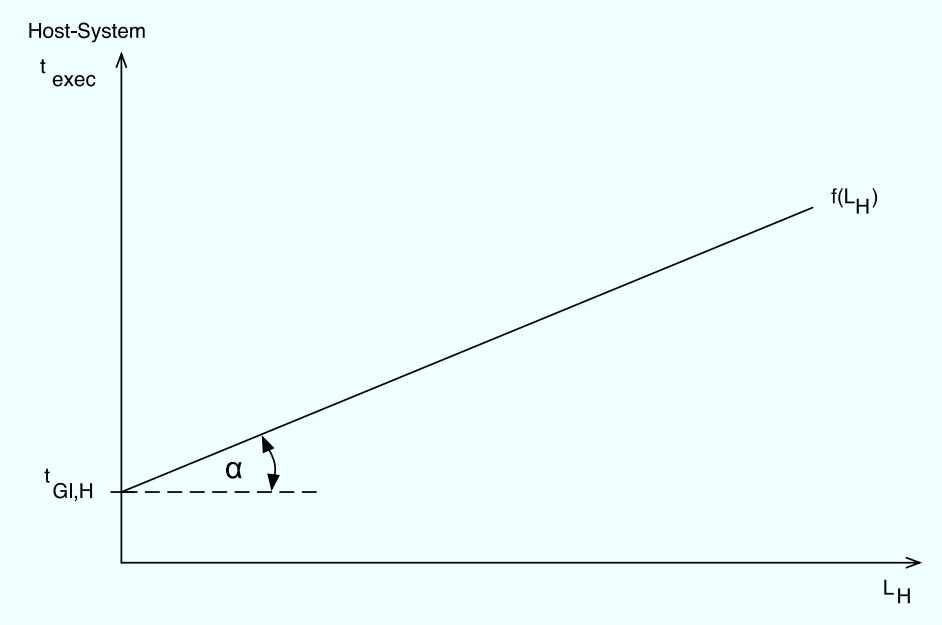
Aufgrund der möglichen Variationen zwischen $t_{GL,t}$ und $t_{GL,H}$, sowie α und β, resultieren 4 mögliche Konstellationen:

1. Ist der Simulationsrechner deutlich leistungsstärker als das Target-System, so wird für die Verarbeitung der Target-Grundlast weniger Zeit benötigt, als auf dem realen System und somit gilt $t_{GL,T} > t_{L,H}$. Die Grundvoraussetzung für eine Echtzeitsimulation ist somit gegeben. Wird für die Ausführung der Befehle des Nutzprogramms, inklusive der Auswertung des Zeitmodells und der Verwaltung der Datenstrukturen im Target-Simulator, ebenfalls weniger Zeit benötigt, als es bei der reinen Ausführung der Befehle auf dem realen Target der Fall wäre, so ist $\alpha < \beta$ und das System wird jede Last immer in Echtzeit ausführen können (siehe Abbildung 6.2).
   Diese Konstellation kann z.B. bei der Simulation eines einfachen Mikrocontrollers auftreten.

2. Wird, wie im 1. Fall, ein sehr leistungsstarkes Simulationssystem vorausgesetzt und ist somit ebenfalls $t_{GL,T} > t_{L,H}$ gegeben, aber für die Ausführung der Befehle der Last wird mehr Zeit benötigt als es auf dem realen Target der Fall ist, dann folgt aus dieser Betrachtung $\alpha > \beta$. Dies führt zur zunehmenden Aufzehrung des zeitlichen Vorsprungs, bis sich die Lastgeraden überschneiden. Ab dem Schnittpunkt wird die weitere Lasterhöhung zu einem zeitlichen Nachlaufen der Simulation gegenüber dem realen Target führen (siehe Abbildung 6.3).
   Diese Beziehung tritt z.B. bei der Simulation einer SPS auf: Während die reale SPS sehr viel Zeit für die Übernahme des Prozessabbildes und für die Ausführung des internen Betriebssystems benötigt, so ist das System sehr stark hinsichtlich der Ausführung der Programm-Operationen optimiert. Da ein SPS-Simulationsmodell für das Prozessabbild nur wenig Zeit benötigt und lediglich für das Zeitmodell Rechenzeit verbraucht (das SPS-Betriebssystem



(a) Zusammenhang beim Target-System



(b) Zusammenhang beim Host-System

Abbildung 6.1: Darstellung der Zusammenhänge zwischen Last und Ausführungszeit

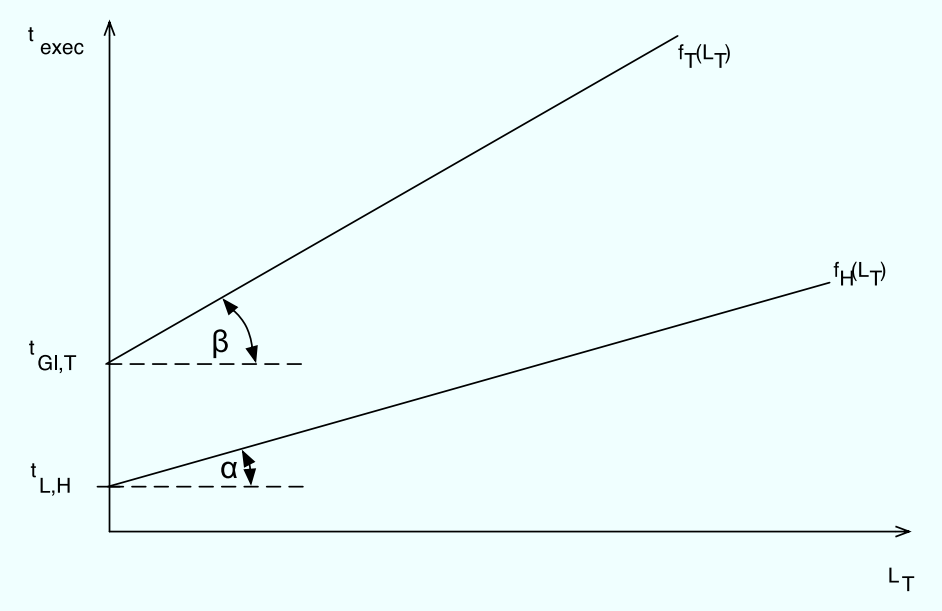


Abbildung 6.2: Möglichkeit 1: $\alpha < \beta$, $t_{GL,T} > t_{L,H}$

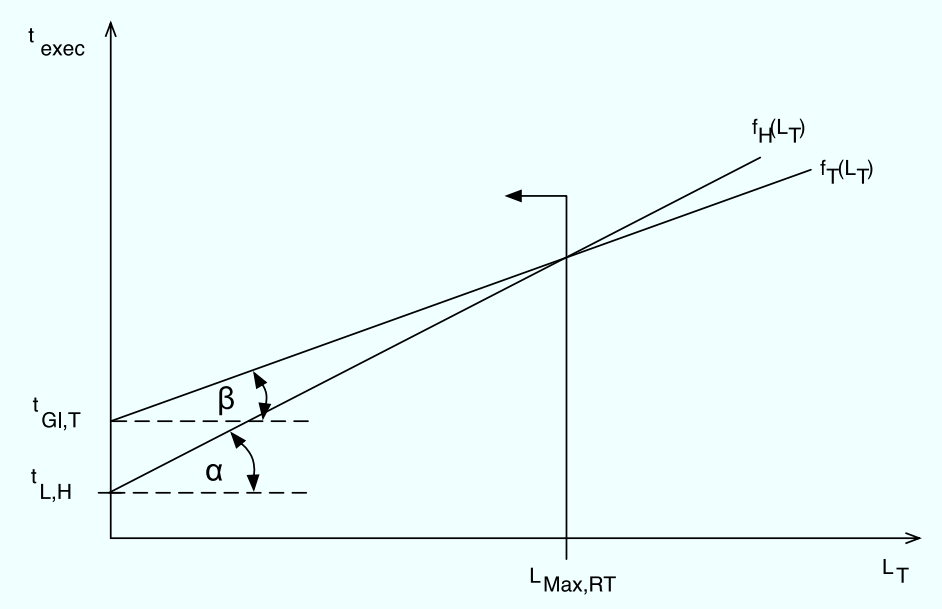


Abbildung 6.3: Möglichkeit 2: $\alpha < \beta$, $t_{GL,T} > t_{L,H}$

wird nicht benötigt), ist die Grundlast im Verhältnis sehr gering. Die im Programm enthaltenen SPS-Befehle müssen allerdings in komplexe Funktionsaufrufe umgewandelt werden, um deren Verhalten nachzubilden, was deutlich länger dauert, als auf der realen Plattform.

3. Wird für die Simulation der Target-Grundlast mehr Zeit benötigt, als die reine Ausführung auf dem realen Target kosten würde, so gilt $t_{GL,T} < t_{L,H}$. Diese Ausgangssituation kann nur zu einer Echtzeitsimulation führen, wenn für die Ausführung der Programmbefehle weniger Zeit benötigt wird, als auf dem Target. Dies führt zu $\alpha < \beta$ (siehe Abbildung 6.4) und zu einer Echtzeitfäigkeit der Simulation ab dem Schnittpunkt der Lastgeraden.

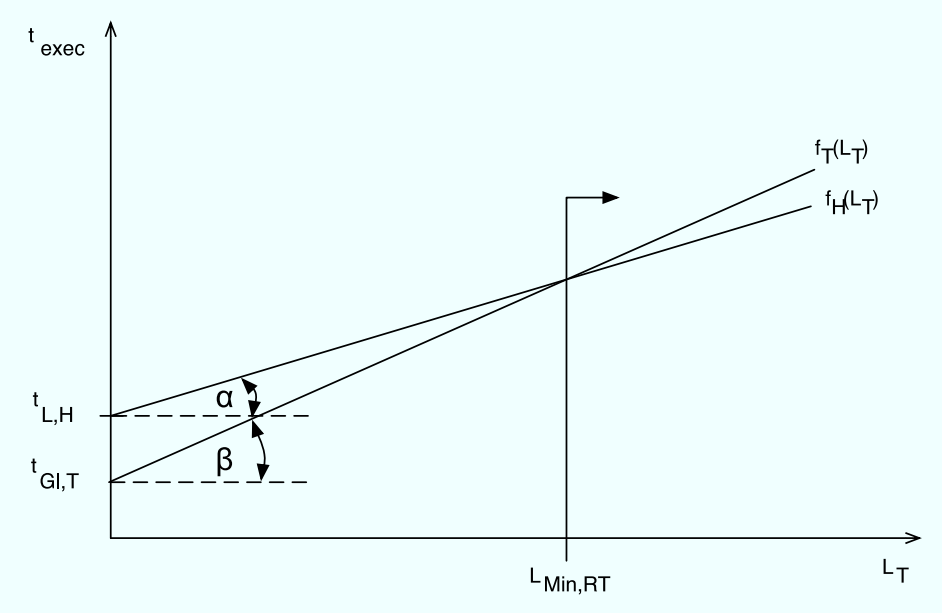


Abbildung 6.4: Möglichkeit 3: $\alpha < \beta$, $t_{GL,T} < t_{L,H}$

4. Die letzte Variante ermöglicht keine Echtzeitsimulation, da – wie in Fall 3 – $t_{GL,T} < t_{L,H}$ gilt, aber hier die Ausführung des Nutzprogramms mehr Zeit in der Simulation benötigt, als in der Realität. Somit folgt $\alpha > \beta$, was zu keiner Überschneidung der Lastkurven führt (siehe Abbildung 6.5).

Die hier demonstrierte Analyse der 4 Varianten lässt die Hoffnung aufkeimen, dass mit wenigen Messungen auf dem realen Target und auf dem Host-System die hier dargestellten Geraden aufgetragen werden können. Nach Berechnung des Schnittpunktes der Geraden wäre somit die erhoffte Vorhersage erbracht, ab welcher Komplexität des Modells (oder hier die Größe der Last) eine Echtzeitsimulation möglich wäre und wann nicht.

Bevor eine solche Ausage getroffen werden könnte, müssen aber noch einige Fragen geklärt werden:

1. Es ist nicht gesichert, ob die Lastkurven wirklich als Geraden auftreten. Dies muss für jedes Target- und Host-System jeweils analysiert werden.

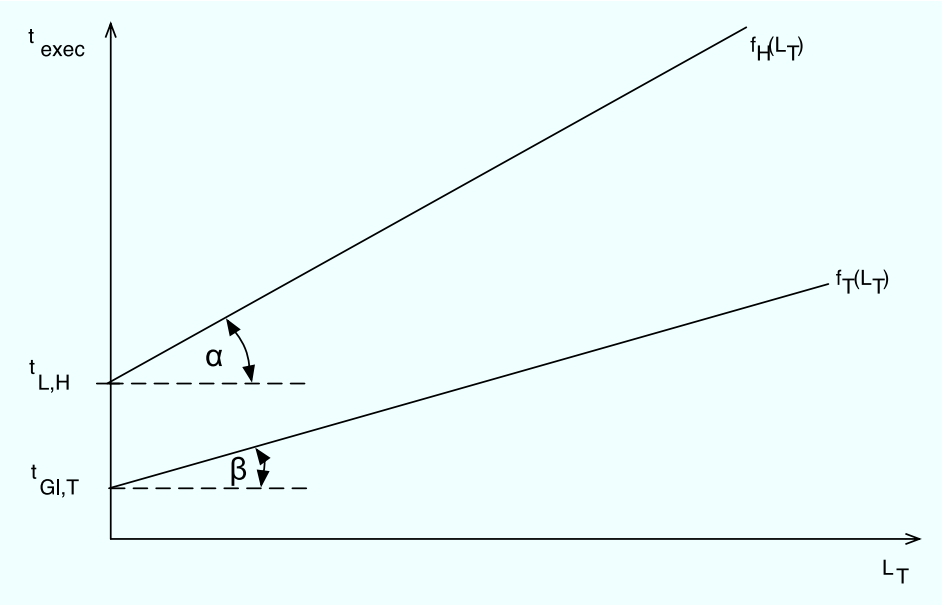


Abbildung 6.5: Möglichkeit 4: $\alpha > \beta$, $t_{GL,T} < t_{L,H}$

2. Die notwendigen Messungen am realen System sind bei abstrakten Modellen (wie z.B. einem UML-Statechart) nicht möglich.

3. Besteht ein virtueller Prototyp aus mehreren Target-Modellen, stellt sich die Frage, wie sich die Modelle gegenseitig beeinflussen.

4. Die Betrachtung der Ausführungszeit der Simulation $t_{exec,H}$ im Verhältnis zu der Zeit des echten Target-Systems $t_{exec,T}$ ist ein notwendiges aber kein hinreichendes Kriterium zur Beurteilung einer harten Echtzeitfähigkeit. In einer diskreten, zeitgesteuerten Simulation ist das Überschreiten der maximalen Zeitgrenze in allen Simulationszyklen auszuschließen.

Die erste Frage ist in der Praxis leicht über Messungen mit entsprechend vielen Messpunkten zu beantworten, so lange aus den Messpunkten anschließend ein funktioneller Zusammenhang interpoliert werden kann.

Bei abstrakten Verhaltensmodellen (Frage 2) besteht natürlich keine Möglichkeit einer Messung an einem realen System. Alternativ ist es aber leicht möglich, die real benötigte Zeit bei einem Zustandsüberhang zu messen und somit den minimalen Wert zu ermitteln, der als virtuelle Zeitfortschreitung gewählt werden kann.

Die Beantwortung der dritten Frage zeigt allerdings ein Problem auf: Zur einfachen Veranschaulichung der Zusammenhänge, wurde der virtuelle Prototyp auf ein Target-Modell beschränkt. Es stellt sich aber die Frage, wie die Zusammenhänge bei komplexeren Prototypen zu beurteilen sind, wenn mehr als ein Modell in der Simulation existieren. In diesem Fall ist möglichlicherweise schon alleine aufgrund des Kommunikations-Overheads die maximale Komplexität des virtuellen Prototypen kleiner als die Summe der Einzelkomplexitäten. Somit stellt sich die Gesamtfunktion der Modellkomplexität nicht mehr nur als Funktion der Last eines Modells, sondern ebenfalls als

Funktion der Anzahl und Art der eingebrachten Modelle dar. Da die Einflüsse untereinander nicht allgemein vorhersagbar sind, ist dieses Problem analytisch nicht mehr erfassbar.

Die letzte Frage bezieht sich auf die Voraussetzung, dass bei einer zeitgesteuerten Simulation jeder Simulationszyklus innerhalb der maximalen Zeitgrenze bleiben muss. Wird die Summe der realen Ausführungseiten aller Modelle für jeden Zeitschritt aufgezeichnet, so ergibt sich eine Grafik entsprechend Abbildung 6.6.

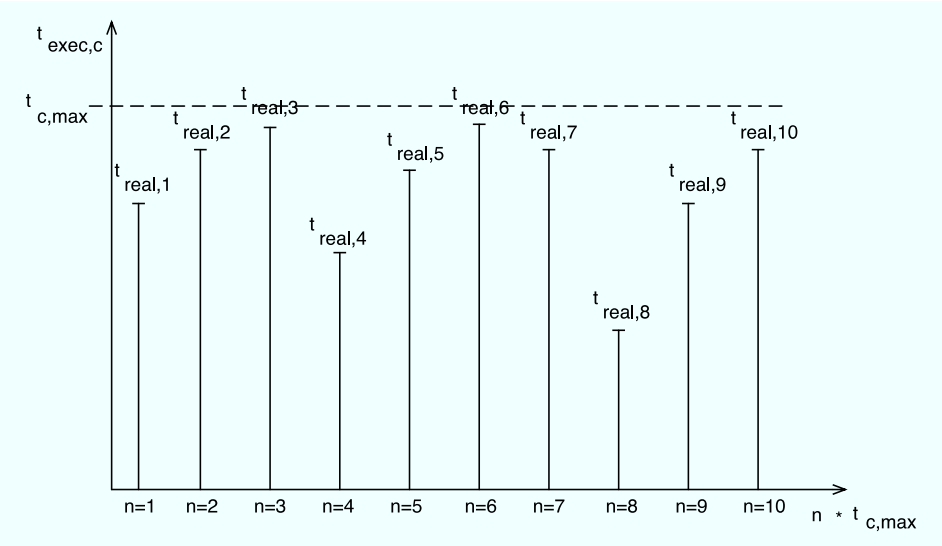


Abbildung 6.6: Darstellung der Ausführungszeiten für jeden Zeitschritt in einer Echtzeitsimulation

In dieser Abbildung wird auf der Abszisse die Simulationszeit dargstellt, die sich bei jedem Zeitschritt um den Wert der vorgegebenen Zykluszeit $t_{c,max}$ erhöht. Diese Zeit steht für alle Simulationsmodule maximal zur Verfügung, um diesen Zeitraum virtuell zu simulieren. Für die Simulation von $t_{c,max}$ benötigen diese eine reale Zeit, die als Balkenhöhe aufgezeichnet wird. Nur wenn die real benötigte Zeit immer kleiner als $t_{c,max}$ ist, kann von harter Echtzeitsimulation gesprochen werden. Die Höhe der Balken hängt nun von den folgenden Faktoren ab:

- Rechenleistung des Host-Systems: Je höher die Rechenleistung, desto kleiner die Balken
- Komplexität des Modells: Je komplexer das Modell, desto größer die Balken
- Rückwirkung des Echtzeitbetriebssystems, was sich als dynamisches Schwanken der Balkenhöhe bemerkbar macht.

Die Rechenleistung des Host-Systems ist als statischer Einfluss zu werten, da er sich zur Laufzeit der Simulation nicht ändert. Die anderen Einflüsse können dagegen höchst dynamisch sein und hängen von der Art der Modelle, der Last und den Einflüssen anderer beteiligter Modelle ab, was anhand realer Messungen in Abbildung 7.2 auf Seite 92 und Abbildung 7.12 auf Seite 102 verdeutlicht wird.

6.2 Zusammenfassung

Zusammenfassend muss festgestellt werden, dass die im vorherigen Abschitt identifizierten Abhängigkeiten und Einflussfaktoren im jedenfall zu komplex sind, um die zu Beginn formulierten Vorhersagen bezüglich eines virtuellen Prototypen praktisch durchführen zu können, sollte der virtuelle Prototyp aus mehreren Modellen bestehen.

Lediglich auf *ein* Modell bezogen ist die oben beschriebene Analyse möglich und hilfreich, und das auch nur dann, wenn das dynamische Zeitverhalten im Wesentlichen unabhängig von äusseren Einflussfaktoren ist. Dies kann z.B. von einer SPS gesagt werden, da sie bei jedem Rechenzyklus immer die gleichen Operationen ausführt, unabhängig von eventuell auftretenden externen Ereignissen. Aus diesem Grunde wurden entsprechende Messungen examplarisch anhand dieses Systems durchgeführt:

Wie bereits beschrieben, handelt es sich bei einer SPS-Simulation um die Variante vom Typ 2, bei der unterhalb einer gewissen Programmgröße von Echtzeitsimulation ausgegangen werden kann (es gilt $\alpha < \beta$ und $t_{GL,T} > t_{L,H}$). Mittels Zeitmessungen auf dem realen Target, sowie in der Simulation, konnte ein linearer Zusammenhang zwischen Last und Ausführungszeit ermittelt werden, der einen Schnittpunkt bei ca. 8000 Operationen aufweist, ab der die Echtzeitsimulation dieses Modells auf dem verwendeten Rechner nicht mehr möglich ist (siehe Abbildung 6.7).

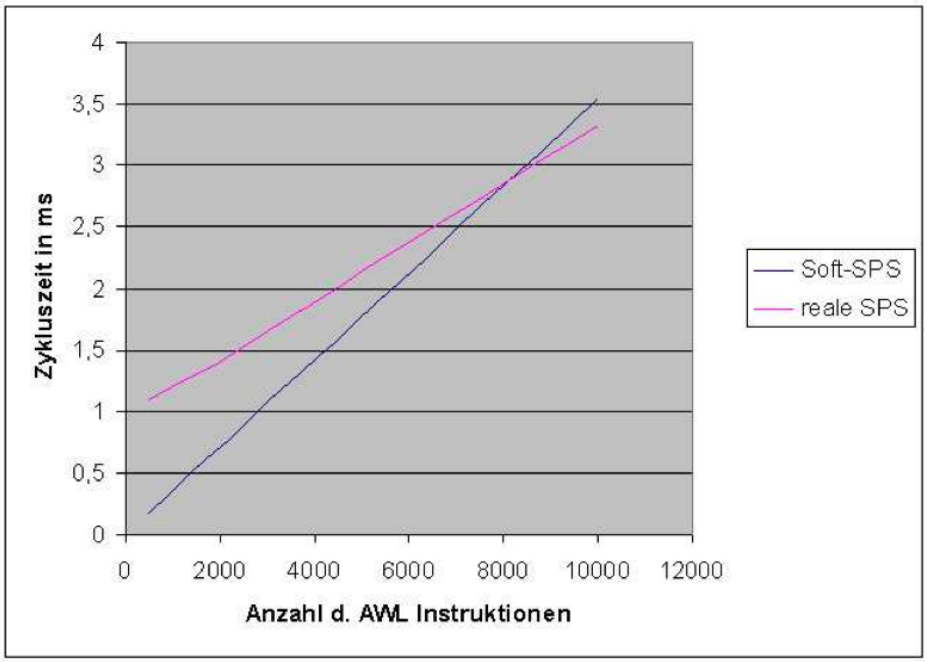


Abbildung 6.7: Vergleich der Laufzeiten zwischen echter und simulierter SPS [32]

Sollten die Modelle in ihrem dynamischen Verhalten allerdings durch äussere Ereignisse in stärkeren Maßen beeinflussbar sein, so sind die durchgeführten Vorhersagen nur von geringer Verlässlichkeit und können maximal für tendenzielle Aussagen herangezogen werden.

Aus diesem Grunde wird in den folgenden, praktischen Beispielen auf eine Vorhersage verzichtet und die im vorherigen Kapitel eingeführte iterative Messmethode verwendet.

7 Anwendungsbeispiele

Das in Kapitel 3 dargelegte Lösungskonzept beschreibt die Überführung eines virtuellen in einen realen Prototypen mittels zweier sich gegenseitig ergänzender Teilaspekte:

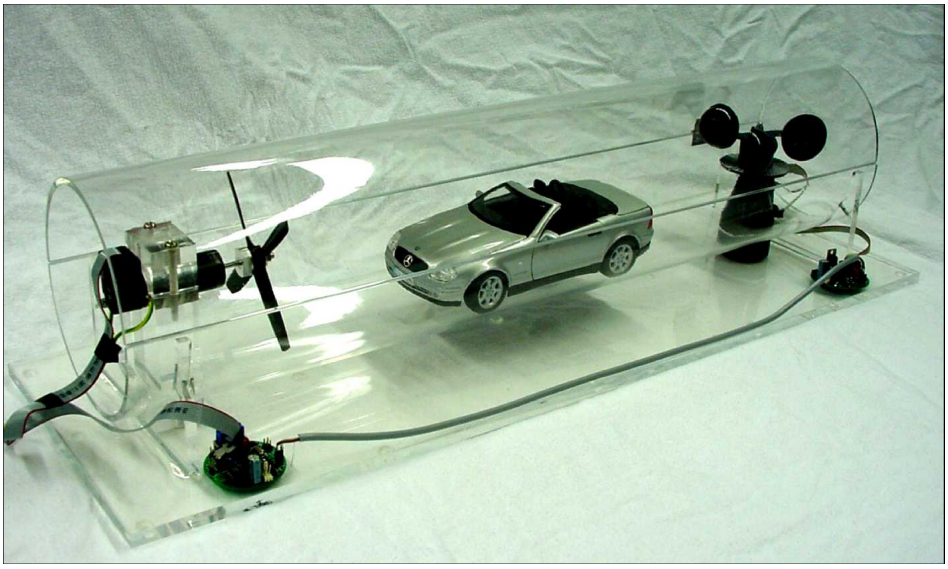
- Der virtuelle Prototyp wird an günstigen und somit lose gekoppelten Stellen aufgetrennt und schrittweise durch reale Teilsysteme ersetzt, bis das ganze System als realer Prototyp vorliegt. Die jeweiligen Mischsysteme werden ausgeführt oder simuliert und mit dem System des vorherigen Schrittes verglichen, um die Korrektheit der Transformation sicher zu stellen.
- Da die Korrektheit der Modelle im virtuellen Prototypen nicht immer bekannt ist, werden wechselseitig virtuelle und reale Repräsentationen des zu überprüfenden Teilsystems direkt und indirekt miteinander verglichen.

Diese beiden Aspekte des inkrementellen V/R-Entwurfsablaufs sollen im Folgenden anhand praktischer Beispiele dargelegt werden, um somit die Umsetzbarkeit dieses Konzeptes zu untermauern.

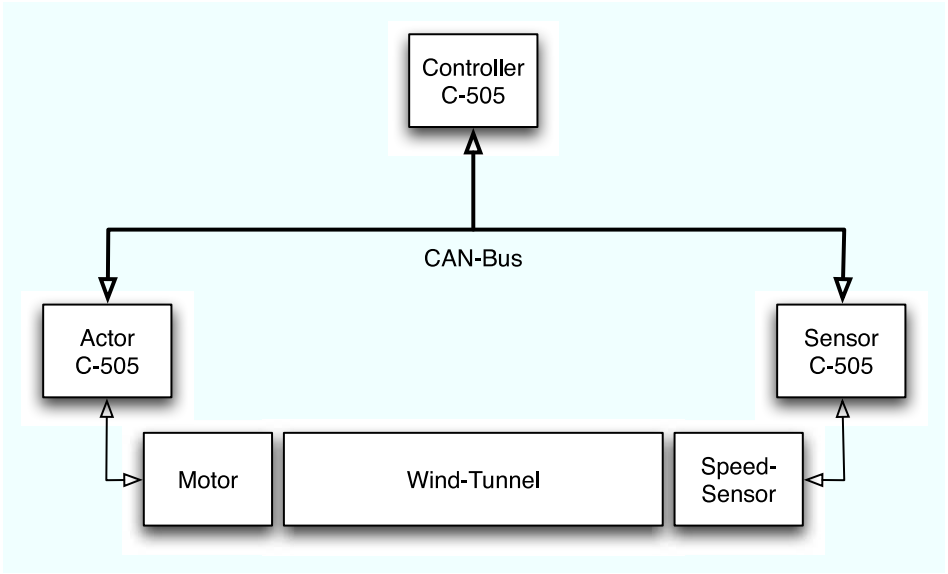
Begonnen wird mit dem schrittweisen Übergang eines virtuellen zum realen Prototypen anhand einer Windkanalregelung, wobei in diesem Beispiel von der Korrektheit der meisten Komponenten des virtuellen Prototypen aufgrund vorhergehender Modellvalidierungen ausgegangen wird. Gefolgt wird dieses Beispiel von der Realisierung einer SPS-Steuerung für eine industrielle Fertigungsanlage, bei der die Korrektheit der Modellimplementierung mittels eines Realitätsabgleichs (oder Cross Checking) demonstriert wird. Anschließend wird das Kapitel mit einer Zusammenfassung über die Erfahrungen mit diesen Beispielen abgeschlossen.

7.1 Beispiel 1: Regelung eines Windkanals

Die Regelung der Windgeschwindigkeit in einem Windkanal sollte im Rahmen eines Versuches inkrementell entwickelt werden. An einem realen Vergleichsmodell wurde ein elektrisch angetriebener Propeller und ein Windgeschwindigkeitssensor in eine Plexiglasröhre integriert. Der Propeller sowie der Sensor wurden jeweils an einem Infineon C-505-Mikrocontroller angeschlossen, um die Ansteuerung bzw. Ablesung der aktiven Komponenten durchzuführen. Ein dritter Controller sollte den zu implementierenden Regelalgorithmus tragen (siehe Abbildungen **7.1** a) und b) **)** , wobei die Controller untereinander ihre Informationen über einen CAN-Bus austauschten. Die hier verwendeten Controller-Modelle sind in der Lage, ihre Programmierung in Form eines Binärimages des Maschinenprogramms direkt auszuführen, wie es auch bei einem echten C-505 der Fall wäre. Somit ist, nach Feststellung der korrekten Funktion der Software, eine Rekompilierung für das reale System nicht nötig, was wiederum eine Fehlerquelle durch fehlerhafte Compiler zu vermeiden half.



(a) Realer Aufbau



(b) Schematischer Aufbau

Abbildung 7.1: Der Windkanal

Die gleichen Komponten wurden virtuell aufgebaut, wobei existierende sowie validierte Modelle des C-505 und des CAN-Busses eingesetzt wurden. Nur das funktionale und zeitliche Verhalten des Windkanals, inklusive der mechanischen Komponenten des Propellers und Windmessers mussten mit Modelica [66] über Differentialgleichungen modelliert und in ein Simulationsmodul überführt werden.

Da der Entwurf des PID-Reglers und die genaue Einstellung der Regelparameter die Hauptaufgabe darstellte, wurde der Entwurfsprozess in 4 Phasen unterteilt:

1. Simulation des komplett virtuellen Prototypen mit der implementierten Software für den Steuer-, Sensor- und Regelcontroller.
2. Anschließen des realen Windkanals an die Simulation. Überprüfen der Ansteuerung des realen Motors für den Propeller und der Verarbeitung der Sensorinformation des realen Windmessers. Eventuelle Korrektur der Software.
3. Anschließen der realen Steuer- und Sensorcontroller über den CAN-Bus an die Simulation. Der virtuelle Regelcontroller wird simuliert und die Regelparameter an die Eigenschaften des realen Windkanals angepasst.
4. Das in der vorherigen Stufe optimierte Regelprogramm kann in Binärform direkt auf dem realen Mikrocontroller ausgeführt werden.

Im Folgenden sollen die hier beschriebenen Phasen des Entwurfs genauer beleuchtet werden.

7.1.1 Phase 1: Aufbau und Simulation des virtuellen Prototypen

Der Aufbau des virtuellen Prototypen gestaltete sich sehr einfach, da die verwendeten Modelle für den CAN-Bus und der Mikrocontroller bereits in einer validierten Form vorlagen und somit lediglich aus einer Bibliothek entnommen und miteinander verbunden werden mussten. Die Softwareentwicklung erfolgte mit einer kommerziellen Toolchain der Firma Keil und nutzte die in den Controllern vorhandenen Peripherien zur Ansteuerung der realen Komponenten und des CAN-Busses. Schwierig und zeitaufwendig war der Versuch, die Modellierung des Windkanals zeitlich korrekt durchzuführen. Normalerweise würde der Entwickler dieses Element in einer möglichst frühen Phase des Entwurfs durch ein reales Modell ersetzen, um den Aufwand zu minimieren. In der Simulation des komplett virtuellen Prototypen wäre eine weniger exakte Version der Röhre ausreichend, um die Software auf ihre grundlegende Funktionsfähigkeit zu überprüfen.

Die komplett virtuelle Simulation des Prototypen ermöglichte die Entwicklung der Software mit komfortablen Debuggingmöglichkeiten im Einzelschrittmodus, ohne die Ergebnisse der Simulation selbst zu beinflussen, was zu den besonderen Stärken des Virtual Prototyping gehört [36].

7.1.2 Phase 2: Anschließen des realen Windkanals an die Simulation

Da zu dem Zeitpunkt des praktischen Versuchs noch keine digitale oder analoge I/O-Möglichkeit zur Verfügung stand, wurde der 2. Schritt ausgelassen, und es wurde gleich zur nächsten Phase übergegangen, in der die Verbindung über den CAN-Bus realisiert wurde.

Die prinzipielle Realisierbarkeit dieses Schrittes kann aber mit dem 2. Beispiel unten und den Untersuchungen im Kapitel 5 als gesichert angesehen werden. In diesem Fall würden die Zähler der digitalen I/O-Karte verwendet, um das Rechtecksignal des Windmessers zu zählen und so einen exakten Wert der momentenen Geschwindigkeit zu ermitteln, wie es auch auf dem realen Controller der Fall war. Weiterhin könnte der Ausgang einer Digital-/Analogkarte für die Ansteuerung der Motordrehzahl des Propellers Verwendung finden.

Da die Software auf den beiden Controllern für die Sensormessung und die Motoransteuerung in diesem Fall eher als trivial anzusehen waren, stellte das Auslassen des 2. Schrittes auch keine Beeinträchtigung des Entwurfprozesses zu diesem Zeitpunkt dar.

Die grössere Herausforderung in dieser Phase liegt in der Erreichbarkeit der Echtzeitfähigkeit der Simulation bei Einhaltung akzeptabler Zykluszeiten. Wie Abbildung **7.2** zeigt, war bei einem

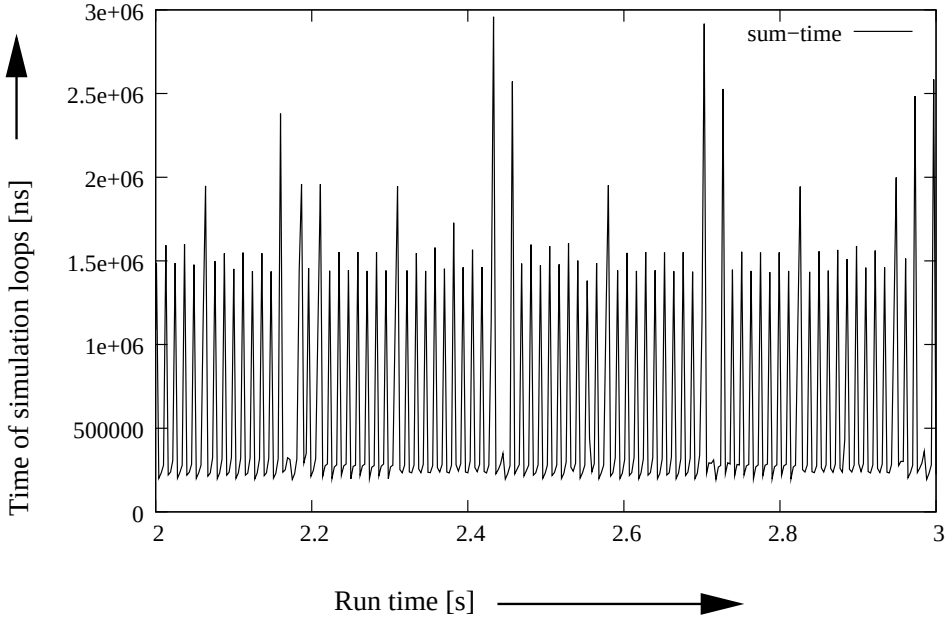


Abbildung 7.2: Summe der Ausführungszeiten aller Modellkomponeten des virtuellen Prototypen (max. Zykluszeit: 3 ms, Rechner: 2 GHz Pentium 4)

Rechner mit 2 GHz eine Simulation unter harten Echtzeitbedingungen nur bei einer Zykluszeit von mindestens 3 ms sicher möglich, da zwei der drei Mikrocontroller im gleichen Zeitschritt eine maximale Verarbeitungszeit von jeweils fast 1,5 ms benötigten, wie die Vergößerung des Zeitbedarfs bei 2.7 Sekunden Simulationszeit in Abbildung **7.3** beispielhaft zeigt.

Da die Änderung der Windgeschwindigkeit eine direkte Folge der Ansteuerung des Propellers darstellte, handelt es sich bei dem Windkanal um ein rückgekoppeltes System, bei dem die Verzögerungen des V/R-Interfaces doppelt (für Hin- und Rückweg) gerechnet werden mussten. Dies hätte bei einer Ankopplung über den Simulationskernel eine maximale Verzögerung von ca. 6 ms zur Folge. Für ein so träge reagierends System wie den Windkanal wäre dieser Wert allerdings noch akzeptabel gewesen, da die systembedingte Verzögerung (von Ansteuerung der Drehgeschwindigkeit des Propellers bis zur messbaren Auswirkung beim Windmesser vergehen ungefähr 1,4 Sekunden) deutlich höher lag (siehe Abbildung 7.4).

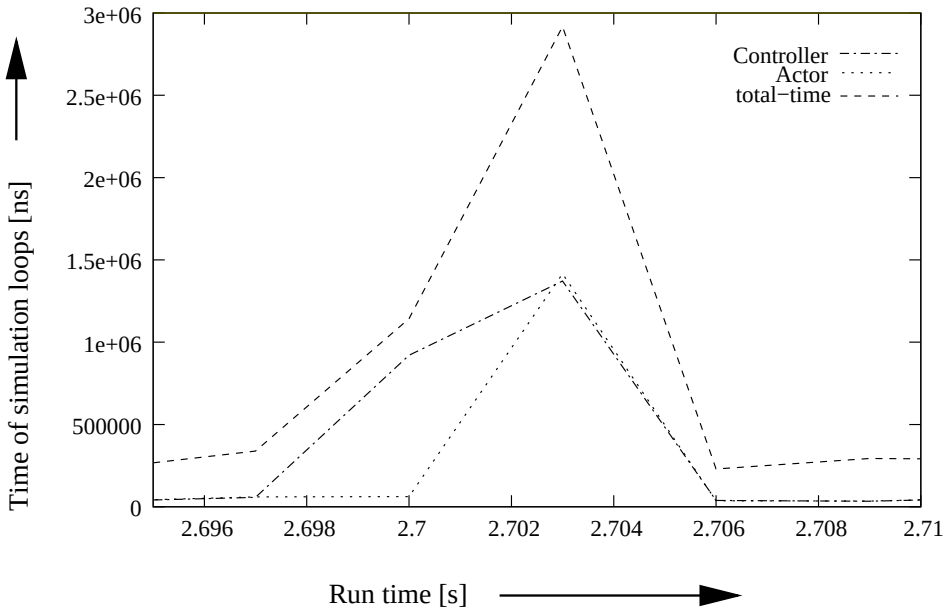


Abbildung 7.3: Vergrößerter Ausschnitt mit maximaler Zykluszeit des Reglers (Controller) und der Motoransteuerung des Propellers (Actor)

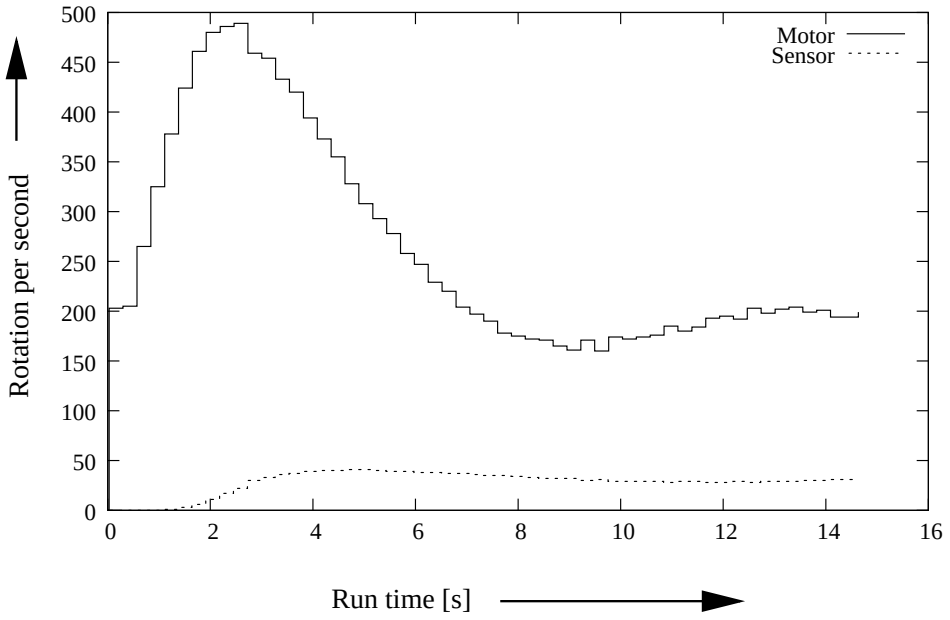


Abbildung 7.4: Vergleich der Umdrehungszahlen zwischen Motor+Propeller und Windmesser im realen System

7.1.3 Phase 3: Anschluss des virtuellen Regelcontrollers über den CAN-Bus

Die funktionale Korrektheit der Software des Controllers zur Regelung des Windkanals konnte schon in der ersten Phase validiert werden. Aufgrund der Tatsache, dass die Modellierung des Verhaltens des Windkanals nur schwer exakt möglich war, konnte zu diesem Zeitpunkt allerdings die Ermittlung der optimalen Regelparameter nicht am virtuellen Modell vollständig erfolgen.

Aus diesem Grunde wurde der virtuelle Regelcontroller in der Simulation belassen und mittels des V/R-Interfaces über den CAN-Bus mit den anderen beiden Controllern verbunden (siehe Abbildung 7.5) . Somit konnten alle Modellierungsfehler ausgeschlossen und die Simulation mit einer

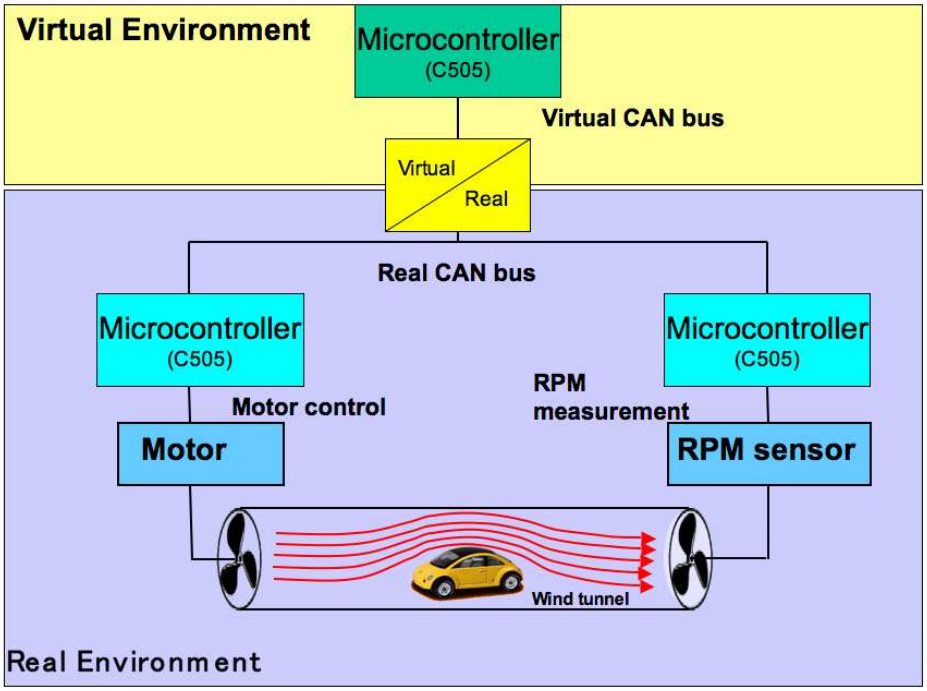


Abbildung 7.5: Mixed Prototype: Nur der Regelcontroller ist in der Simulation verblieben.

Zykluszeit von 1,5 ms ausgeführt werden. Die Verzögerungszeiten bei der Übertragung des V/R-Interfaces lagen bei dem hier verwendeten Testsystem – aufgrund einer langsamen Anbindung des CAN-Bus Controllers mit einer Latenzzeit von ca. 1 ms für eine Nachricht – bei insgesamt ca. 5 ms (rückgekoppelt), was eine Ausführung ohne signifikante Verfälschungen ermöglichte. Dies konnte erfolgreich nachgewiesen werden, indem das Modell des Windkanals an das Verhalten des realen Pendants angepasst, und das Regelverhalten zwischen dem komplett virtuellen und dem gemischten System miteinander verglichen wurde. Wie in Abbildung 7.6 ersichtlich, war das Verhalten beider Systeme in engen Grenzen vergleichbar.

7.1.4 Phase 4: Ausführen des Regelprogramms auf dem realen Controller

Bei dieser Phase wurde das Image des Binärprogramms direkt auf dem Mikrocontroller des komplett realen Prototypen ausgeführt, ohne sichtbare Unterschiede zu den Messungen in Abbildung

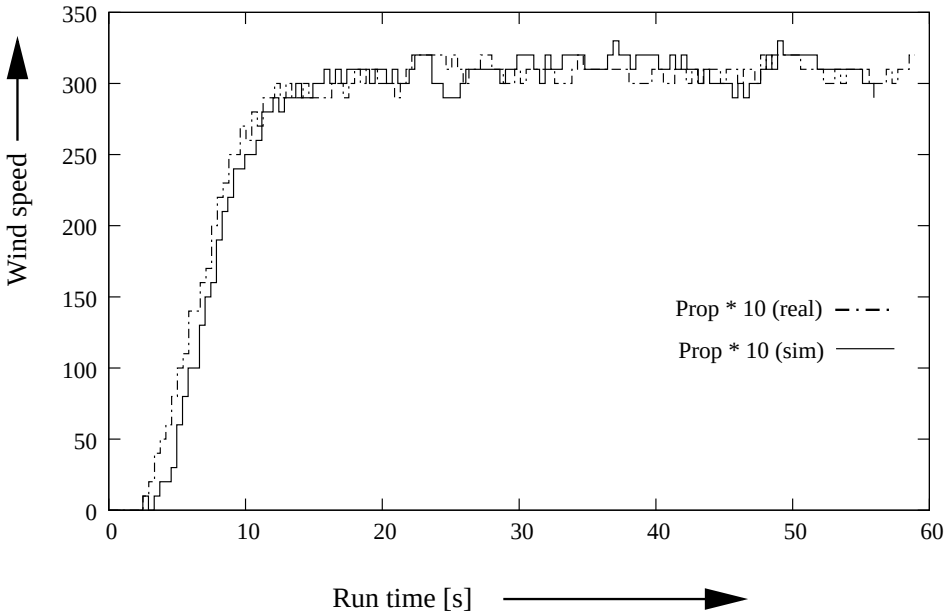


Abbildung 7.6: Vergleich der Windgeschwindigkeiten zwischen einer komplett virtuellen und einer gemischten Simulation

7.6 zu zeigen.

Somit konnte praktisch gezeigt werden, dass der inkrementelle Entwurf einen schrittweise Übergang von einem virtuellen zu einem realen Prototypen ermöglicht und somit die prinzipiellen Vorteile des Virtual Prototyping – die gute, einflussfreie Beobachtbarkeit des Systems – mit den Vorteilen des Rapid Prototypings – der Vermeidung unnötig genauer Modelle – kombinierbar ist. Weiterhin konnte gezeigt werden, dass die zusätzlichen Verzögerungen des V/R-Interfaces in diesem Fall zu keinen erkennbaren Verfälschungen führten.

7.2 Beispiel 2: Steuerung einer industriellen Fertigungsanlage

Im Rahmen einer Studienarbeit wurde die Implementierung einer SPS-Steuerung für eine im Modell vorliegende industrielle Fertigungsanlage unter Anwendung der inkrementellen Entwurfsmethode durchgeführt [32]. Das reale System bestand aus einer Modellanlage der Firma FESTO (siehe Abbildung 7.7). Diese Anlage wird aus Aktoren und Sensoren zusammengesetzt, wie sie auch in echten industriellen Anlagen vorzufinden sind.

Die grundlegende Abeitsweise dieser Anlage ist folgende:

Ein Werkstück wird über einen Drehteller nacheinander an mehreren Stationen vorbeigeführt. In der ersten Station (dem Prüfanker) wird das Werkstück auf korrekte Lochung geprüft. Nur, wenn das Werkstück ein Loch vorweist, wird es in der Folgestation (dem Entgrater) entgratet und anschließend vom Auswerfer ausgeworfen.

Die Simulationen wurden auf dem Industrie-PC durchgeführt, der in Kapitel 5 vorgestellt und ausgemessen wurde. Zur Anbindung der realen Komponenten wurde ausschließlich die digitale

I/O-Karte verwendet.

Die Steuerung sollte am Ende des Entwurfprozesses auf einer so genannten Speicherprogrammierten Steuerung (SPS) der Firma Siemens der Modellreihe Simatic S7 als AWL-Code ausgeführt werden und nicht nur den hier beschriebenen Betriebsablauf korrekt gewährleisten, sondern ebenfalls – wie in einer realen Fabrikanlage – auf fehlerhafte Sensorwerte und andere Störungen korrekt reagieren. Diese Anforderungen führten zu einer Aufgabe, die trotz des relativ einfachen Aufbaus dieser Anlage eine recht komplexe Programmierung verlangte, die mittels direkter Programmierung in dem Assemblerähnlichen AWL-Code nur schwer fehlerfrei durchführbar wäre und somit die Schädigung der Anlage und – in der Realität – von Menschenleben riskieren würde.

Der Studienarbeiter hatte die Aufgabe, diejenigen Schritte nachzuvollziehen, die ein realer Entwickler mit der hier vorgestellten Methode ausführen würde, um diese Entwurfsaufgabe korrekt zu bewältigen. In den folgenden Abschnitten sollen die jeweilgen Phase des Entwurfs genauer betrachtet werden.

7.2.1 Phase 1: Entwurf am virtuellen Prototypen

Als erster Schritt wurde der virtuelle Prototyp der Anlage erzeugt. Dies geschah über ARTiSAN Real-time Studio, indem UML-StateCharts erstellt wurden, aus denen lauffähiger C++ Code erzeugt und als ausführbares Simulationsmodul in die Simulation eingebunden wurde. Schrittweise wurden die jeweiligen virtuellen Komponenten der Anlage funktional und im Zeitverhalten implementiert und überprüft. Für den später notwendigen Blackbox-Test war es besonders wichtig sicher zu stellen, dass alle Sensorinformationen in der gleichen Weise von diesem Modell erzeugt wurden, wie sie auch in der realen Anlage in dem jeweiligen Zustand auftreten, und dass es umgekehrt auf die echten Steuerinformationen reagierte. Diese Modellierung nahm die meiste Zeit in Anspruch, da sehr viel Wert auf die Korrektheit gelegt werden musste, schließlich stellte dieses Modell die Basis für die spätere Entwicklung der Steuerung dar. Aufgrund der komfortablen grafischen Entwurfsmöglichkeiten der ARTiSAN Umgebung und der automatischen Codegenerierung wurde dieser Vorgang, im Vergleich zu einer Programmierung „per Hand“, aber wesentlich beschleunigt. Diese zwangsläufig sorgfältige Analysearbeit sollte sich später aber als positiver Nebeneffekt herausstellen, da die Anlage vom Entwickler vor seiner eigentlichen Implementationsarbeit sehr viel besser verstanden wurde, als es bei einem Entwurfsprozess der Fall wäre, bei der auf eine exakte Modellierung – wie beim Rapid Prototyping – verzichtet würde. Diese Arbeitsweise ermöglichte somit ein gezielteres Vorgehen, bei den folgenden Entwurfschritten und somit eine Beschleunigung der Arbeit.

Die Steuerung wurde ebenfalls als UML-StateChart entworfen und innerhalb der Simulation mit der virtuellen Anlage verbunden, was problemlos unter Echtzeitbedingungen mit einer Zykluszeit von 1 ms simulierbar war, wie in Abbildung 7.8 dargestellt wird.

Mit diesem System konnte die Steuerung schrittweise verbessert und von Fehlern bereinigt werden. Da die Beobachtung eines virtuellen Systems ohne geeignete Visualisierungen schwierig wäre, wurde über eine Netzwerkverbindung das auch in der Industrie angewandte Visualisierungsprogramm LabView der Firma National Instruments angeschlossen, wie in Abbildung 7.9 schematisch dargestellt wird. Diese Visualiserung erhielt ebenfalls die Signale der Datenleitungen zwischen Anlage und Steuerung und konnte somit eine realitätsnahe Abbildung der Abläufe darstellen.

In diesem Schritt wurde bewusst darauf verzichtet schon jetzt eine AWL-Implementierung der Steuerung zu wagen, da die Integration auf hoher Abstraktionsebene als StateChart die Komple-

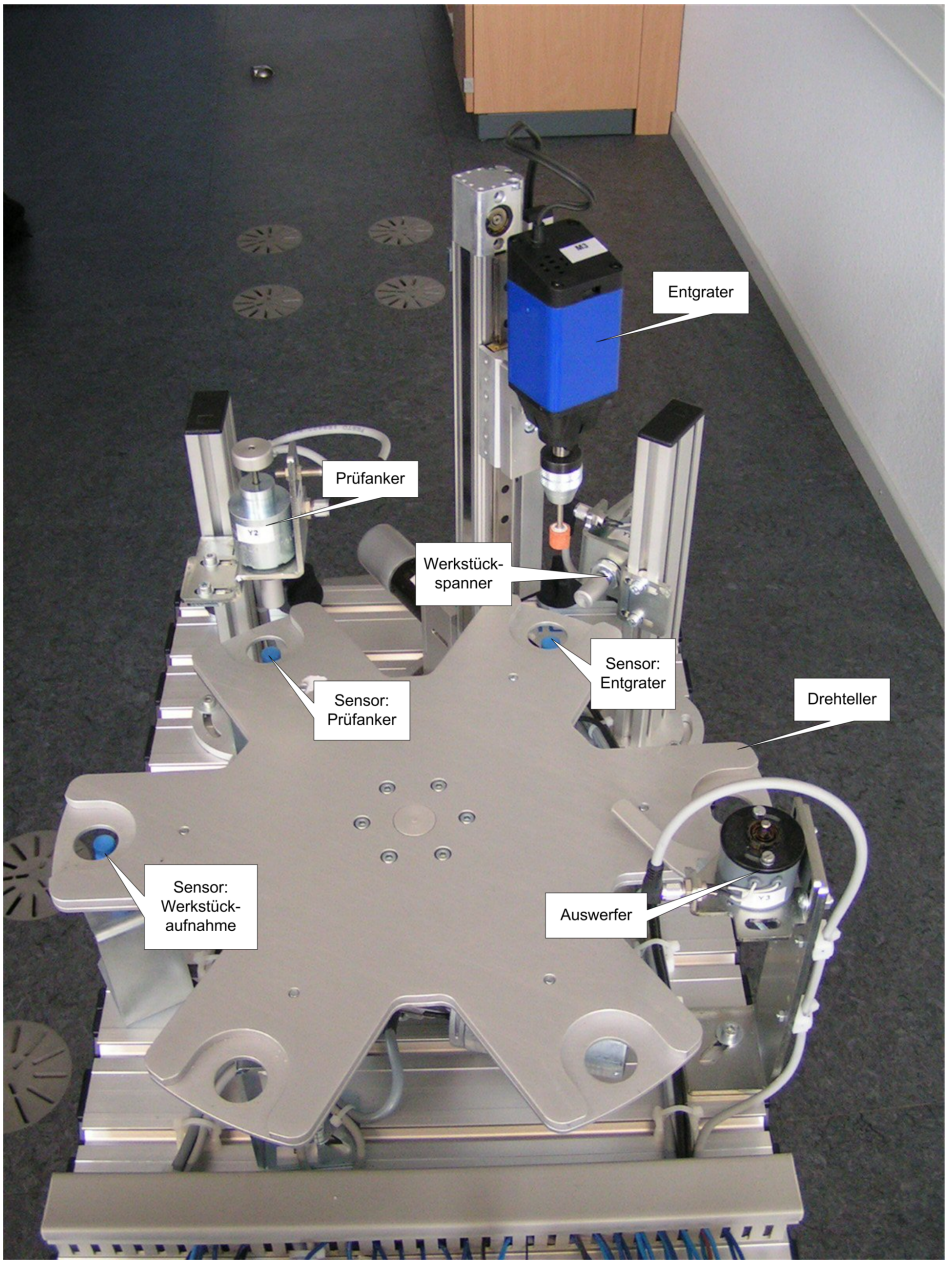


Abbildung 7.7: Die industrielle Fertigungsanlage von FESTO [32]

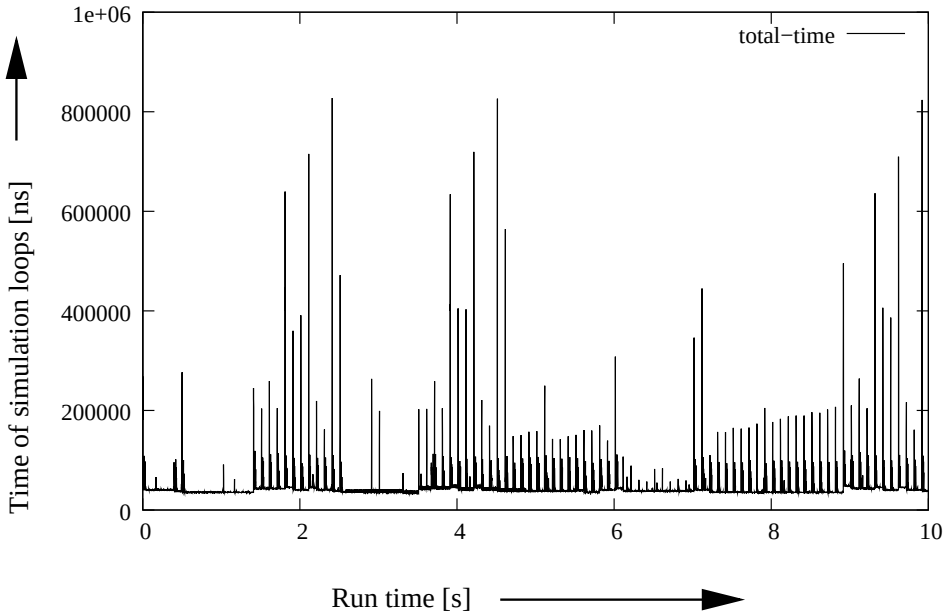


Abbildung 7.8: Darstellung der Zykluszeiten in der Simulation

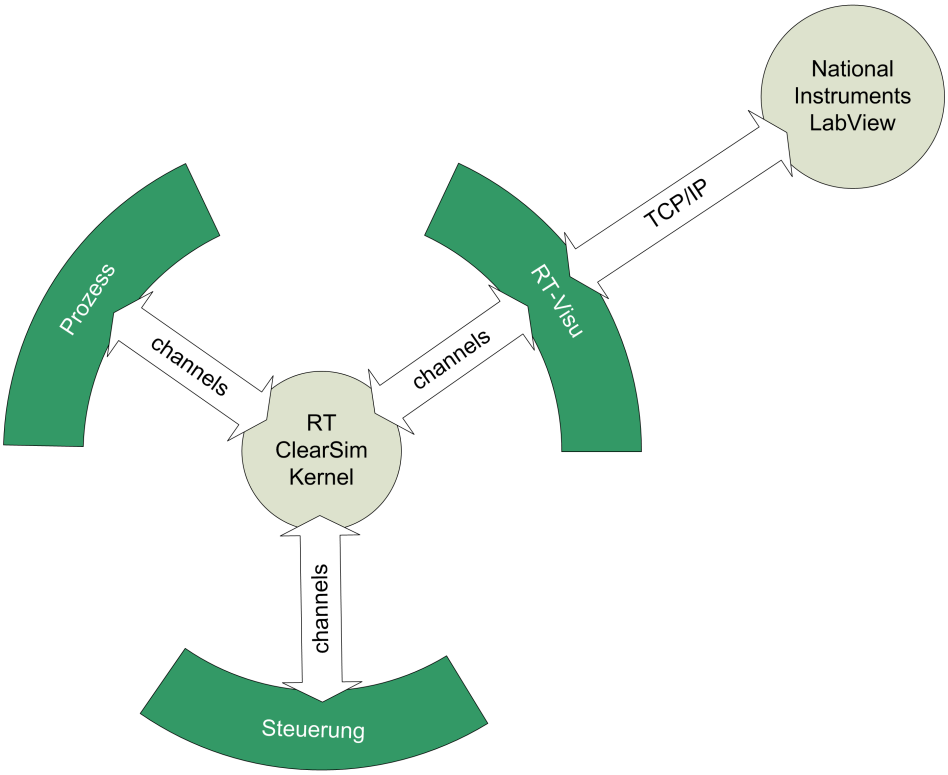
xität der Steuerung für den Entwickler beherrschbarer hielt. Am Ende bestand die finale Version der Steuerung aus 4 parallelen Zustandsmaschinen, mit jeweils ungefähr 10 Zuständen.

7.2.2 Phase 2: Übergang zum realen Prototypen

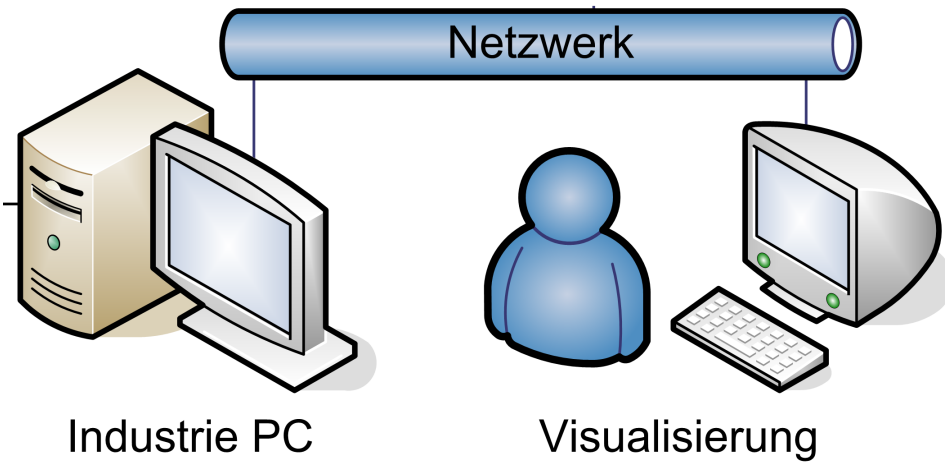
Nachdem die virtuelle Anlage korrekt angesteuert wurde, sollten ebenfalls gegenüber dem realen Pendant keine Überraschungen auftreten, falls bei der Modellierung keine Fehler gemacht wurden. Um dies sicher zu stellen, wurde die Steuerung an die reale Anlage angeschlossen (entsprechend Abbildung 7.10) und schrittweise in Betrieb genommen (so genannter Einrichtungsbetrieb), um so bei einem Fehler die Beschädigung der Anlage zu vermeiden. Das virtuelle Gesamtsystem bestand nur noch aus dem Modell der Steuerung und dem V/R-Interface, welches die Verbindung zwischen den virtuellen und realen Signalen gewährleistete. Beides konnte ebenfalls innerhalb der vorgeschriebenen Zeitkonstanten der Echtzeitsimulation ausgeführt werden (siehe Abbildung **7.11)**.

An dieser Stelle ist wichtig anzumerken, dass bei diesem Mischsystem nicht mehr die ausführlichen Tests nötig wurden, wie sie im rein virtuellen Fall bei dem vorhergehenden Schritt ausgeführt werden mussten. Der Schwerpunkt lag nicht mehr auf der Überprüfung der vollständig korrekten Funktion der Steuerung, sondern auf dem Vergleich des funktionalen und zeitlichen Verhaltens der angesteuerten Anlage mit dem Modell in der Simulation. Nur wenn die reale Anlage ein anderes Verhalten gezeigt hätte – was nicht der Fall war – hätte die Steuerung entsprechend angepasst und die aufwendigen Tests an dem realen Modell wiederholt werden müssen.

Nach dem erfolgreichen Abschluss dieser Phase konnte die UML-Version der Steuerung als ausreichend validiert angesehen werden. Würde die Ansteuerung auch in Zukunft über einen Industrie-PC vorgenommen, so hätte der Entwurfsprozess an dieser Stelle erfolgreich abgeschlossen werden

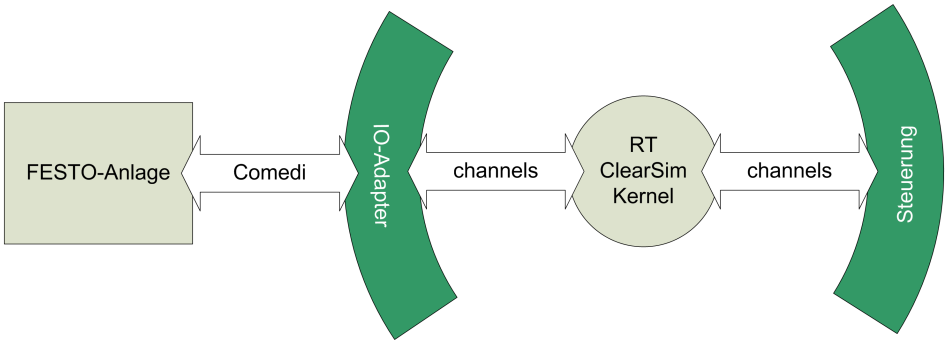


(a) Schematischer Aufbau der Simulationsumgebung

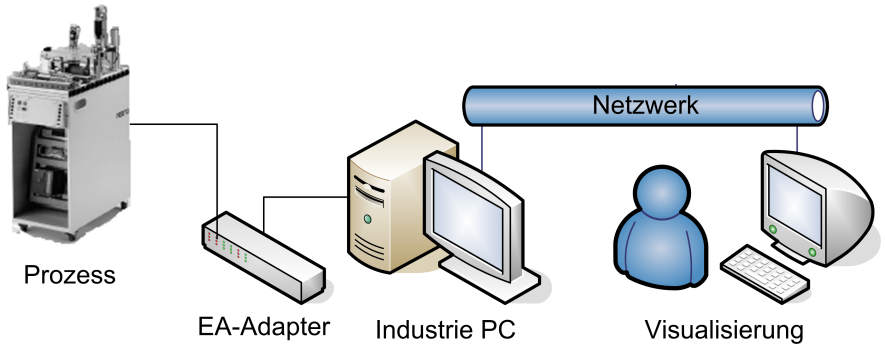


(b) Technische Darstellung des Aufbaus der Simulationsumgebung

Abbildung 7.9: Testumgebung für Mixed-Reality Simulationen [32]



(a) Schematische Aufbau der Testumgebung (ohne Visualisierung)



(b) Technische Darstellung der Testumgebung

Abbildung 7.10: Gemischter Betrieb: Virtuelle Steuerung und reale Anlage [32]

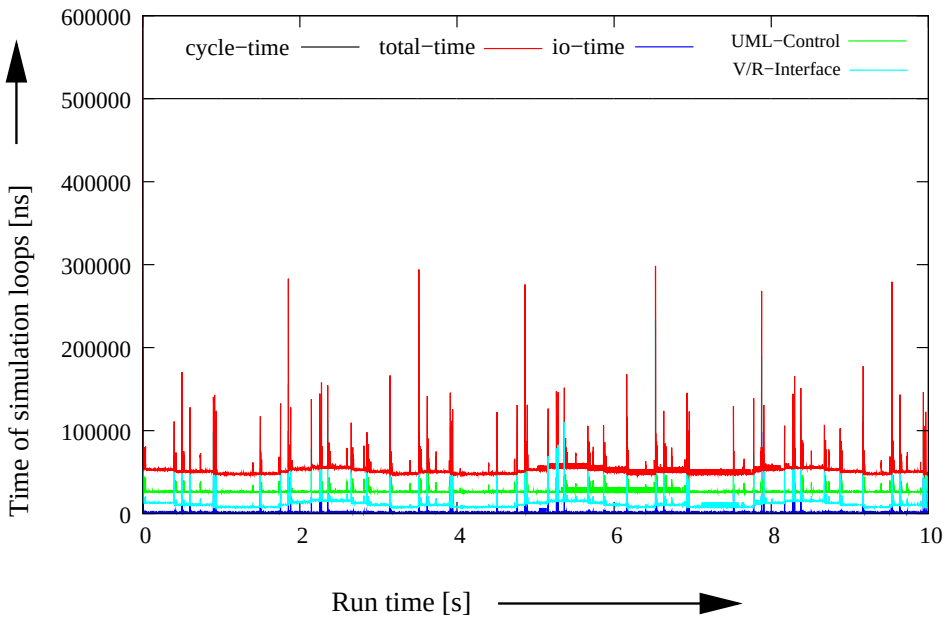


Abbildung 7.11: Simulation der UML-Steuerung mit dem V/R-Interface

können. Da in der Industrie, aufgrund der grossen Angst vor Produktionsausfällen, vorrangig SPS-Systeme eingesetzt werden, musste der Entwurfsprozess weiter geführt werden. Die Beschreibung auf hoher Abstraktionsebene musste somit in einen funktionsfähigen AWL-Code überführt werden. Erschwerend kam hinzu, dass dieser Vorgang nicht automatisiert erfolgen konnte und somit der Entwickler gezwungen war, die UML-Strukturen manuell in AWL umzuformen. Unabhängig davon, ob dieser Schritt manuell oder automatisiert erfolgte, konnte nicht davon ausgegangen werden, dass beide Modelle funktional und vom Zeitverhalten äquivalent sind, da Menschen und auch Codegeneratoren Fehler machen. Somit musste ein Weg gefunden werden, diese Äquivalenz zu sichern. Die UML-Steuerung lag ja bereits validiert vor und musste somit als korrekt angesehen werden. Die Ausführung als virtuelles Modell auf dem Simulator, zusammen mit der realen Anlage in Echtzeit, ermöglichte es nun, die Kommunikation auf den Signalleitungen zu protokollieren und somit die Grundlage für die zukünftigen Blackbox-Tests zu legen.

7.2.3 Phase 3: Validierung des Modells der Fertigungsanlage

Nachdem die validierte Steuerung wieder mit der virtuellen Fertigungsanlage verbunden wurde, konnte das erstellte Protokoll mit der vorher aufgezeichneten Version verglichen werden und so das Modell vor allem anhand des jetzt exakt bekannten Zeitverhaltens der realen Anlage angepasst werden. Funktionale Fehler wurden bei diesem Versuch nicht gefunden, wären aber sicherlich bei diesem Vergleich ebenfalls erkannt und korrigiert worden.

Dieses nun validierte Modell der Anlage konnte nun Gegenstand der nächsten Phase werden.

7.2.4 Phase 4: Implementierung des AWL-Programms für die Siemens-SPS

Da eine automatische Übersetzung der Statecharts in AWL-Code nicht zur Verfügung stand, wurde dies von dem Studienarbeiter manuell durchgeführt. Da das funktionale Verhalten der Steuerung schon durch die vorherigen Phasen bekannt war, wurde der Code in Anlehnung an die existerende UML-Realisierung implementiert. Ausgeführt wurde das Programm auf einem virtuellen SPS-Simulator mit dem Timingmodell der hier real verwendeten Simatik S7, der im Rahmen einer Masterarbeit am Institut entwickelt wurde [45]. Dieses Modell wurde mit dem validierten Modell der Anlage verbunden und wie das vorherige UML-Modell der Steuerung auf seine Funktionsfähigkeit überprüft. Da dieser Teil der Simulation komplett virtuell durchgeführt wurde, konnte diese Simulation zur besseren Beobachtbarkeit beliebig verlangsamt werden. Aber auch für eine Simulation in Echtzeit mit einer Zykluszeit von 2 ms war die Rechenleistung des Industrie-PC ausreichend, was eine Life-Beobachtung des Ablaufes über die Visualisation ermöglichte (siehe Abbildung 7.12).

Nachdem die Steuerung als AWL-Code vorlag und als funktionstüchtig gelten konnte, hätte das Programm direkt auf die reale SPS geladen werden können. Zur Sicherheit sollte der abschließende Schritt eventuelle Fehler im SPS-Simulator auschließen.

7.2.5 Abschließende Überprüfung des AWL-Programms

Das funktionale und zeitliche Verhalten der virtuellen SPS konnte zwar weitestgehend als validiert gelten. Um eine Beschädigung der Fertigungsanlage aber mit hoher Sicherheit auszuschließen,

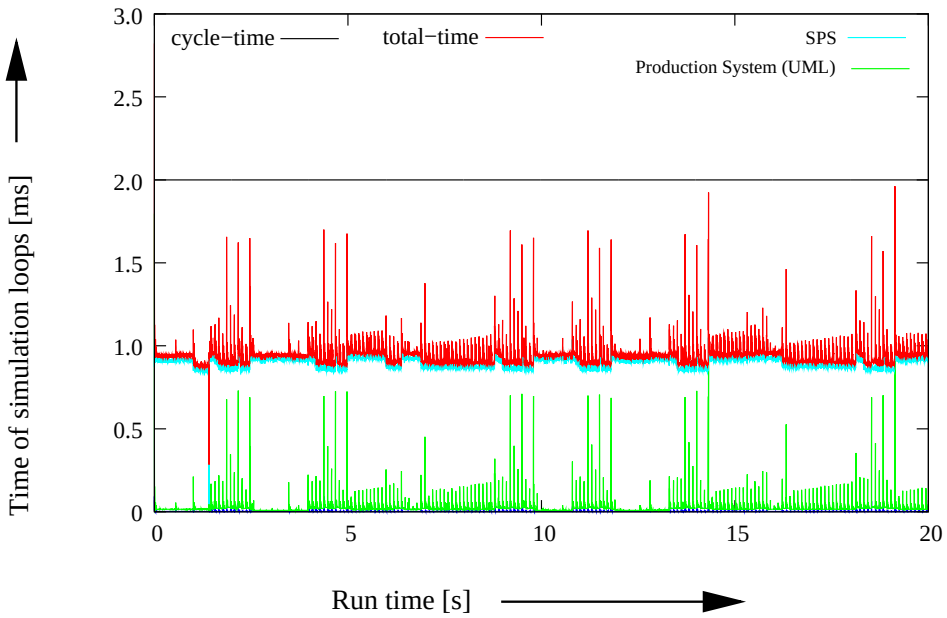


Abbildung 7.12: Virtuelle Simulation der SPS und der Fertigungsanlage in Echtzeit

wurde die reale SPS über das V/R-Interface an den Industrie-PC angeschlossen und ein abschließender Funktionstest durchgeführt. Nach dessen erfolgreichem Abschluss konnte mit sehr hoher Wahrscheinlichkeit von einer korrekten Funktion der Steuerung ausgegangen werden, was sich nach Verbindung der realen SPS mit der realen Anlage auch bestätigte.

Somit konnte der Entwurf erfolgreich abgeschlossen werden.

7.3 Zusammenfassung

Die hier demonstrierten Beispiele zeigen, dass der inkrementelle Entwurf, wie er hier vorgestellt wird, nicht nur technisch realisierbar ist, sondern auch deutliche Vorteile bringt. Die Vorteile des virtuellen Prototypings können bei dieser Methode voll ausgeschöpft werden und ermöglichen eine effiziente und exakte Überprüfung des zu entwickelnden Systems. Wo es Sinn macht, kann aber ebenfalls auf aufwendige Modellierungsarbeiten der Umgebung verzichtet werden, indem reale Modelle zu einem möglichst frühen Zeitpunkt an die Simulation angekoppelt werden, was gerade die Vorteile des Rapid Prototypings integriert, ohne Einschränkungen bei der Exaktheit des simulierten Teilsystems in Kauf nehmen zu müssen. Die Nachteile des Virtuellen-Prototyping – die Unsicherheit über die Korrektheit der Modellierung – kann ebenfalls mit dieser Methode geschickt gelöst werden, indem mittels gegenseitigem Vergleich der Modelle (Cross Checking) eine Validierung auch schon zu einem frühen Entwurfzeitpunkt möglich wird.

Die im vorherigen Kapitel beschriebenen Einflüsse durch das V/R-Interface führten bei diesen Beispielen zu keiner Beeinträchtigung der Simulationsergebnisse. Auch die maximalen Zykluszeiten bei der Echtzeitsimulation, die von der hier verwendeten Modellkomplexität und der Rechenleistung des Simulationsrechners diktiert wurde, konnte in keinem der beiden Beispielen zu einem Fehlverhalten der Simulation führen oder die Vergleichbarkeit zwischen dem teilweise virtuellen

und komplett realen System in Frage stellen. Dies kann sicherlich nicht verallgemeinert werden und muss durch den Entwickler für jeden Anwendungsfall individuell beurteilen. Die Tatsache, dass einfache Mikrocontroller oder eine komplette SPS mit einer Zykluszeit von 1,5 ms bzw. 2 ms simulierbar sind, unterstreicht die Praxistauglichkeit der Umgebung. Sollten allerdings Reaktionszeiten auf äussere Ereignisse von weniger als 1 ms verlangt werden, so stößt dieser Ansatz an seine Grenzen, da hierdurch die nötige Modellkomplexität für reale Anwendungen schwierig einzuhalten wären. Deutlich schnellere Rechner und andere Ansätze wie Parallelisierung können hier in Zukunft sicherlich bezüglich der maximale Komplexität der Modelle eine deutliche Verbesserung bringen. Sollten allerdings die Echtzeitparameter wie z.B. der Scheduling-Jitter nicht ebenfalls verbessert werden, ist die minimale Grenze der Zykluszeit auch dann nicht reduzierbar.

8 Zusammenfassung und Ausblick

Heutige Entwickler stehen im zunehmenden Maße vor der Aufgabe, komplexe heterogene Systeme beherrschen zu müssen. Diese Systeme bestehen – unter anderem – aus softwareintensiven eingebetteten Systemen, die über Schnittstellen miteinander kommunizieren und somit aufeinander Einfluss nehmen.

Bisherige Entwurfsmethoden können die hieraus resultierenden Anforderungen nicht vollkommen befriedigen. Während einerseits das *Virtual Prototyping* die Komplexität beherrschbar macht und eine sehr gute Beobachtbarkeit des Systemverhaltens ermöglicht, ist es andererseits nicht in der Lage, eine inkrementelle Transformation des virtuellen Prototypen in ein reales System zu unterstützen. Sollten bei dem realen Prototypen Probleme auftreten, die aufgrund von Ungenauigkeiten im Modell oder anderer unberücksichtigter Einflüsse niemals ausgeschlossen werden können, so ist es kaum möglich, mittels der Simulationsergebnisse auf eine Ursache zu schließen. Ein weiterer Nachteil liegt in der Notwendigkeit, ausreichend validierte Modelle verwenden zu müssen, da sonst die Ergebnisse der Simulation nicht vertrauenswürdig sind und eventuell erst am Ende des Entwurfsprozesses (bei direktem Vergleich mit dem realen System) ein Fehler erkannt wird.

Auch das *Rapid Prototyping* ist bei dieser Problemstellung nur begrenzt eine Alternative. Es erlaubt zwar den Verzicht auf aufwendige Modellierungen und ermöglicht eine schnelle Aussage über die prinzipielle Realisierbarkeit einer Spezifikation, es kann aber nur eine unvollständige Vorhersage des zukünftigen Systemverhaltens bezüglich des zu verwendenden Target-Systems bieten.

Das Hauptziel dieser Arbeit war, die Vorteile des Virtual Prototypings zu erhalten und mit den Vorteilen des Rapid Prototypings zu vervollständigen. Es entstand eine Entwurfsmethode, die als inkrementeller V/R-Entwurfsablauf bezeichnet wird und mittels gemischt virtuell-realer Systemsimulation die ständige Überprüfung der Korrektheit des Systementwurfs gewährleistet, während das virtuelle System schrittweise in ein reales System überführt wird. Weiterhin ermöglicht es – über das so genannte Cross Checking – eine indirekte Validierung der eingesetzten Modelle schon zu einem frühen Entwurfszeitpunkt.

Zur Demonstration der Realisierbarkeit und zur Feststellung der praktischen Grenzen des Konzepts, wurde eine Beispielimplementation geschaffen, indem der im Institut für Systems Engineering, System- und Rechnerarchitektur entwickelte Systemsimulator *ClearSim-MultiDomain* hinsichtlich harter Echtzeitsimulation und der Möglichkeit zur Integration realer Komponenten in die Simulation erweitert wurde.

Mittels Messungen wurden die Randbedingungen und die Grenzen der hier vorgestellten Realisierung analysiert und aufgezeigt. Es konnte gezeigt werden, dass die systembedingten Ungenauigkeiten und Verzögerungen bis zu einer geforderten Reaktionszeit auf äußere Ereignisse von ungefähr 1 ms innerhalb akzeptabler Grenzen bleiben und die hier geschaffene Lösung für den Einsatz an realen Problem-Szenarien praktisch verwendbar ist. Anhand zweier Beispiele konnte ebenfalls demonstriert werden, wie die in dieser Arbeit formulierte Entwurfsmethode in der Praxis umgesetzt wird und effizient zu einem fehlerfreien Entwurf führt.

Der zu erwartende Anstieg an Rechenleistung wird in Zukunft die in Echtzeit simulierbare Modellkomplexität weiter ansteigen lassen oder die minimalen Zykluszeiten reduzieren. Da die zukünftigen Rechnerarchitekturen aber wahrscheinlich nicht bezüglich ihres Echtzeitverhaltens optimiert werden, ist die hier beschriebene Grenze von ungefähr 1 ms voraussichtlich auch bei neuen Rechnersystemen nicht wesentlich reduzierbar.

Insgesamt lässt sich feststellen, dass es mittels des hier formulierten und praktisch realisierten V/R-Entwurfsablaufes möglich wird, einen durchgehenden Entwurf von einem virtuellen zu einem realen Prototypen zu realisieren, ohne die ständige Überprüfbarkeit des Gesamtsystems auf allen Abstraktionsebenen zu verlieren. Dies führt bei komplexen Systemen zu kürzeren Entwurfszeiten, da auch Fehlerquellen berücksichtigt werden können, die sonst erst im fertigen System sichtbar werden und somit oftmals zu teuren Verzögerungen bei der geplanten Markteinführung eines Produkts führen.

Auch wenn die hier vorgestellte Methodik bereits praktisch gut anwendbar ist, so stellt sich doch die Frage, wie die Äquivalenz bei der Transformierung der Modelle von einer Beschreibungssprache zu einer anderen und bei dem schrittweisen Übergang von einem virtuellen zu einem realen Prototypen *formal* zu gewährleisten ist, wie er in Kapitel 3 verlangt wird. In dieser Arbeit wurde auf die Fachkompetenz des Entwicklers vertraut, die eingesetzten Test-Pattern so zu gestalten, dass der erzeugte Suchraum die in der Praxis relevanten Fälle abdeckt. Dieses *Grey Box Testing* funktioniert zwar durchaus befriedigend, aber es ist dennoch die Fragestellung offen, inwieweit die Erfahrungen und Lösungen anderer Entwurfsmethoden für eine Formalisierung dieses Ablaufes herangezogen werden können. Da diese Fragestellung allein ein sehr komplexes Thema umfasst, konnte es in diesem Rahmen nicht beantwortet werden.

Weiterhin stellt sich die Frage, mit welchen Methoden die Modellkomplexität weiter erhöht werden kann. In der Vergangenheit wurde, wenn die Rechenleistung selbst nicht weiter gesteigert werden konnte, immer auf das Konzept der Parallelisierung zurückgegriffen. Dies musste in der Praxis allerdings mit hohem Overhead bezahlt werden, da die zeitliche Synchronisierung der verschiedenen Rechenknoten sehr aufwendig wurde. Dieser Nachteil kann mit einer Echtzeitsimulation theoretisch aufgehoben werden, da – so lange alle Rechenknoten in Echtzeit und mit gleich großen Zeitschritten simulieren – jegliche Zeitsynchronisierung überflüssig wird. In diesem Fall ist lediglich für einen effizienten Datenaustausch zu sorgen.

Ein weiteres interessantes Anwendungsfeld ergibt sich durch die Echtzeitsimulation komplexer heterogener Systeme, indem die Simulation zur Überprüfung von Fehlverhalten herangezogen wird. Werden die Simulationsergebnisse ständig mit dem realen Verhalten des Systems verglichen und bewertet, so könnte es zu einem Eingriff oder zur Notabschaltung durch die Bewertungseinheit kommen, sollten hier gravierende Abweichungen registriert werden.

Diese Fragen und weitere Anwendungsmöglichkeiten dieser Methode ergeben interessante Forschungsmöglichkeiten für zukünftige Arbeiten.

9 Anhang

9.1 Index

Literaturverzeichnis

[1] *Introduction to Linux for Real-Time Control*. http://www.aeolean.com/html/RealTimeLinux/RealTimeLinuxReport-2.0.0.pdf

[2] *Webside of dSPACE*. http://www.dspace.com

[3] *Webside of the Object Management Group*. http://www.omg.org/mda/

[4] *Webside of Wikipedia Encyclopedia*. http://www.wikipedia.org

[5] *Siemens C-167 Manual*. Siemens AG, 1996

[6] *Object Management Group (OMG): XML Model Interchange (XMI)*. http://www.omg.org/docs/ad/98-10-05.pdf, 1998

[7] *it+ti - Informationstechnik und Technische Informatik*. 98

[8] Aarno, Daniel: *Real-Time Linux Derivatives for Use In Robotic Control*, Department of Numerical Analysis and Computer Science (NADA), KTH, SWEDEN, Diplomarbeit, 2004. – TRITA-NA-E04006

[9] Berry, Gerard ; Gonthier, Georges: The Esterel Synchronous Programming Language: Design, Semantics, Implementation. In: *Science of Computer Programming* 19 (1992), Nr. 2, S. 87–152

[10] Briner, Jack V. J.: Parallel Mixed-Level Simulation of Digital Circuits Using Virtual Time / Duke University. 1990 (TR90-38). – Forschungsbericht

[11] Bruns, Jochen ; Müller-Schloer, Christian: An Integrated System-Level Modelling and Simulation Environment. In: *Proc. ESM 99*, 1999, S. 247–251

[12] Bruns, Jochen ; Müller-Schloer, Christian ; Scherber, Stefan: Entwurf und Validierung virtueller Prototypen heterogener eingebetteterSysteme in einer Workstation-Umgebung. In: *Proc. APS '99*. Jena, Oktober 1999

[13] Buck, J. ; Ha, S. ; Lee, E. ; Messerschmitt, D.: Ptolemy: a Mixed Paradigm Simulation/Prototyping Platform in C, C++ Conference Santa Clara, California, 1991

[14] Buck, J.T. ; Ha, S. ; Lee, E.A. ; Messerschmitt, D.G.: *Ptolemy: A Framework for Simulatingand Prototyping Heterogeneous Systems*. vol. 4. Int. journal of Computer Simulation, special issue on SSimulation Software Development", 1994

[15] Burst, A. ; Spitzer, B. ; Müller-Glaser, K.D.: On Code Generation for Rapid Prototyping Using CDIF. In: *Proceedings of OOPSLA 98*, 1998

[16] Chiodo, Massimiliano ; Giusto, Paolo ; Hsieh, Harry ; Jurecska, Attila ; Lavagno, Luciano ; Sagiovanni-Vincentelli, Alberto: A Formal Methodology for Hardware/Software Co-design of Embedded Systems. In: *IEEE Micro* 14 (1994), S. 26–36. – ISSN 0272–1732

[17] Claus, Prof. Dr. V. ; Schwill, Dr. A.: *DUDEN Informatik*. Dudenverlag, 2003. – ISBN 3–411–10023–0

[18] Daniel D. Gajski, Sanjiv Narayan Jie G.: *Specification and Design of Embedded Systems*. P T R Prentice Hall, 1994. – ISBN 0–13–150731–1

[19] Eilers, Stefan: *Extension of ClearSim-MultiDomain Towards Real-Time Simulation*, Universität Hannover, Diplomarbeit, 2000

[20] Electronic Industries Association, CDIF Technical C.: CASE Data Interchange Format - Overview / Electronic Industries Association, CDIF Technical Commitee, Extract of Interim Standard. 1997. – Forschungsbericht

[21] Ellervee, P. ; Kumar, S. ; Jantsch, A. ; Hemani, A. ; Svantesson, B. ; Öberg, J. ; Sander, I. *IRSYD - An Internal representation for System Description (Version 0.1)*

[22] Engesser, Herrmann ; Claus, Prof. Dr. V. ; Schwill, Dr. A.: *DUDEN Informatik*. Dudenverlag, 1992. – ISBN 3–411–05232–5

[23] Fischer, Franz ; Hopfner, Thomas ; Kolloch, Thomas ; Muth, Annette ; Petters, Stefan ; Färber, Georg: Rapid Prototyping of Real Time Systems using DSL. In: *iti+ti* Heft 42 (2000), S. 45 – 53

[24] Ford, Ray: Concurrent Algorithms for Real-Time Memory Management. In: *Software, IEEE* 5 (1988), S. 10–23. – ISSN 0740–7459

[25] Fujita, Masahiro ; Nakamura, Hiroshi: The standard SpecC language. In: *ISSS*, 2001, S. 81–86

[26] Fummi, Franco ; Martini, Stefano ; Perbellini, Giovanni ; Poncino, Massimo ; Ricciato, Fabio ; Turolla, Maura: Heterogeneous Co-Simulation of Networked Embedded Systems. In: *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, 2004, S. 168–173

[27] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Entwurfsmuster - Elemente wiederverwendbarer objektorientierter Software*. Addison Wesley Verlag, 1996. – ISBN 3–8273–1862–9

[28] Gupta, R. ; Micheli, G. D.: Hardware/Software Co-design. In: *IEEE Proceedings* Bd. 85, 1997, S. 349–365

[29] Heimfarth, Tales ; Gotz, Marcelo ; Rammig, Franz J. ; Wagner, Flavio R.: RTC: A Real-time Communication Middleware on Top of RTAI-Linux. In: *The Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2003)*, 2003, S. 19–25

[30] Henkel, J. ; Benner, T. ; Ernst, R.: Hardware generation and partitioning effects in the COSYMA system. In: *Proceedings of the International Workshop on Hardware-Software Codesign*, 1993

[31] Hewitt, C. E.: Viewing control structures as patterns of passing messages. In: *Journal of Artificial Intelligence* (1977). – ISSN 8–3:323–364

[32] Hoffmann, Martin: *Design und Demonstration der inkrementellen Entwurfsmethode mit dem Systemsimulator ClearSim-MultiDomain*, Universität Hannover, Diplomarbeit, 2005

[33] Jerraya, Ahmed A. ; Wolf, Wayne: Hardware/Software Interface Codesign for Embedded Systems. In: *Computer* 2/05, S. 63–69

[34] Kalawsky, R.S.: Keynote Address: The Future of Virtual Reality and Prototyping. In: *Proceedings of HCI'98*, Springer, 1999. – ISBN 3–540–76261–2

[35] Kim, K. H. ; Ishida, Masaki ; Liu, Juqiang: An Efficient Middleware Architecture Supporting Time-Triggered Message-Triggered Objects and an NT-based Implementation. In: *Second IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 1999)*, 1999, S. 54

[36] Krisp, H. ; Bruns, J. ; Eilers, S. ; Müller-Schloer, C.: Multi-Domain Simulation for the Incremental Design of Heterogeneous Systems. In: *ESM*. Prag, 2001

[37] Krisp, H. ; Müller-Schloer, C.: Virtual Prototyping of Embedded Systems by High-Level Modeling and Simulation. In: *4. GI/ITG/GMM-Workshop: Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen* Bd. Band 2. Meißen, Feb. 2001, S. 99–106

[38] Krisp, Holger ; Müller-Schloer, Christian: Objektorientierte Modellierung und Simulation eingebetteter Systeme mitClearSim-MultiDomain und UML. In: *ASIM*. Paderborn, Sep. 2001

[39] Krummen, Detlef: *Erarbeitung und Implementierung eines echtzeitfähigen Motormodells für einen direkteinspritzenden Benzinmotor*, Universität Hannover, Diplomarbeit, 1999

[40] Kruse, Alexander: *Kopplung des physikalischen Simulators dSpace mit dem Systemsimulator ClearSim*. Institute of Systems Engineering, System and Computer Architecture (SRA), 1997

[41] Kühl, Markus ; Reichmann, Clemens ; Spitzer, Bernhard ; Müller-Glaser, Klaus D.: Eine durchgehende Entwurfsumgebung fur das Rapid Prototyping von eingebetteten elektronischen Systemen. In: *it+ti* Heft 43 (2001), S. 6 ff.

[42] Levine, D. ; Facello, M. ; Hallstrom, P. ; Reeder, G. ; Walenz, B. ; Stevens, F.: STALK: An Interactive Virtual Molecular Docking System. In: *IEEE Computational Science & Engineering.*, 1996

[43] Liao, S. Y.: Towards a New Standard for System Level Design. In: *Proceedings of the Eighth International Workshop on Hardware/Software Codesign*, 2000, S. 2–7

[44] Lunze, J.: *Regelungstechnik 2 - Mehrgrößensysteme*. Springer-Verlag, 2005. – ISBN 3–540–22177–8

[45] Mahmoudi, Ghadi: *Entwicklung eines ClearSim-Moduls zur Simulation von SPSen der Simatic S7 Reihe*, Uni-Hannover, Diplomarbeit, 2003

[46] Marquet, Philippe ; Piel, Éric ; Soula, Julien ; Dekeyser, Jean-Luc: Implementation of ARTiS, an asymmetric real-time extension of SMP Linux. In: *Proceedings of the Sixth Realtime Linux Workshop*, 2004

[47] Mehl, Horst: *Methoden verteilter Simulation*. Vieweg Verlag, 1994. – ISBN 3–528–05439–5

[48] Müller-Glaser, Klaus D. ; Burst, Alexander ; Bernhard Spitzer, Markus K.: Rapid Prototyping von eingebetteten elektronischen Systemen. In: *it+ti* Heft 42 (2000), S. 8 – 15

[49] Müller-Schloer, Christian ; Scherber, Stefan ; Bruns, Jochen: Ein Werkzeug zur Bewertung von Funktion und Leistung von eingebetteten Systemen, Proc. SICAN Herbsttagung, 1997

[50] Niehaus, J. ; Damm, W. ; Metzner, A. ; Mikschl, A. ; Ossietzky, Carl von: Die EVENTS-Architektur. In: *it+ti* Heft 42 (2000), S. 40 – 44

[51] Nilsen, Kelvin D. ; Gao, Hong: The Real-Time Behavior of Dynamic Memory Management in C++. In: *IEEE Real-Time Technology and Applications Symposium*, IEEE Computer Society, 1995, S. 142–153

[52] Oodes, Tim ; Krisp, Holger ; Müller-Schloer, Christian: On the Combination of Assertions and Virtual Prototyping for the Design of Safety-Critical Systems. In: Schmeck, Hartmund (Hrsg.) [u. a.]: *Trends in Network and Pervasive Computing – ARCS 2002* Bd. 2299. Karlsruhe : Springer, 2002. – ISSN 0302–9743, S. 195–208

[53] Oodes, Tim ; Müller-Schloer, C.: A New Approach to the Design of Safety-Critical Systems based on Virtual Prototyping, Assertions and Simulation. In: *Advanced Research in Virtual and Rapid Prototyping*, Escola Superior de Tecnologia e Gestão de Leiria, 2003, S. 321 – 328

[54] P10003.1d, I. S. P.: *Draft standard for Information Technology - Portable Operating System Interface (POSIX) - Part 1: Real Time System API extension*. IEEE, 1999

[55] Passerone, Roberto ; Rowson, James A. ; Sangiovanni-Vincentelli, Alberto L.: Automatic Synthesis of Interfaces Between Incompatible Protocols. In: *Design Automation Conference*, 1998, S. 8–13

[56] Proctor, Frederick M. ; Shackleford, William P.: Real-time Operating System Timing Jitter and its Impact on Motor Control. In: *Proceedings of the 2001 SPIE Conference on Sensors and Controls for Intelligent Manufacturing II* Bd. 4563-02, 2001

[57] Proctor, Frederick M. ; Shackleford, William P.: TIMING STUDIES OF REAL-TIME LINUX FOR CONTROL. In: *Proceedings of DETC01, ASME 2001 Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, 2001

[58] Sailer, U. ; Essers, U.: *Nutzfahrzeug-Echtzeitsimulation auf Parallelrechnern mit Hardware-in-the-Loop*. Expert Verlag, 1997

[59] Schattkowsky, Tim: UML 2.0 - Overview and Perspectives in SoC Design. In: *DATE '05: Proceedings of the conference on Design, automation and test in Europe*, 2005, S. 832–833

[60] Scherber, S.: *Modellierung und Simulation software-intensiver eingebetteter Systeme*, Diss., 2001

[61] Scherber, Stefan ; Müller-Schloer, C.: Entwicklungsumgebung zur Modellierung und Simulation heterogener mechatronischerSysteme. In: *Proc. Workshop Multi Nature Systems 99*. Univ. Jena, 1999

[62] Siebenborn, Axel ; Schmitt, Stephen: Entwurf und Bewertung eingebetteter Systeme. In: *Abschlussbericht DFG-Schwerpunktprogramm 1040: Entwurf und Endwurfsmethodik eingebetteter Systeme*, 2004, S. 9–12

[63] Siegmund, R. ; Müller, D.: SystemCSV - an extension of SystemC for mixed multi-level communication modeling and interface-based system design. In: *DATE '01: Proceedings of the conference on Design, automation and test in Europe*. Piscataway, NJ, USA : IEEE Press, 2001. – ISBN 0–7695–0993–2, S. 26–33

[64] Spitzkowsky, J. ; Müller-Schloer, C.: An Execution-Driven Real-Time Simulator for Embedded Control Systems: Techniques,Applications and Evaluation. In: *Proc. ESM*. Budapest, 1996, S. 103–107

[65] Stolpe, Ralf ; Deppe, Markus ; Zanella, Mauro: Rapid-Prototyping von verteilten, hierarchischen regelungen am Beispiel eines Fahrzeugs mit hybriden Antriebsstrang. In: *iti+ti* Heft 42 (2000), S. 54 – 58

[66] Tiller, M.: *Introduction to Physical Modeling with Modelica*. Springer, 2001. – ISBN 0–7923–7367–7

[67] Varea, Mauricio: Petri Net oriented modelling and synthesis for Embedded Systems / University of Southhampton. 2000. – Forschungsbericht